A Prototype for Incremental Rule-based Stream Reasoning

Edward Tibor Toth
Matrikel-Nr. 0725631

Betreuer
O.Univ.Prof. Dr. Thomas Eiter
Dipl.-Ing. Harald Beck
## Contents

1 Introduction 2

2 Preliminaries 2
   2.1 LARS ................................................. 2
   2.2 Truth-Maintenance Systems ......................... 7

3 Stream Stratification 9

4 Extending Truth-Maintenance for LARS 10

5 Answer Update Algorithm 13
   5.1 Main Procedure ...................................... 13
   5.2 Sub-procedures ....................................... 15

6 Implementation 19
   6.1 Programming Methodology ............................ 20
   6.2 Algorithm in Scala .................................... 21
   6.3 Discussion ............................................. 25

7 Tests & Evaluation 26

8 Conclusion 29

Appendices 30

A TMS.scala 30

B TimeWindowAtomOperators.scala 39
1 Introduction

We are going to present a prototypical implementation of the answer update algorithm presented in [BDE15]. The algorithm was introduced to update results derived by stream reasoning [DCvF09] rather than recompute results from scratch. Recomputing results is highly impractical, if a stream reasoning framework is used with semantics similar to Answer Set Programming (ASP) [BET11] and thus there is a trade-off between complexity and data throughput. Instead, the answer update algorithm uses the previously calculated conclusions as input, which it updates if necessary.

For this novel implementation of the algorithm we put our emphasis on keeping the structure and the overall appearance as close as possible to the proposed algorithm. Furthermore, by using Scala we wanted to write a program that consists of readable and easy to maintain code. Also by examining the algorithm we wanted to uncover possible inconsistencies or redundancies, and correct and remove them. Additionally we wanted to simplify the algorithm, if possible. However as of writing this paper the implementation does not incorporate everything perceived in [BDE15]. It should also be noted that we did some testing on the functionality of the program, but did not yet test it on its performance.

In the preliminaries we are going to introduce a fragment of a Logic-based Framework for Analysing Reasoning over Streams (LARS) which is a rule-based modelling language [BDTEF15], and Truth-Maintenance Systems which update knowledge in a non-monotonic way [RK91]. In Sections 3 and 4 we present concepts used and/or adapted for the algorithm. Afterwards we present the algorithm on a high level and also in detail, including the sub-procedures that are used. Finally we discuss the implementation. We explain why we chose Scala as the implementation language and we are going to compare the implementation to the proposed algorithm and talk about the similarities and the differences between them. Moreover we are going to highlight some parts of the code which show interesting aspects of the program.

2 Preliminaries

In this section we are going to review some technical background.

2.1 LARS

The Logic-based framework for Analysing Reasoning over Streams (LARS) is a rule-based modelling language [BDTEF15]. It uses concepts for stream processing presented in [BW01]. To cope with the amount of data from possibly infinite streams these concepts use snapshots of recent data, which are obtained by so-called window operators (see below). Although LARS
also uses window operators to get snapshots from streams, it should be noted that its semantics only consider finite streams. Snapshots are continuously being queried by the user, either periodically or on request. LARS is also influenced by [DCvF09] which studies logical reasoning over streaming data.

When LARS was first introduced, several aspects of stream reasoning have already been investigated, although semantics of various languages [BDTEF15] were usually defined only informally. However, advanced reasoning features like non-monotonicity or non-determinism have almost exclusively been studied on static data. LARS provides a rule-based formalism and different means to abstract from time (e.g. its window operator). It has a non-monotonic semantics which can be seen as an extension of Answer Set Programming ([GL88], [FLP04], [BET11]) for stream reasoning. There are many application scenarios where stream reasoning might be used, such as in public transportation. The real time information of vehicle locations can be combined with time tables to reason about expected arrival times and show travel options.

The following examples and definitions can also be found in [BDE15] and [BDTEF15].

**Example 1** On his way to his programming class near station Y, Franz arrives at station X. He can decide to either take the tram line t or the bus line b. The tram and the bus usually need about the same amount of time to get from X to Y. Thus, Franz could just take the bus, which is expected to arrive first at Y. He remembers though that there are often traffic jams on the bus line. If that is the case, the tram could be faster.

The term *stream*, as we use it throughout this paper means that sets of atoms are mapped to time points on a timeline. Formally a stream S is a pair of a timeline \(T\) and an evaluation function \(v: N \rightarrow 2^A\).

Throughout, \(A\) denotes the set of *atoms*.

**Definition 1 (Streams)** A stream \(S = (T, v)\) consists of a closed interval \(T \in \mathbb{N}\), called the timeline, with time points \(t \in T\) and an evaluation function \(v: N \rightarrow 2^A\).

We distinguish *input atoms* from *derived atoms*. A stream is called a *data stream* if it contains only input atoms. We will refer to data streams by \(D = (T_D, v_D)\).

**Definition 2 (Substreams, Restriction)** Let \(S = (T, v)\) be a stream, with a timeline \(T\) and an evaluation function \(v\). Then a stream \(S' = (T', v')\) is a substream of \(S\), denoted by \(S' \subseteq S\), if \(T' \subseteq T\) and \(v'(t') \subseteq v(t'), \forall t' \in T'\).
Moreover, \( S|_{T'} \) denotes the restriction of \( S \) to \( T' \), i.e., the stream \((T', v|_{T'})\), where
\[
v|_{T'}(t) = \begin{cases} v(t) & \text{if } t \in T' \\ \emptyset & \text{else.} \end{cases}
\]

Streams can have very long timelines and thus hold big amounts of data. Therefore we only consider recent data and use window functions to do so. The substreams returned by window functions are called windows and can be determined by a number of parameters. Two important window functions are the sliding tuple-based and time-based window functions. The former select a specified number of the most recent tuples and the latter select all atoms within the last \( n \) time points.

**Definition 3 (Window function)** A window function is any function \( w \) which returns a substream \( S' \) of a given stream \( S \) at a time point \( t \in T \).

In this work, we only consider sliding time-based windows.

**Definition 4 (Time-based window function)** Let \( S = (T, v) \) be a stream, where \( T = [t_{\text{min}}, t_{\text{max}}] \), and let \( t \in T \). Moreover, let \( k \in \mathbb{N} \). Then, the time-based window function \( \tau \) of size \( k \) is defined by
\[
\tau^k(S, t) = (T', v|_{T'}),
\]
where \( T' = [\max(t_{\text{min}}, t-k), t] \).

Note that the window function presented in Definition 4 is also referred to as sliding time-based window function since it progresses at every time point. For a more general definition see e.g. [BDTEF15].

In the sequel, we will also write evaluation functions as a set of mappings from time points to sets of atoms. For instance, \( v = \{42 \mapsto \{a, b\}\} \) is the evaluation function where \( v(42) = \{a, b\} \) and \( v(t') = \emptyset \) for all other time points \( t' \) in the considered timeline.

**Example 2** Consider \( \tau^5 \), i.e., the time-based window function of size 5 and let \( S = ([0,9], v) \), where \( v = \{3 \mapsto \{a\}\} \). Then, \( \tau^5(S, 6) = ([1,6], \{3 \mapsto \{a\}\}) \), and \( \tau^5(S, 2) = ([0,2], \emptyset) \).

**Example 3** We can model the scenario of Example 1 with the public transport before the following way: We use a timeline \( T_D = [1, 30000] \) where each unit represents 0.1 seconds. For better readability, we write \( m \) for minutes, i.e., \( T_D = [1, 50m] \). The data stream is \( D = (T_D, v_D) \), where \( v_D \) only maps 37.2m to \textit{bus} and 39.1m to \textit{tram}, i.e., \( v_D = \{37.2m \mapsto \{\textit{bus}\}, 39.1m \mapsto \{\textit{tram}\}\} \). The time-based window of size 3 of \( D \) at time point \( t = 39.7m \) is \( S' = (T', v_D) \), where \( T' = [36.7m, 39.7m] \). Note that both atoms in \( D \) are contained in the selected timeline, thus the evaluation function remains.
Any window function \( w \) can be used in so-called window operators of form \( \ominus w \). Given a formula \( \ominus w \alpha \), \( \alpha \) will be evaluated on the window obtained by \( w \). Within the currently selected window, LARS allows to specify when a formula has to hold. In addition to streaming data, also a static set of background data \( B \subseteq A \) is considered. Formula evaluation is always relative to a given time point \( t \in T \). Let \( S = (T, v) \) be a stream and \( a \in A \) be an atom. Informally,

- \( a \) holds, if \( a \in v(t) \) or \( a \in B \),
- \( \Diamond a \) 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 \( t' \).

**Definition 5 (Extended Atom)** Let \( a \in A \) be an atom, \( t \in \mathbb{N} \) be a time point and let \( w \) be a window function. Then, the set \( A^+ \) of extended atoms is defined by the grammar

\[
A | @_t a | \ominus w @_t a | \ominus w \Diamond a | \ominus w \Box a.
\]

Any expression of the form \( \ominus w @_t a \), \( \ominus w \Diamond a \) or \( \ominus w \Box a \) is called a window atom, and \( @_t a \) is called an \( @ \)-atom.

Throughout this paper, we will only use time-based window functions and abbreviate \( \ominus \tau^k \) by \( \ominus k \) for a window operator that uses a time-based window of size \( k \).

**Example 4 (cont’d)** Given the data stream \( D \) from Example 3, \( bus \) holds at \( t = 37.2m \), and \( @_{37.2m} bus \) holds at any time point. Then, \( \ominus 3m @_{37.2m} bus \) and \( \ominus 3m \Diamond bus \) hold at all time points \( t \in [37.2m, 40.2m] \).

As mentioned above, LARS programs are extended answer set programs, and thus have a similar syntax to ASP.

A LARS program \( P \) consists of rules \( r \) of form

\[
\eta \leftarrow \beta(r),
\]

where \( H(r) = \eta \) is the head and

\[
\beta(r) = \beta_1, \ldots, \beta_i, \ not \ \beta_{i+1}, \ldots, \ not \ \beta_n,
\]

where \( n \geq 0 \), is the body. Here, \( \eta \) is an atom or an \( @ \)-atom, and each \( \beta_i \) is an atom or a window atom. We call \( B^+(r) = \{\beta_1, \ldots, \beta_i\} \) the positive body and \( B^-(r) = \{b_{i+1}, \ldots, \beta_n\} \) the negative body of \( r \).
Example 5 Figure 1 shows a program with 5 rules. For instance, rule (r1) specifies that we conclude the expectation of an arrival of a bus at station \( m \) three minutes after the arrival of a bus, if atom \( on \) holds. Note that this rule makes use of a time variable \( T \) and an addition \( "T + 3m" \) implicit in the \( @- \) operator in the head. Variables serve as abbreviations for according ground rules and arithmetic and similar operators/relations can be considered as syntactic sugar given predefined predicates.

The semantics of programs are as follows. For a stream \( S = (T,v) \), we define \( A(S) = \{ a \in v(t) \mid t \in T \} \). Given a set \( W \) of window functions and background data \( B \), a triple \( M = \langle S,W,B \rangle \) is called a structure. Let \( D = (T_D,v_D) \) be a data stream. Then, a stream \( I = (T_D,v) \supseteq D \) that extends \( D \) only by derived atoms is called an interpretation stream, i.e., no atom in \( A(I) \setminus A(D) \) is an input atom. The structure \( \langle I,W,B \rangle \) is an interpretation for \( D \). Throughout, \( W \) and \( B \) will be implicit. We assume \( W \) always contains the time-based window functions used in the considered examples. Moreover, \( B \) is empty, unless stated otherwise.

Given a structure \( M \), a time point \( t \) and \( \alpha \in A^+ \), we write \( M,t \models \alpha \), if

\[
(i) \ M,t \models \beta \text{ for all } \beta \in B^+(r) \tag{4}
\]

and

\[
(ii) \ M,t \nvdash \beta \text{ for all } \beta \in B^-(r). \tag{5}
\]

Further, \( M,t \models r \) holds iff \( M,t \models \beta(r) \) implies \( M,t \models H(r) \). If \( M,t \models r \) for all \( r \in P \), then \( M,t \models P \) holds and we call \( M \) a model of \( P \) (for \( D \) at \( t \)). Moreover, \( M \) is minimal, if there is no model \( M' = \langle I',W,B \rangle \neq M \) of \( P \) such that \( I' = (T,v') \) and \( v' \subseteq v \). Note that smaller models must have the same timeline.

Definition 6 (Answer Stream) Let \( I \) be an interpretation stream for \( D \) and let \( P \) be a program. Then, \( I \) is an answer stream of \( P \) for \( D \) at time \( t \), if \( M = \langle I,W,B \rangle \) is a minimal model of the reduct

\[ P^{M,t} = \{ r \in P \mid M,t \models \beta(r) \}. \]
The set of all such answer streams \( I \) is denoted by \( \text{AS}(P, D, t) \).

### 2.2 Truth-Maintenance Systems

Truth Maintenance Systems [RK91] hold a knowledge base of what is believed to be true, and update that knowledge, based on new incoming information. If the new information contradicts what was believed to be true, the old contradicting knowledge is discarded and updated assuming the new information to be true. This new information is treated as knowledge from now on and also can be updated in turn. On the other hand, if the new information confirms the current knowledge, the value can be kept and nothing needs to be done. By updating the knowledge base on incoming new information TMS are in effect an implementation of non-monotonic reasoning. To avoid updating correct knowledge with false new knowledge an algorithm using a TMS requires justifications which support this new knowledge. A TMS using justifications was presented in [Doy79] and is called Justification-based TMS (JTMS).

A JTMS data structure \( L \) is a pair \((N, \mathcal{J})\), where \( N \) is a set of nodes and \( \mathcal{J} \) is a set of justifications \( J \) which has the form

\[
J = \langle I | O \rightarrow n \rangle, \quad \text{where} \quad I, O \subseteq N, n \in N.
\]

We call a node \textit{in} if it is in set \( I \) and \textit{out} if it is in set \( O \). Thus we have

\[
I = \{in_1, \ldots, in_k\}
\]

and

\[
O = \{o_1, \ldots, o_l\}.
\]

A node \( n \) holds, iff all \textit{in} \( \in I \), hold, and no \textit{out} \( o \in O \), holds. We assign truth values to all \textit{in} \( \in I \) and \textit{out} \( o \in O \), where \( in = true \) (\( o = true \)) means that \textit{in} (\textit{out}) holds. This means that \( I \) is a subset of a \textit{model} \( M \subseteq N \) of a TMS data structure \( L \), and the set \( O \) has no common nodes with \( M \). The sets \( M, I \) and \( O \) form the following set \( J^M \):

\[
J^M = \{ J | I \subseteq M \text{ and } O \cap M = \emptyset \}, \quad \text{where} \quad J = \langle I | O \rightarrow n \rangle
\]

\( J^M \) consists only of those justifications that are \textit{valid} within a model \( M \). A model \( M \) can have additional properties:

For a justification \( J = \langle I | O \rightarrow n \rangle \) with \( I = \{in_1, \ldots, in_m\} \) and \( O = \{o_1, \ldots, o_n\} \) a logic program can be defined as follows [Elk90]:

\[
P_L = \{ r_J | J \in \mathcal{J} \}
\]

where \( r_J \) are rules of the form

\[
n \leftarrow in_1, \ldots, in_m, \text{not } o_1, \ldots, \text{not } o_n.
\]
Model Condition

\[ \text{founded} \quad \text{if } \forall n_i \in M \text{ there exists a total order } n_1 < \cdots < n_m, \]
\[ \text{s.t. some } \langle I|O \rightarrow n_i \rangle \in J^M \text{ with } I \subseteq \{n_1, \ldots, n_i - 1\}, \]
\[ \text{i.e. all elements in } M \text{ have a supporting function} \]

\[ \text{closed} \quad \text{if } \forall \langle I|O \rightarrow n_i \rangle \in J^M, n \in M \]

\[ \text{admissible} \quad \text{if } M \text{ is founded and closed} \]

Figure 2: Model conditions

For a set of nodes \( N \), the subset \( M \subseteq N \) is an admissible model of a JTMS data structure \( \mathcal{L} \), iff \( M \) is an answer set of \( P_{\mathcal{L}} \). See also Figure 2.2.

For a given admissible model \( M \) the TMS algorithm will try to update \( M \) on arrival of a new justification \( J' = \langle I'|O' \rightarrow n' \rangle \) to an admissible model \( M' \) of \( \mathcal{L} = (N, J \cup \{J'\}) \). This is done by using the concepts of (affected) consequences and support. We will introduce these concepts here and formally define them in Section 4.

Before we can introduce the concepts we need to introduce the necessary terms we are going to use. Nodes are represented as atoms. The set of justifications is replaced by a program \( P \). The set of atoms in a program \( P \) is denoted by \( A_P \). See also [BDE15].

2.2.1 Status

An important concept is the status \( s \) of an atom, which can be any of the set
\[ \{\text{in}, \text{out}, \text{unknown}\}, \]
where \( s = \text{in} \) means true and \( s = \text{out} \) means false. The status unknown is only used during evaluation.

2.2.2 Rule conditions

The condition of a rule is determined by the statuses of its atoms in the body of the rule. We have three different conditions:

- **Valid** rules are those where all atoms in the positive body \( (B^+) \) have status \( \text{in} \) and all atoms in the negative body \( (B^-) \) have status \( \text{out} \).
- **Invalid** rules have some atoms with status \( \text{out} \) in \( B^+ \) or some atoms with status \( \text{in} \) in \( B^- \).
- **Unfounded** rules have only atoms with status \( \text{in} \) in \( B^+ \), some atoms with status \( \text{unknown} \) and no atoms with status \( \text{in} \) in \( B^- \).
2.2.3 Support

The support(a) of an atom a is a set of atoms that depends on the status of a.

- If a has status in the supporting atoms are composed of bodies of all rules in the program that are valid and atom a is in their head.
- For status out of an atom a the set is composed of atoms from $B^+$ (which have status out) and atoms from $B^-$ (which have status in), where a is in the head of the rule. This applies only if there is no valid rule for which a is in the rule head.
- If a has status unknown there is no support for a, i.e. the support is the empty set.

2.2.4 (Affected) Consequences

The consequences of an atom a are those heads of rules where a is in their body. The affected consequences is given by the set of atoms for which a is in their support.

3 Stream Stratification

Stratification [ABW88] is a means to add a procedural aspect to logic programs (i.e. order of evaluation for predicates) by splitting the program into strata which can be evaluated successively. However, this method is restricted to programs with acyclic negation. By using stratification on inputs from streams and defining stream stratification the temporal validity of atoms can be predicted hierarchically [BDE15]. To create a stratification of a program $P$ we need to define a dependency graph first.

Definition 7 (Stream dependency graph) For the dependency graph of a program $P$ we use a directed graph $G_p = (V,E)$. The set of vertices $V$ contains the extended atoms $\alpha \in \mathcal{A}^+(P)$ in $P$, and atoms and $@$-atoms contained in $a$. Formally we write

$$V = \mathcal{A}^+(P) \cup \{a | \Box^* a \in \mathcal{A}^+(P)\} \cup \{@_t a | \Box @_t a \in \mathcal{A}^+(P)\},$$

and $* \in \{@_t, \Box, \Box^*\}$. The set $E$ of edges contains

- $\alpha \rightarrow^\geq \beta$, 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
- $\Box^* a \rightarrow^> a$, if $\Box^* a \in \mathcal{A}^+(P)$.
Intuitively, an edge $\alpha \rightarrow \succ \beta$ with $\succ \in \{>, \geq, =\}$ means that the truth value of $\alpha$ depends on the truth value of $\beta$. The $\succ$ label indicates when it is possible to separate the program into strata, which can in turn be evaluated one after another.

**Example 6** Let $P = \{v_1 \leftarrow \oplus^5 @_t v_2\}$. Then $G_P = (V, E)$, where

$$V = \{v_1, \oplus^5 @_t v_2, @_t v_2, v_2\}$$

and

$$E = \{v_1 \rightarrow \geq \oplus^5 @_t v_2, \oplus^5 @_t v_2 \rightarrow > v_2, @_t v_2 \leftrightarrow = v_2\}.$$

For a window atom $\oplus\diamond a$ the atom $a$ has to be evaluated before the window atom is evaluated. In the dependency graph this is an edge of the form $\oplus\diamond a \rightarrow > a$. Now we can define stream stratification for logic programs.

**Definition 8 (Stream stratification)** Let $P$ be a program and $G_P = (V, E)$ its stream dependency graph. A mapping

$$\lambda : A^+(P) \rightarrow \{0, \ldots, n\}, \ n \geq 0,$$

is said to be a stream stratification if $\alpha \rightarrow \succ \beta \in E$ implies $\lambda(\alpha) \succ \lambda(\beta)$ for all $\succ \in \{>, \geq, =\}$. If such a stratification exists, $P$ is called stream stratified.

By definition, stratification also means that the dependency graph has no cycles with an edge $\alpha \rightarrow > \beta$, i.e. two window atoms can not depend on each other. Checking that no strongly connected sub-graph of $G_P$ contains an edge $\alpha \rightarrow > \beta$, can be done in linear time using standard methods.

The evaluation of a stream stratified program $P$ is the same as for any other stratified logic program. The strata are evaluated one after another, where the results of previous strata are the input for the current stratum. We say the result is pushed to the next stratum.

### 4 Extending Truth-Maintenance for LARS

We are now going to review an extension of truth-maintenance techniques for LARS as presented in [BDE15]. Similar to a TMS, which updates a stable model in light of new rules, the answer update algorithm incrementally updates a LARS answer stream.

To use a truth-maintenance system for LARS some of the concepts presented in Section 2.2 were adapted in [BDE15]. In addition, stream stratification will be used to handle the stream input. For a TMS data structure $\mathcal{L}$ a label $L(\alpha) = (s, \mathcal{T})$ is introduced, where $s \in \{in, out, unknown\}$ is the status of an extended atom $\alpha \in A^+$ and $\mathcal{T}$ is a set of time intervals during
Rule Conditions on statuses of atoms in $B(r)$

<table>
<thead>
<tr>
<th>Condition</th>
<th>Valid Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>valid</td>
<td>$B^+$ all in $\land$ $B^-$ all out</td>
</tr>
<tr>
<td>invalid</td>
<td>$B^+$ some out $\land$ $B^-$ some in</td>
</tr>
<tr>
<td>unfounded</td>
<td>$B^+$ all in $\land$ $B^-$ none in $\land$ $B^-$ some unknown</td>
</tr>
</tbody>
</table>

Figure 3: Rule conditions

which $\alpha$ has status $s$. If $L(\alpha) = (s, T)$, we also write $st(\alpha)$ to refer to $s$. This label extends the status from Section 2.2 by mapping it to time intervals. The rule conditions also apply here, and are depicted in Figure 3. For better readability abbreviations were used, e.g. “$B^+$ all in” is an abbreviation for $\forall a \in B^+ : st(a) = \text{in}$. The support and the (affected) consequences are also extended to incorporate the window atoms and $\hat{\alpha}$-atoms. We write $\hat{\alpha} \in P$ if $\alpha$ occurs in $P$. Similarly, we write $a \in \hat{\omega}$ if atom $a$ occurs in a window atom $\omega$.

**Definition 9 (Support)** The support is the union of the positive, negative and $\hat{\alpha}$-support of $\alpha$. The positive and negative support of $\alpha$ support the status of an atom by rules and the $\hat{\alpha}$-support captures the support for labels of atoms $\alpha$ of the form $\hat{\alpha}$.  

$$
\text{Supp}^+(\alpha) = 
\begin{cases} 
\emptyset & \text{if } st(\alpha) \neq \text{in} \\
\bigcup_{r \in P^H(\alpha) \land \text{valid}(r)} B(r) & \text{otherwise}
\end{cases}
$$

$$
\text{Supp}^-(\alpha) = 
\begin{cases} 
\emptyset & \text{if } st(\alpha) \neq \text{out} \\
\bigcup_{r \in P^H(\alpha)} f(r) & \text{where } f(r) \text{ selects some random} \\
\beta \in B^+(r), \text{where } st(\beta) = \text{out}, \text{ or} \\
\beta \in B^-(r), \text{where } st(\beta) = \text{in}
\end{cases}
$$

$$
\text{Supp}^\alpha(\alpha) = 
\begin{cases} 
\emptyset & \text{if } \alpha \notin A \text{ or } st(\alpha) = \text{out} \\
\bigcup_{\hat{\alpha} \in P \land \hat{\alpha} \neq \alpha} \hat{\alpha} & \text{if } st(\hat{\alpha}) = \text{in}
\end{cases}
$$

$$
\text{Supp}(\alpha) = \text{Supp}^+(\alpha) \cup \text{Supp}^-(\alpha) \cup \text{Supp}^\alpha(\alpha)
$$

The set of consequences for an atom $\alpha$ is the union of the consequences of the head of a rule $r$ in a program $P$, the window atoms for $\alpha$ and the atoms at time $t$ for $\alpha$. The consequences capture the structural dependencies between atoms $a \in A$ and extended atoms $\alpha \in \mathcal{A}^+$.

**Definition 10 (Consequences)** Let $\alpha \in \mathcal{A}^+$ be an extended atom and $P$ a program, then the set of consequences for $\alpha$ is defined as:
Cons_h(\alpha) = \{ H(r) \mid \exists r \in P, \alpha \in B(r) \}

Cons_w(\alpha) = \begin{cases} \{ \oplus \star \alpha \in P \} & \text{if } \alpha = a, \text{ where } \star \in \{ \odot, \square \} \text{ and } a \in A \\ \{ \oplus \odot \beta \} & \text{if } \alpha = \odot \beta \end{cases}

Cons_\odot(\alpha) = \begin{cases} \{ a \} & \text{if } \alpha = \odot t a \text{ and } \odot t a \in P \\ \emptyset & \text{otherwise} \end{cases}

Cons(\alpha) = Cons_h(\alpha) \cup Cons_w(\alpha) \cup Cons_\odot(\alpha).

The set Cons^*(\alpha) is the transitive closure of Cons(\alpha), i.e., the smallest set C where Cons(\alpha) \subseteq C and \bigcup_{\beta \in C} Cons(\beta) \subseteq C.

**Definition 11 (Affected consequences)** All consequences with \alpha in their support, plus all atoms a \in A, where \alpha is an \@-atom, are the affected consequences of \alpha.

ACons(\alpha) = \{ \beta \in Cons(\alpha) \mid \alpha \in Supp(\beta) \} \cup Cons_\odot(\alpha)

For a set A of atoms, the affected consequences at time t are defined as:

ACons(A, t) = \begin{cases} \bigcup_{a \in A} Cons^*(a) & \text{if } t = 0 \\ \bigcup_{a \in A} ACons^*(\odot t a) \setminus A & \text{if } t > 0 \end{cases}

By ACons^*(\alpha) we denote the transitive closure of the affected consequences.

Its definition is similar to Cons^*(\alpha).

The affected consequences can also be derived per stratum. In that case only those atoms are allowed which belong to the current stratum.

**Example 7** The grounded rule \(r_1\), from the program in Figure 1, at time \(t = 37.2m\) is:

\((r_1')\) \text{ @40.2m expBusM} \leftarrow \oplus \odot 3m \oplus \odot 37.2m bus, \text{ on.}\)

Consequences for this rule are as follows:

Cons_h(on) = \{ @40.2m expBusM \}

Cons_h(\oplus \odot 3m \oplus \odot 37.2m bus) = \{ @40.2m expBusM \}

Cons_w(\odot 37.2m bus) = \{ \oplus \odot 3m \oplus \odot 37.2m bus \}

Cons_\odot(\odot @40.2m expBusM) = \{ expBusM \}

With the following labels \((r_1')\) is valid:

\[ L(\oplus \odot 3m \oplus \odot 37.2m bus) = (in, [37.2m, 40.2m]) \]

\[ L(on) = (in, [38.0m, 39.0m]) \]

The support and the affected consequences are

\[ Supp^+(\odot @40.2m expBusM) = \{ \oplus \odot 3m \oplus @37.2m bus, \text{ on} \}, \]

\[ ACons(\oplus \odot 3m \oplus @37.2m bus) = \{ @40.2m expBusM \}. \]
5 Answer Update Algorithm

In this section we will present the answer update algorithm. First we will explain how the algorithm works at a high level. Afterwards, in Section 5.2, we will go into detail on what the sub-procedures do.

5.1 Main Procedure

The general idea of the algorithm is to update the status of atoms from a given program $P$ within a time interval so that the answer, which is derived from the program, is true for the current time and the available information. The atoms which make it necessary to make an update are contained in the data stream $D$, which is an input parameter for the answer update algorithm. To handle the update in an efficient way several concepts are used. The high level ideas of those concepts we presented in the previous sections and we will now show how they were implemented.

The input program $P$ for the algorithm is split into strata using the method for stream stratification introduced in Section 3. The strata are also numbered, where the stratum in which there are no atoms which depend on other atoms, gets the number 1. We will also use the terms “first stratum” and “lowest stratum” to refer to the stratum with number 1. The lowest stratum is a special case in the algorithm. It is the only one where input atoms are used. This requires to calculate some sets differently than on any other stratum. We will go into detail about where these differences are and how the process is different in the next section.

The update algorithm consists of several parts. The first part is the collection and the status update of expired and fired atoms. Since we are using an extended TMS this means that not only the status is updated, but also the set of time intervals associated with it, i.e., we update the label of atoms. The sub-procedure Expired in Line 6 of Algorithm 1 collects all atoms for which their status is not out, but their time label does not contain any time point between a specified time point and the current time point. After that, ExpireInput in Line 7 updates the labels of all those atoms. Similarly Fired in Line 9 collects all atoms whose status is not in, and FireInput in Line 10 sets the labels accordingly. See Sections 5.2.1 - 5.2.4 for more details.

In the next part, starting at Line 15 with UpdateTimestamps, only the timestamps of those atoms are updated for which the status does not change. This is done based on sets of window atoms with changed labels and old labels respectively, which were stored before. In Algorithm 1 these are the variables $C$ and $L'$. This is explained in detail in 5.2.5 UpdateTimestamps.

In the next part, starting at Line 15 with UpdateTimestamps, only the timestamps of those atoms are updated for which the status does not change. This is done based on sets of window atoms with changed labels and old labels respectively, which were stored before. In Algorithm 1 these are the variables $C$ and $L'$. This is explained in detail in 5.2.5 UpdateTimestamps.

Now the algorithm looks at all the atoms which do not have a status yet, or rather which have a status unknown. Several subroutines are involved in setting the statuses. At first the status of every atom is set to unknown in Line 16 if its time label does not contain the specified time point $t$. 

Algorithm 1: AnswerUpdate($t, D, L$)

Input: Time point $t$, data stream $D$, TMS structure $L$ of a stream-stratified program $P = P_0 \cup \ldots \cup P_n$ for $D$ at time $t' < t$

Output: Update of $L$ for $P$ at time $t$ or fail

1 foreach stratum $\ell := 1 \rightarrow n$ do
2 \quad \begin{align*}
3 & C := \emptyset \\
4 & L' := \text{current labels}
5 \end{align*}
6 /* Update labels of window atoms */
7 foreach $\omega \in \text{Expired}(\ell, t', t)$ do
8 \quad \begin{align*}
9 & \text{ExpireInput}(\omega, t) \\
10 & C := C \cup \{\omega\}
11 \end{align*}
12 foreach $\langle \omega, t_1 \rangle \in \text{Fired}(\ell, t', t)$ do
13 \quad \begin{align*}
14 & \text{FireInput}(\omega, t_1) \\
15 & C := C \cup \{\omega\}
16 \end{align*}
17 /* Update rule heads */
18 //Where possible, extend rule head intervals
19 updateTimestamps($C, L', \ell, t$)
20 setUnknown($\ell, t$) repeat
21 \quad //determine label for some unknown head
22 \quad if $\text{setRule}(\ell, t) = \text{fail}$ then return fail
23 \quad //fix supporting assignment for some unfounded valid rule
24 \quad if $\text{makeAssignment}(\ell, t) = \text{fail}$ then return fail
25 until no new assignment made
26 setOpenOrdAtomsOut($\ell, t$)
27 PushUp($\ell, t$) //heads serve as input for the next stratum

Afterwards for all atoms $\alpha$, which now have status unknown, all rules where $\alpha$ is in the rule head are checked for (in)validity. This is done by SetRule in Line 18. Then the status and the time label of $\alpha$ are set appropriately by setHead (see 5.2.7 SetRule/SetHead). There is also the case that a rule was unfounded (see Sections 2.2 and 4). The subroutine MakeAssignment in Line 20 will pick one such unfounded rule, set the status of the atom $\alpha$ to in and the status of all atoms in the negative body to out. There are also subroutines for the special cases of atoms and @-atoms where the label has to be set in an extra step. See 5.2.7 SetHead and 5.2.9 SetOpenOrdAtomsOut for details. Finally in Line 23 the atoms collected in Fired are added as input and pushed to the next stratum.
5.2 Sub-procedures

We now go through the details of the sub-procedures.

5.2.1 Expired

As the name suggests, this method collects all atoms which are expired at time \( t_1 \) between time point \( t' \) and \( t \), i.e., \( t' < t_1 < t \) (that is all atoms \( a \in \mathcal{A} \) from a window atom \( \omega = \square^k \star a \in \mathcal{A}^\tau \), where \( \star \in \{\square, \Diamond\} \), which had status \( \text{in} \) in \( k \) time points before a time point \( t' < t_1 \leq t \).) All \( \text{Fired} \) (see below) atoms are exempt from this set, and removed from the set of expired atoms, if necessary.

Expired, shown in (12), is the union of the sets of expired atoms at a time \( t_1 \). These subsets (13), are unions of the sets of expired atoms for each window atom (14). Time \( t^* \) depends on the quantifier (i.e. \( \{\square, \Diamond\} \)) of the window atom and whether the window atom incorporates an \( @ \)-atom. The basis for the subsets is a list of pairs of \( a \in \omega \) and the time point \( t' \) at which the atom had status \( \text{in} \) in (15).

\[
\text{Expired}(\ell, t', t) := \bigcup_{t' < t_1 \leq t} \text{Expired}(\ell, t_1)
\]

(12)
\[
\text{Expired}(\ell, t) := \bigcup_{\omega \in P} \{(a, \omega) \mid a \in \text{exp}(\omega, t) \} \setminus \text{Fired}(\ell, t)
\]

(13)
\[
\text{exp}(\omega, t^*) := \{a \mid (a, t') \in q(\omega) \land t' < t^*\}
\]

(14)
\[
q(\omega) := \{(a, t') \mid a \in \mathcal{A} \land t' < t \land \text{st}(a) = \text{in}\}
\]

(15)

Note that only the set of expired atoms for stratum \( \ell \) is being collected.

5.2.2 ExpireInput

After the collection process, the set of tuples of form \( \langle a, \omega \rangle \) returned by \( \text{Expired} \) is traversed. For the label \( L(\omega) = (s, T) \) of each window atom \( \omega \), we check whether \( t_2 < t \) for some \( [t_1, t_2] \in T \). If \( t \) is past the interval from the label, the status of \( a \) is set to \( \text{out} \) and its time label is set to \( S_{\text{out}} = [t_1^*, t_2^*] \), where \( t_i^* \) is again depending on the quantifier in \( \omega \). In addition, all window atoms whose labels have changed are stored in a set \( C \). This will be needed, along with the original labels \( L' \) in \( \text{UpdateTimestamps} \).
5.2.3 Fired

This method collects all atoms which are firing between a time point \( t' \) and time point \( t \), with \( t' < t \).

On the first stratum the fired atoms are collected from the input stream. In higher strata they are collected from the previous strata. In addition to the collected atoms, the time point \( t_1 \) (i.e., the time point at which the firing happens) is also saved. Similar to \( \text{Expired} \) the set of fired atoms is also the union of fired atoms at a time point \( t' \leq t_1 \leq t \) \(^{(16)}\). In contrast to \( \text{Expired} \) there is also a distinction in the collection process between the first stratum \(^{(17)}\) and higher strata \(^{(18)}\). In the first stratum \( \text{Fired}(1, t) \) is collected from atoms in the input stream (i.e. the value of \( v(t) \)), rather than from \( \text{Push}(\ell) \) and \( \text{PushNow}(\ell) \).

\[
\text{Fired}(\ell, t', t) = \bigcup_{t' < t_1 \leq t} \text{Fired}(\ell, t_1) \tag{16}
\]

\[
\text{Fired}(1, t) = \{ \langle a, \omega \rangle | a \in v(t) \land \omega \in \text{Cons}_\omega(a) \} \cup \\
\{ \langle \oplus_t a, \oplus_t a \rangle | a \in v(t) \land \oplus_t a \in P_1 \} \tag{17}
\]

\[
\text{Fired}(\ell, t) = \{ \langle \alpha, \omega, t \rangle | \langle \alpha, \omega \rangle \in \text{PushNow}(\ell) \} \cup \\
\{ \langle \alpha, \omega, t_1 \rangle | \langle \alpha, \omega, t_1 \rangle \in \text{Push}(\ell) \land t_1 \leq t \} \tag{18}
\]

The set of fired atoms at higher strata is composed of atoms which can be pushed immediately (i.e., at the current time \( t \)) and those pushed at a time determined by the functions \( \text{wf}(\alpha, \omega, \ell) \) \(^{(21)}\) and \( aR(\alpha, \omega, t_1, t_2) \) \(^{(22)}\).

\[
\text{PushNow}(\ell) = \{ \langle a, \omega \rangle | \text{wf}(a, \omega, \ell) \land a \in \bigcup_{i=1}^{l-1} \text{Updated}(i) \} \tag{19}
\]

\[
\text{Push}(\ell) = \left\{ \langle \oplus_t a, \omega, t \rangle | \text{wf}(a, \omega, \ell) \lor \text{wf}(\oplus_t a, \omega, \ell) \rangle \
\land \langle \oplus_t a \in \bigcup_{i=1}^{l-1} \text{Updated}(i) \\land \text{L}(\oplus_t a) = [t_1, t_2] \land t \in aR(\alpha, \omega, t_1, t_2) \rangle \right\} \tag{20}
\]

The following functions determine when and where to push the atoms.

\[
\text{wf}(a, \omega, \ell) = \begin{cases} 
\text{true} & \text{if } \omega \in \text{Cons}_\omega(\alpha) \land (\lambda(\omega) = \ell) \\
\text{false} & \text{otherwise}
\end{cases} \tag{21}
\]

\[
aR(\alpha, \omega, t_1, t_2) = \begin{cases} 
[t' - n, t'] & \text{if } n > 0, \text{ where } \text{tm}(\oplus_t a) = [m, n] \\
[t_1, t_2] & \text{otherwise}
\end{cases} \tag{22}
\]
5.2.4 FireInput

The resulting set of Fired is traversed and the labels of the window atoms at $\ell$ are updated. The time intervals to which the labels are assigned, depend on the window operators and functions, and whether an @-atom is involved. All window atoms with changed labels are stored in $C$ (same as for ExpireInput).

5.2.5 UpdateTimestamps/UpdateTimestamp

Using the set $C$ and a data structure $L'$ that keeps the old labels, sets can be computed which allow to determine how timestamps are updated. The sets are:

- $KI$ (keep in): contains the window atoms that keep their status in and for which the time labels are extended.

- $KO$ (keep out): contains the window atoms that keep their status out and for which the time labels are extended.

- $I2O$ (in to out): contains the window atoms that have their status changed from in to out.

- $O2I$ (out to in): contains the window atoms that have their status changed from out to in.

We also define conditions on the sets $KI$, $KO$, $I2O$ and $O2I$, which are for a given rule $r$:

\[
B^+(r) \cap I2O = \emptyset \land B^-(r) \cap O2I = \emptyset \quad (U1)
\]

\[
B^+(r) \cap KI \neq \emptyset \lor B^-(r) \cap KO \neq \emptyset \quad (U2)
\]

\[
B^+(r) \cap KO \neq \emptyset \land B^-(r) \cap KI \neq \emptyset \quad (U3)
\]

Condition (U1) is true, if the support of $r$ did not change. Condition (U2) means that a positive (negative) atom from the body of $r$ keeps its status in (out), and its time label is extended. The time label of $H(r)$ is extended if its status is in. Finally Condition (U3) is similar, with the statuses switched.

UpdateTimestamp($r$, status) is called with status = in if (U1) and (U2) hold, and status = out if (U1) and (U3) hold. This is repeated until none of the three rules (U1) - (U3) is satisfied by a rule $r$. UpdateTimestamp sets the end point in the time label for $H(r)$ to $MinEnd(r, t)$ and, depending on its status, adds $H(r)$ to either $KI$ or $KO$.

\[
MinEnd(r, t) = \begin{cases} 
\min\{t_2 \mid b \in B(r) \land tm(b) = [t_1, t_2]\} & \text{if } \forall b \in B(r) : t \in tm(b) \\
\mathit{t} & \text{otherwise}
\end{cases} \quad (23)
\]
5.2.6 SetUnknown

This function sets all atoms in $ACons(A, \ell, t)$ to unknown if $t$ is not within their time label. $A$ is the set of all fired and expired atoms in stratum $\ell$ at time $t$.

5.2.7 SetRule/SetHead

SetRule$(\ell, t)$ calls SetHead$(\alpha, \ell, t)$ for every $\alpha \in ACons(\ell, t)$ which has status unknown. SetHead (see Algorithm 2) in return does several things. First, it determines if there are (in)valid rules for which $\alpha$ is in their head. We denote the set of rules with $\alpha$ in their head at a stratum $\ell$ with $P_H^\ell(\alpha)$. If there is at least one rule which is founded valid the endpoint of the time label $L(\alpha)$ is set to the biggest value of $\text{MinEnd}(r, t)$ (see (23)) of all rules on stratum $\ell$ with $\alpha$ in their head:

$$t^* := \max \{ \text{MinEnd}(r, t) \mid r \in P_H^\ell(\alpha) \},$$

and

$$L(\alpha) := (in, [t, t^*]).$$

On the other hand, if all rules are founded invalid, then $t^*$ is the smallest value:

$$t^* := \min \{ \text{MinEnd}(r, t) \mid r \in P_H^\ell(\alpha) \},$$

and

$$L(\alpha) := (out, [t, t^*]).$$

Afterwards the function UpdateOrdAtom$(\alpha, \text{status})$ is called, with status either in or out, and the positive (respectively negative) support is updated. Finally it is checked if there are atoms in the consequences of $\alpha$ which have status unknown. For every such atom $\beta$, SetHead$(\beta, \ell, t)$ is called recursively.

UpdateOrdAtom$(\alpha, \text{status})$ updates only atoms of the form $\alpha = @t_1 a$ ($@$-atoms). If status and $st(a)$ are both in, UpdateOrdAtom inserts the interval $[t_1, t_1]$ into $tm(a)$. If $st(a) \neq in$, the label of $a$ is set to $L(a) = (in, [t_1, t_1])$. If status = out and $st(a) = in$ the interval $[t_1, t_1]$ is removed from $tm(a)$. If $st(a) = out$ as well, the label is set to unknown: $L(a) = (unknown, \emptyset)$.

5.2.8 MakeAssignment

For every affected consequence of $\alpha$ in stratum $\ell$ with status unknown the sub-procedure MakeAssignment assigns a truth value. After the previous functions every such atom is only supported by unfounded rules. If such rules exist MakeAssignment sets the status of $\alpha$ to in (i.e. $st(\alpha) = in$) and all atoms in the negative body of the rule are set to out if their status is unknown.
Algorithm 2: $\text{setHead}(\alpha, \ell, t)$

1. if $\exists r \in P^H_\ell(\alpha): \text{valid}(r)$ then
2. \[ t^* := \max\{\text{MinEnd}(r, t) \mid r \in P^H_\ell(\alpha)\} \]
3. \[ L(\alpha) := (in, [t, t^*]) \]
4. UpdateOrdAtom($\alpha$, $in$)
5. if $\alpha \in \mathcal{A}$ then
6. \[ \text{foreach } \beta_t \alpha \in \text{Supp}^\oplus(\alpha) \text{ do} \]
7. \[ \text{add } [t_1, t_1] \text{ to } tm(\alpha) \]
8. \[ Updated(\ell) := Updated(\ell) \cup \{\alpha\} \]
9. update $\text{Supp}^+(\alpha)$ as defined
10. \[ \text{foreach } \beta \in \text{Cons}(\alpha, \ell): st(\beta) = \text{unknown} \text{ do} \]
11. \[ \text{setHead}(\beta, \ell, t) \]
12. else if $\forall r \in P^H_\ell(\alpha): \text{invalid}(r)$ then
13. \[ t^* := \min\{\text{MinEnd}(r, t) \mid r \in P^H_\ell(\alpha)\} \]
14. \[ L(\alpha) := (out, [t, t^*]) \]
15. UpdateOrdAtom($\alpha$, $out$)
16. update $\text{Supp}^-(\alpha)$ as defined
17. \[ \text{foreach } \beta \in \text{Cons}(\alpha, \ell): st(\beta) = \text{unknown} \text{ do} \]
18. \[ \text{setHead}(\beta, \ell, t) \]

5.2.9 SetOpenOrdAtomsOut

Here any ordinary atom $a$ with status unknown is set to $(out, [t, t])$. When this procedure is called an atom $a$ can have status unknown, if

- it is not concluded by any rule,
- but $\ominus$-atoms of the form $\alpha = \ominus_t a$ update its status,
- and neither $\text{SetRule}$ nor $\text{MakeAssignment}$ set any of its $\ominus$-atoms (i.e. $\ominus_t a$) to status in.

5.2.10 PushUp

Before proceeding to the next stratum consequences are pushed up. For every index $i > \ell$ $\text{PushNow}(i)$ is added to $\text{Fired}(i, t)$ and pairs of $(a, \omega)$ are added to $\text{Fired}(i, t')$ for every triple $(a, \omega, t') \in \text{Push}(i, t)$.

6 Implementation

In this section first we discuss the choice of Scala as implementation language, i.e. the benefits and drawbacks compared to other languages. Afterwards we will discuss the implementation of the algorithm presented in the
previous section. We will compare Algorithm 1 to the implementation and point out the differences between them. We will also discuss why there are differences in some parts of the implementation.

6.1 Programming Methodology

The goal of the implementation was to keep the program as close as possible to the pseudocode presented in [BDE15]. The implementation should of course work as was intended by the algorithm in the first place but its structure should also resemble the pseudocode and the formal definitions from the previous sections as much as possible. The benefit we gain from that is better readability and an easier comparison of the two. Although these requirements were the most important there were other aspects to be considered as well. The program should be written in a modern (i.e. well supported and developed) programming language, which is fairly common (or very close to a common language in terms of its syntax) and it should be cross-platform if possible.

We wanted to maintain the representation of the formal definitions, so we needed to find such a programming language. Many languages which support that, use the functional programming paradigm (e.g. Haskell). The problem which arises from that is that functional programming languages are not well suited to keep state. This was a requirement because the algorithm is based on updating state.

Fortunately in recent years functional programming has found its way into several programming languages which also include mechanisms for imperative or even object-oriented programming. There are many multi-paradigm languages which support functional and imperative (object-oriented) style programming. There was only one programming language which fully satisfied all of the requirements which is called Scala [OCD+06].

Scala first appeared in 2004. It aims to combine object-oriented and functional style programming (as mentioned before), is compiled into Java bytecode and thus fully compatible with Java libraries. The Java Virtual Machine (JVM) [LYBB], and the compatibility with Java, are strong arguments for Scala, because of the vast ecosystem which has evolved around it. For example if we would want to visualize the output, we would benefit from all the frameworks that already exist for Java, and in increasing number also for Scala itself. By compiling into Java bytecode our program becomes highly portable as well, since there is an implementation of the JVM for virtually all common platforms.

Since Java also fulfils all of the requirements, except for the functional paradigm, the question is why not use Java. The choice to use Scala instead of Java had several reasons. For one Scala has a much cleaner, less verbose, syntax than Java. For example, it uses a sophisticated static type inference mechanism. This removes the need to explicitly declare types (similar to
many scripting languages, such as Python) with the benefit that types are checked at compile time and thus bugs concerning wrong types are caught at an early stage (in contrast to e.g. Python, where types are checked dynamically during execution) and are easier to debug.

Last but not least Scala itself is very scalable (hence the name). Scala’s scalability comes from the integration of both object-oriented and functional programming concepts. Further the scalability is manifested in Scala’s “read–eval–print loop” (REPL) which allows for quick one-line expressions and instant results on one hand. On the other hand code generated from Scala comparable to Java code. In addition many issues are caught at compile time rather than at execution [Ode].

In the next section we will point out the benefits of Scala and compare parts of the main method of the algorithm to the pseudocode in Algorithm 1 and formal definitions from previous sections.

6.2 Algorithm in Scala

We were able to achieve our goal of implementing the algorithm as similar to the pseudocode as possible, for almost all methods. For example, for the main Algorithm 1 this was straightforward. (See also the implementation in Listing 1.) The most noticeable differences are the type of the value \( wAOp \) in the parameter list of \( answerUpdate \) in the implementation and that the labels \( L \) are passed to almost every sub-method. The reason for the type of \( wAOp \) is to ensure the generic nature of the algorithm. All window function specific operations are encapsulated in a separate class. All these classes have \( WindowAtomOperators \) as a common trait (Traits in Scala are similar to interfaces in Java). Furthermore the value \( wAOp \) does not have the type of the trait but is a \( HashMap \) which holds any class that implements the trait \( WindowAtomOperators \). This way multiple different window functions can be used in the same call of the \( answerUpdate \) method. In case no special window function is specified, we set the default to use a time-based window function, along with its operators.

Listing 1: Answer Update Algorithm

```scala
    waOperators += wAOp
    val Lp = L.copy
    for (l <- 1 to n) {
        var C = Set[WindowAtom]()
        for (omega <- Expired(D, l, tp, t, L)) {
            ExpireInput(omega, t, L)
        }
    }

    Lp
}
```

21
The methods Fired and SetHead were the most difficult methods to implement. Especially Fired, because it depends on several sub-procedures to collect its atoms. Listing 2 displays the method Fired along with its sub-procedures. Note that there are two different Fired methods. We will now elaborate the differences between the two Fired methods and the sub-procedures Push and PushNow.

- The first Fired method takes a substream in the interval \([tp, t]\) defined by the method parameters, and then iterates over every time point in the interval. For every time point \(t_1 \in [tp, t]\) the second Fired method is called. It implements Equation (16).

- In the second Fired method there are two cases which can apply. The first case applies if the current stratum is the first stratum. The second case applies for any other stratum. For every higher stratum the fired atoms are computed by using the methods Push and PushNow. This implements the equations (17) and (18).

- Push and PushNow collect pairs of window atoms and time points when these atoms are pushed. For PushNow this is always the current time (i.e. the value \(t: Int\) passed to the method as an argument). For every @-atom Push checks if there is a corresponding window atom
using \( \text{wf} \), and then computes its active range using \( aR \). Note that \( aR \) is specific to each window function. These methods implement the equations (19) - (22).

\textit{Fired} and \textit{Expired} are the two methods which have the most differences between their formal definitions in \cite{BDE15} and the implementation. This could not be avoided because we needed to check the types of atoms to distinguish between atoms, \( @ \)-atoms and window atoms and had to loop over all atoms and time points.

**Listing 2: Fired**

```python
    var result = Set[WindowAtom, Int]()
    if (tp >= t) return Set()

    val tlp = Timeline(tp + 1, t)
    val Dp = S(tlp, D.v | tlp)

    for (t1 <- tp to t) {
        result += Fired(l, t1, Dp, L)
    }
    return result
}

    l match {
        case 1 =>
            var result = Set[WindowAtom, Int]()
            for (a <- D(t1)) {
                for (r <- stratum(l).rules) {
                    val tmp = (r.B ++ Set(r.h)) filter (p => ConsW(stratum(l), a).contains(p) || p.contains(AtAtom(t1, a)))
                    tmp.collect { case wa: WindowAtom => wa } foreach { wa => result += ((wa, t1)) }
                }
                return result
        case _ => Push(l, t1, L) ++ PushNow(l, t1)
    }
}

    var result = Set[WindowAtom, Int]()
    for (i <- 1 to l-1) {
        if (updated.contains(i)) {
            updated(i) collect { case a: AtAtom => a } foreach {
                ata =>
                    for (iv <- tm(ata, L)) {
```
\{(\text{wf}(\text{ata}, 1) \leftrightarrow \text{wf}(\text{ata}.\text{atom}, 1)) \text{ foreach } \{ a \mapsto \text{val} \ wfn = \text{waOperators(}a.\text{wop}.\text{wfn}.\text{getClass)} \}
\}
\text{if } (\text{wfn}.\text{aR}(a, \text{iv}, \text{ata}.\text{t}).\text{contains}(t)) \{ \text{result } += ((a, \text{ata}.\text{t})) \}
\}

\}
\text{push foreach } \{ r \mapsto \text{result } += ((r, t)) \}
\text{result}
\}
\text{def} \text{PushNow}(l : \text{Int}, t : \text{Int}) : \text{Set}[\text{WindowAtom}, \text{Int}] = \{}
\text{var} \text{result} : \text{Set}[\text{WindowAtom}, \text{Int}] = \text{Set}(\)
\text{for } (i < -1 \text{ to } l-1) \{ 
\text{if } (\text{updated}.\text{contains}(i)) \{
\text{updated}(i) \text{ collect } \{ \text{case a:Atom } \mapsto a \} \text{ foreach } \{ 
\text{val} \ wa = \text{wf}(ea, l).\text{filterNot}(a \mapsto a.\text{nested}.\text{exists} \{ 
\text{case at:AAtom } \mapsto \text{true} \}) 
\text{wa \ foreach } \{ a \mapsto \text{result } += ((a, t)) \} 
\}
\}
\text{result } += \text{pushNow}
\}
\text{def} \text{wf}(atom : \text{ExtendedAtom}, l : \text{Int}) : \text{Set}[\text{WindowAtom}] = \{}
\text{ConsW}(\text{stratum}(l), atom) \text{ collect } \{ \text{case wa:WindowAtom } \mapsto wa \}
\}

The case is different for \text{SetHead} (see Listing 3). This method resembles the proposed algorithm (Algorithm 2). There are some slight changes nonetheless. For instance we need to add an extra argument \text{prev} to the method. In the next section we will discuss this further.

\begin{Verbatim}
\text{Listing 3: SetHead}
\end{Verbatim}
if (ph.exists(r => fVal(L, r))) {
  if (timeSet.nonEmpty) {
    val tStar = timeSet.max
    L.update(alpha, Label(in,(t, tStar)))
    UpdateOrdAtom(alpha, in, L)
    alpha match {
      case wa: WindowAtom => /* do nothing */
      case _ =>
        val suppAt = support.suppAt(alpha)
        suppAt.foreach {
          case ata: AtAtom =>
            L.addInterval(ata, new ClosedInterval(ata.t, ata.t))
        }
        addToUpdated(alpha, l)
        support.updateP(stratum(l), L)
    }
  }
}
else if (ph.nonEmpty && ph.forAll(r => fInval(L, r))) {
  if (timeSet.nonEmpty) {
    val tStar = timeSet.min
    L.update(alpha, Label(out,(t, tStar)))
    UpdateOrdAtom(alpha, out, L)
    support.updateN(stratum(l), L)
  }
}
Cons(stratum(l), alpha) foreach { beta =>
  if (prev != beta && beta != alpha && L.status(beta) == unknown) {
    if (SetHead(alpha, beta, l, t, L) == fail) {
      return fail
    }
  }
}
success

6.3 Discussion

While writing the prototype we changed some details of the proposed answer update algorithm. Most of these changes are reflected in the implementation, although we also modified parts of the pseudocode from [BDE15].

The changes include the sets returned by the sub-procedures Expired and Fired in Algorithm 1. In Line 6 we replaced the pair \(\langle a, \omega \rangle\) with a single window atom and in Line 9 we replaced the former triple \(\langle a, \omega, t_1 \rangle\) with the pair \(\langle \omega, t_1 \rangle\). Atom \(a\) was redundant, because it is contained within \(\omega\). Further we changed the name of the TMS data structure from \(M\) to \(L\) in order to be consistent with the implementation. In Algorithm 2 we changed the names of \(fVal\) to \(valid\) and \(fInval\) to \(invalid\) for better readability. In Line 5 of Algorithm 2 we also replaced the expression “\(a\) is ordinary” with
“\(a \in A\)” because we do not use the term ordinary to describe atoms.

For the algorithm to return the correct results we modified some procedures to inspect atoms within window atoms. These procedures include FireInput and ACons. FireInput not only checks for \(@\)-atoms directly within the program, but also within window atoms. The procedure Acons\((\alpha)\), which collects all atoms where \(\alpha\) is in the support of its consequences, and it also checks if \(\alpha\) is contained within an extended atom of its support.

We also changed \(L\), which acts as the TMS data structure in our implementation, from a global variable to a local one which is passed to methods as an argument. The consequence of this change is that we had to turn the immutable map, held internally by \(L\), into a mutable map. In Scala the difference is that the contents of a mutable data structure, in this case a map, can be changed. On the other hand, immutable data structures create a new instance every time some value is changed or the data structure is assigned to a new variable. Because we need to change the content of \(L\), an immutable map would be impractical since \(L\) would reference a different object every time we passed it to a method. So all the changes in the method would be lost once it returned. If we kept \(L\) as an immutable map, \(L\) would need to return from every method. The drawback would be that we either could not return any other values, or need to pair \(L\) with every return value. As a side effect this change enhanced the readability of the program in the sense that now every use of \(L\) can be seen at first glance. This was especially helpful in the process of debugging the code.

We also changed the argument list of SetHead. In Listing 3 the argument \(f\) appears twice in the call of method SetHead\((f, f, l, t, L)\). This was necessary to prevent SetHead from recursing infinitely. Note that the first two arguments of SetHead are the same only for the first call. For every subsequent call it is checked if the first two arguments are equal. If they are, SetHead is not called again.

In an effort to comply with Scala’s idiom, we use pattern matching, and built-in functions for collections. For instance such functions include foreach, filter and exists. Pattern matching allows us to replace if-else constructs and directly match fields in case classes. All of these mechanisms make the code more concise and increase its readability.

7 Tests & Evaluation

We will now show the results of running our implementation of the answer update algorithm. We will present the input and output for every subprocedure of the answer update algorithm. We will also point out where the implementation differs from the proposed algorithm in [BDE15] and why we changed it. Throughout, we refer to the program \(P\) from Figure 1.
Example. Let $D = (T, v)$ be the data stream, where $T = [0m, 50m]$ and $v = \{37.2m \mapsto \{\text{bus, request}\}, 39.1m \mapsto \{\text{tram}\}, 40.2m \mapsto \{\text{expBus}\}, 44.1m \mapsto \{\text{expTr}\}\}$. Consider time point $t = 37.2m$.

Prior to calls to \texttt{answerUpdate}, the strata of $P$ are determined. Stratum $P_1$ consists of rules $(r_1), (r_2)$ and $(r_3)$:

$(r_1)$ \quad $\ominus_{37.2m+3m}\text{expBusM} \leftarrow \ominus^{3m}\ominus_{37.2m}\text{bus, on}$.

$(r_2)$ \quad $\ominus_{39.1m+5m}\text{expTramM} \leftarrow \ominus^{3m}\ominus_{39.1m}\text{tram, on}$.

$(r_3)$ \quad $\ominus_{37.2m}\text{on} \leftarrow \ominus^{1m}\ominus_{37.2m}\text{request}$.

$P_2$ consists of rules $(r_4)$ and $(r_5)$:

$(r_4)$ \quad $\text{takeBusM} \leftarrow \ominus^{5m}\ominus\text{expBusM, not takeTramM, not } \ominus^{3m}\ominus\text{jam}$.

$(r_5)$ \quad $\text{takeTramM} \leftarrow \ominus^{5m}\ominus\text{expTramM, not takeBusM}$.

The \texttt{bus} arriving at $t = 37.2m$ and the \texttt{request}, denoted by $\ominus_{37.2m}\text{bus}$ and $\ominus_{37.2m}\text{request}$ are in the first stratum, i.e., in $P_1$.

At time $t = 37.2m$ the labels of all atoms $\alpha \in A$ are $L(\alpha) = (\text{out, } [0, 0])$. Recall that $L$ denotes the data structure holding the labels for all atoms.

\textbf{Expired.} The first sub-procedure called in \texttt{answerUpdate} is \texttt{Expired}. More specifically, we call \texttt{Expired}(\textit{D, l, tp, t, L}), where $D$ is the data stream, $l = 1$ is the stratum, $tp = t = 37.2m$ are time points.

For every window atom $\omega$ returned by \texttt{Expired}, \texttt{ExpireInput}(\textit{omega, t, L}) is called. For our current arguments \texttt{Expired} returns the empty set, i.e.,

$$\text{Expired}(\textit{D, l, tp, t, L}) = \text{Set}()$$

Therefore, \texttt{ExpireInput} is not executed.

\textbf{Fired/FireInput.} Next, \texttt{Fired}(\textit{D, l, tp, t, L}) is called with the same arguments as for \texttt{Expired}. It contains two pairs:

\texttt{Fired}(\textit{D, 1, 37.2m, 37.2m, L}) = Set(\{\ominus^{3m}\ominus_{37.2m}\text{bus, } 37.2m\}, \{\ominus^{1m}\ominus_{37.2m}\text{request, } 37.2m\})

Now \texttt{FireInput}(\textit{omega, t1, l, D, L}) is called, where $\omega = \ominus^{3m}\ominus_{37.2m}\text{bus}$ and $t1 = 37.2m$ in one loop, and $\ominus^{1m}\ominus_{37.2m}\text{request}$ and $t1 = 37.2m$ in the other loop.

\texttt{FireInput} updates the label for $\omega$ and \texttt{@}-atoms at time $t1$. Note that \texttt{FireInput} checks whether there are such \texttt{@}-atoms within window atoms. In the original presentation of the algorithm only those \texttt{@}-atoms where updated that are directly contained in $P$, i.e., not in the scope of a window. However, all of them need to be accounted for to compute the transitive consequences later. Without this change \texttt{@}-atoms would sometimes not have the correct labels.

The updated label for $\omega$ is $(\text{in, } [22320, 22320])$ for both $\ominus^{3m}\ominus_{37.2m}\text{bus}$ and $\ominus^{1m}\ominus_{37.2m}\text{request}$. Note that $22320 = 37.2m$.  

27
UpdateTimestamps. We call UpdateTimestamps\((C, L, L_p, l, t)\), where argument \(C\) contains all updated window atoms. Here, we have \(C = \{⊞^3m\text{\texttt{bus}}, ⊞^1m\text{\texttt{request}}\}\). The value \(L_p\) is a copy of \(L\) which contains the original labels. At this point, UpdateTimestamps does not do anything.

SetUnknown. The method SetUnknown\((l, t, L, A)\) is called where \(A = \{\text{\texttt{bus}}\}\) is the set of input atoms. It calculates \(ACons(a)\) for every \(a \in A\). SetUnknown returns the following set, and set all statuses to unknown:

\[
\text{SetUnknown}(1, 37.2, L, \{\text{\texttt{bus}}\}) = \{\text{\texttt{expBusM}}, \text{\texttt{@40.2m expBusM on}}\}
\]

SetRule/SetHead. Next, method SetRule\((l, t, L, unknowns)\) is called with \(unknowns = \{\text{\texttt{expBusM}}, \text{\texttt{@40.2m expBusM, on}}\}\) which was returned by SetUnknown. For every \(a \in unknowns\), SetRule calls SetHead\((a, a, l, t, L)\). First SetHead sets the label of \(on\) to \((\text{\texttt{in}}, [37.2m, 38.2m])\). In a further iteration the method determines that rule \((r_1)\) is valid, since \(on\) now has status in. Thus it assigns \(\text{\texttt{@40.2m expBusM}}\) to \(\text{\texttt{in}}\) and, SetHead assigns

\[
L(\text{\texttt{@40.2m expBusM}}) = (\text{\texttt{in}}, [37.2m, 38.2m])
\]

The call of SetHead for \text{\texttt{expBusM}} does nothing because \text{\texttt{expBusM}} is not in the head of any rule.

SetOpenOrdAtomsOut. After SetRule is called, nothing needs to be done in MakeAssignment, since no atom in unknowns remains unknown. Thus, we next call SetOpenOrdAtomsOut\((l, t, L)\) which sets the status of \text{\texttt{expBusM}} to out.

PushUp. The only extended atom being pushed to the next stratum is \(⊞^5m\text{\texttt{expBusM}}\).

Consider now stratum 2. The label for the window atom \(⊞^5m\text{\texttt{expTramM}}\) is still \((\text{\texttt{out}}, [0, 0])\) as initialized. Due to its status, rule \((r_5)\) is invalid. Therefore, the label of \text{\texttt{takeTrM}} is updated to \((\text{\texttt{out}}, [37.2, 37.2])\). However, the status for \(⊞^5m\text{\texttt{expBusM}}\) is set to \(\text{\texttt{in}}\). Although \(⊞^5m\text{\texttt{expTramM}}\) is \text{\texttt{out}} \text{\texttt{takeBusM}} cannot be set to \(\text{\texttt{in}}\) because \(⊞^5m\text{\texttt{jam}}\) from \((r_4)\) is still \(\text{\texttt{out}}\).

If we call the algorithm again later at \(t = 38m\), UpdateTimestamps will extend the time intervals in the labels for \(\text{\texttt{on}}\) and \(\text{\texttt{@40.2m expBusM}}\). If \(\text{\texttt{on}}\) and \(\text{\texttt{@40.2m expBusM}}\) get their time intervals updated depends on the atoms in the body of the rules they are concluded by. The method checks which atoms the status stayed the same. In this example the statuses for \(⊞^3m\text{\texttt{bus}}\) and \(⊞^1m\text{\texttt{request}}\) are not changed from \(\text{\texttt{in}}\). Now for every rule the three conditions mentioned in Section 5.2.5 are checked. The conditions \((U1)\) and \((U2)\) become true, and thus the
method $UpdateTimestamp(r : StdRule, s : Status, t : Int, L : Labels)$ is called with $r = (r_3)$ and $s = in$ for on, and with $r = (r_1)$ and $s = in$ for @40.2m expBusM. The argument $t$ is the same as for $UpdateTimestamps$. The $UpdateTimestamp$ method sets the new time interval for the rule head. In our example the new labels are $L(on) = Label(in, Set([37.2m, 38.2m]))$ and $L(@40.2m expBusM) = Label(in, Set([40.2m, 40.2m]))$.

8 Conclusion

In this work we have presented a detailed description of the answer update algorithm of [BDE15] and a prototypical implementation. We tried to stay as close to the proposed algorithm and the mathematical definitions as possible. In some cases however this was impossible or impractical as we have pointed out. The program is also far from being finished. For one, we only considered time-based window functions whereas at least tuple-based window functions would be interesting to implement as well.

Another topic which is a subject for future work is exploiting the functional programming capabilities of Scala even more. We did use it where it was convenient, e.g. by using immutable collections, but more imperative constructs could be replaced by functional ones to improve readability. Our code still has many properties of imperative, respectively object-oriented programming. There is also little to no error handling in the code, which we omitted to concentrate solely on the algorithm itself.

For further reading the code for all procedures presented in this paper can be found in Appendix A and Appendix B. Appendix A depicts the main procedure along with all the window function independent sub-procedures and Appendix B shows all the methods specific for time-based windows. The whole framework can also be found on Github.\footnote{https://github.com/hbeck/lars, accessed: 2016-11-18}
Appendices

A TMS.scala

```scala
package lars.tms

import lars.core.ClosedIntInterval
import lars.core.semantics.formulas._
import lars.core.semantics.programs.extatoms._
import lars.core.semantics.programs.general.inspect.
  ExtensionalAtoms
import lars.core.semantics.programs.standard.inspect.PH
import lars.core.semantics.programs.standard.{StdProgram, StdRule}
import lars.core.semantics.streams.{S, Timeline}
import lars.core.windowfn.TimeWindowFixedParams
import lars.core.windowfn.time.TimeWindowFixedParams
import lars.strat.Strata
import lars.tms.acons.ACons
import lars.tms.cons.{Cons, ConsStar, ConsW}
import lars.tms.incr.Result
import lars.tms.status.Status.{in, out, unknown}
import lars.tms.status.rule.{fInv, fVal, ufVal}
import lars.tms.status.{Label, Labels, Status}
import lars.tms.support._

//import scala.collection.immutable._
import scala.collection.immutable.HashMap

/**
 * Created by hb on 6/25/15.
 */
case class TMS(P: StdProgram) {

  private val stratum: Map[Int, StdProgram] = Strata(P)
  private val n = stratum.keySet.reduce(math.max)
  private var updated = Map[Int, Set[ExtendedAtom]]()
  private var waOperators: HashMap[Class[_ <: WindowFunctionFixedParams], WindowAtomOperators] =
    HashMap(classOf[TimeWindowFixedParams] =>
      TimeWindowAtomOperators)
  private var pushNow = Set[(WindowAtom, Int)]()
  private var push = Set[WindowAtom]()
  private var A = Set[Atom]()
  private var support = Support()

  def answerUpdate(L: Labels, tp: Int, t: Int, D: S,
```
waOperators +++ wAOp
val Lp = L.copy

for ( l <- 1 to n ) {
    var C = Set[WindowAtom]()
    for ( omega <- Expired(D, l, tp, t, L) ) {
        ExpireInput(omega, t, L)
        C += omega
        A += omega.atom
        addToUpdated(omega, l)
    }
    for ((omega, t1) <- Fired(D, l, tp, t, L)) {
        FireInput(omega, t1, l, D, L)
        C += omega
        A += omega.atom
        addToUpdated(omega, l)
    }
    UpdateTimestamps(C, L, Lp, l, t)
    val unknowns = SetUnknown(l, t, L, A)
    var madeNewAssignment = false
    do {
        if (SetRule(l, t, L, unknowns) == fail) return None
        val opt: Option[Boolean] = MakeAssignment(l, t, L, unknowns)
        if (opt.isEmpty) {
            return None
        }
        madeNewAssignment = opt.get
    } while (madeNewAssignment)
    SetOpenOrdAtomsOut(l, t, L)
    PushUp(l, t, L)
}

Option(L)

def addToUpdated(atom: ExtendedAtom, l: Int) = {
    val heads = stratum(l).rules exists {rule => rule.h.contains(atom)}
    if (heads) {
        updated += l -> (updated.getOrElse(l, Set()) ++ Set(atom))
    }
}

def init(): Labels = {
    val L = Labels()
    initLabels(L)
    initUpdated()
    answerUpdate(L, 0, 0, S(Timeline(0, 0)))
    support = initSupport(L)
    L
}

def initLabels(L: Labels) = {
    val inputAtoms: Set[Atom] = ExtensionalAtoms(P)
A = inputAtoms

A foreach {a \Rightarrow L.update(a, Label(out, (0,0))}

val transCons: Set[ExtendedAtom] = A.flatMap(ConsStar(P, _))

for (x <- transCons){
    x match {
        case wa: WindowAtom \Rightarrow L.update(x, Label(out, (0,0)))
        case _ \Rightarrow L.update(x, Label(unknown))
    }
}

def initUpdated() = {
    for (i <- 1 to n) {
        updated += i -> Set()
    }
}

def initSupport(L: Labels): Support = {
    val support = Support()
    var programAtoms: Set[ExtendedAtom] = Set()
    for (rules <- P.rules) {
        programAtoms += (rules.B ++ Set(rules.h))
    }

    programAtoms foreach {
        a \Rightarrow support.suppP += a -> SuppP(P, L, a)
        support.suppN += a -> SuppN(P, L, a)
        support.suppAt += a -> SuppAt(P, L, a)
    }
    support
}

    var result = Set[WindowAtom]()
    var omegaSet = getOmega(stratum(l))

    if (omegaSet.isEmpty) return result
    for (omega <- omegaSet) {
        for (t1 <- tp to t) {
            if (waOperators(omega.wop.wfn.getClass).exp(omega, L, t1, Fired(l, t1, D, L))) {
                result += omega
            }
        }
    }
    result
}

def getOmega(P: StdProgram): Set[WindowAtom] = {
    var result = Set[WindowAtom]()
    var pH = Set[ExtendedAtom]()
P. rules.foreach(pH += _\cdot h)
(pH ++ P. rules.flatMap(_\cdot B)) collect {case wa:WindowAtom =>
  wa} foreach {result += _}
result
}
  var result = Set[(WindowAtom, Int)]()
  if(tp >= t) return Set()
  val tlp = Timeline(tp + 1, t)
  val Dp = S(tlp, D.v \mid tlp)
  for(t1 <- tp to t) {
    result +== Fired(l, t1, Dp, L)
  }
  result
}
def Fired(l: Int, t1: Int, D: S, L: Labels): Set[(WindowAtom, Int)] = {
  l match {
    case 1 =>
      var result = Set[(WindowAtom, Int)]()
      for(a <- D(t1)) {
        for(r <- stratum(l).rules)
          val tmp = (r.B ++ Set(r.h)) filter(p =>
            ConsW(stratum(l), a).contains(p) \|\|
p.contains(AtAtom(t1, a)))
          tmp.collect { case wa:WindowAtom => wa} foreach {
            wa => result += ((wa, t1))
          }
      }
      result
    case _ =>
      Push(l, t1, L) ++ PushNow(l, t1)
  }
}
  var result = Set[(WindowAtom, Int)]()
  for (i <- 1 to l-1) {
    if (updated.contains(i)) {
      updated(i) collect {case a:AtAtom => a} foreach {
        ata =>
          for (iv <- tm(ata, L)) {
            (wf(ata, l) ++ wf(ata.atom, l)) foreach {
              a =>
                val wfn = waOperators(a.wop.wfn.getClass)
                if (wfn.aR(a, iv, ata.t).contains(t)) {
                  result += ((a, ata.t))
                }
          }
      }
  }
}

33
push foreach \{ r \Rightarrow result += ((r, t)) \}
result

def PushNow(l: Int, t: Int): Set[(WindowAtom, Int)] = {
var result: Set[(WindowAtom, Int)] = Set()
for (i <- 1 to l-1) {
if (updated.contains(i)) {
updated(i) collect \{case a: Atom \Rightarrow a\} foreach {
  ea
    val wa = wf(ea, l).filterNot(a \Rightarrow a.nested.exists {
      case at: AtAtom \Rightarrow true})
  wa foreach \{a \Rightarrow result += ((a, t))\}
}
}
result ++ pushNow
}

ConsW(stratum(l), atom) collect \{case wa: WindowAtom \Rightarrow wa\}
}

def ExpireInput(omega: WindowAtom, t: Int, L: Labels): Unit = {
if (!tm(omega, L).exists(e \Rightarrow e.upper > t)) {
  val wfn = waOperators(omega.wop.wfn.getClass)
  val s_out = wfn.SOut(omega, t)
  L.update(omega, Label(out, (s_out.lower, s_out.upper)))
}
}

val ata = new AtAtom(t, omega.atom)
if (P.rules.exists (r \Rightarrow r.B.exists(_.contains(ata)) || r.h.
contains(ata))) {
  L.update(ata, Label(in, (t, t)))
}
val wfn = waOperators(omega.wop.wfn.getClass)
val s_in: Option[ClosedInterval] = wfn.SIn(omega, t, l, D, L)
if (s_in.isDefined) {
  L.update(omega, Label(in, (s_in.lower, s_in.upper))
}
}
\begin{verbatim}

def tm(a: ExtendedAtom, L: Labels): Set[ClosedIntInterval] = L.intervals(a)

def MinEnd(r: StdRule, t: Int, L: Labels): Int = {
    var t2Set = Set[Int]()
    if (r.B forall (tm(_, L) exists (_, contains(t)))) {
        r.B foreach (tm(_, L) foreach (t2Set += _.upper))
        return t2Set.min
    } t
}

def UpdateTimestamps(C: Set[WindowAtom], L: Labels, Lp: Labels, l: Int, t: Int): Unit = {
    var ki, ko, i2o, o2i = Set[ExtendedAtom]()
    for (wa <- C) {
        val newStatus = L.status(wa)
        if (newStatus == Lp.status(wa)) {
            if (newStatus == in)
                ki += wa
            else
                ko += wa
        } else {
            if (newStatus == in) o2i += wa
            else i2o += wa
        }
    }
    for (rule <- stratum(l).rules) {
        val u1 = (rule.Bp intersect i2o).isEmpty && (rule.Bn intersect o2i).isEmpty
        val u3 = (rule.Bp intersect ko).nonEmpty || (rule.Bn intersect ki).nonEmpty
        if (u1 && u2) {
            val head = UpdateTimestamp(rule, in, t, L)
            if (head.isDefined) ki += head.get
        } else if (u1 && u3) {
            val head = UpdateTimestamp(rule, out, t, L)
            if (head.isDefined) ko += head.get
        }
    }
}

def UpdateTimestamp(r: StdRule, s: Status, t: Int, L: Labels): Option[ExtendedAtom] = {
    if (L.status(r.h) == s) {
        var newIntervals = Set[ClosedIntInterval]()
    }
}
\end{verbatim}
for (interval <- L.intervals(r.h)) {
    if (interval.contains(t)) newIntervals += Set(new
        ClosedIntInterval(interval.lower, MinEnd(r, t, L))
    )
    newIntervals += interval
}
L.update(r.h, Label(s, newIntervals))
return Option(r.h)
}
None
}
    var unknowns : Set[ExtendedAtom] = Set()
    for (a <- ACons(stratum(l), L, A, support, t)) {
        if (!L.intervals(a).exists(_.contains(t))) {
            L.update(a, Label(unknown, L.intervals(a))
            unknowns += a
        }
    }
    unknowns
}
def SetRule(l: Int, t: Int, L: Labels, unknowns: Set[ExtendedAtom]) : Result = {
    for (a <- unknowns) {
        if (SetHead(a, a, l, t, L) == fail) return fail
    }
    success
}
def SetHead(prev: ExtendedAtom, alpha: ExtendedAtom, l: Int, t: Int, L: Labels): Result = {
    val ph = PH(stratum(l), alpha)
    val timeSet = minTime(ph, t, L)
    if (ph.exists(r => fVal(L, r))) {
        if (timeSet.nonEmpty) {
            val tStar = timeSet.max
            L.update(alpha, Label(in, (t, tStar)))
            UpdateOrdAtom(alpha, in, L)
        alpha match {
            case wa: WindowAtom => /* do nothing */
            case _ =>
                val suppAt = support.suppAt(alpha)
                suppAt.foreach {
                    case ata: AtAtom => L.addInterval(ata, new
                        ClosedIntInterval(ata.t, ata.t))
                }
            addToUpdated(alpha, l)
            support.updateP(stratum(l), L)
        }
    }
}
else if (ph.nonEmpty && ph.forAll(r => fInvval(L,r))) {
    if (timeSet.nonEmpty) {
        val tStar = timeSet.min
        L.update(alpha, Label(out, (t, tStar)))
        UpdateOrdAtom(alpha, out, L)
        support.updateN(stratum(l), L)
    }
    Cons(stratum(l), alpha) foreach { beta =>
        if (prev != beta && beta != alpha && L.status(beta) == unknown) {
            if (SetHead(alpha, beta, l, t, L) == fail) {
                return fail
            }
        }
    }
    success
}

def minTime(rules: Set[StdRule], t: Int, L: Labels): Set[Int] =
{
    var minSet = Set[Int]()
    rules.foreach(r => minSet += MinEnd(r, t, L))
    minSet
}

def UpdateOrdAtom(alpha: ExtendedAtom, s: Status, L: Labels): Unit = alpha match {
    case ata: AtAtom =>
        val a = ata.atom
        if (s == in) {
            if (L.status(a) == in) {
                L.update(a, Label(in, tm(a, L) + new ClosedIntInterval(ata.t, ata.t)))
            } else {
                L.update(a, Label(in, (ata.t, ata.t)))
            }
        } else if (s == out) {
            if (L.status(a) == in) {
                L.update(a, Label(out, tm(a, L) - new ClosedIntInterval(ata.t, ata.t)))
            } else {
                L.update(a, Label(unknown, (0, 0)))
            }
        }
    case _ => None
}

def MakeAssignment(l: Int, t: Int, L: Labels, unknowns: Set[ExtendedAtom]): Option[Boolean] = {
    val stillUnknown = unknowns.filter(p => L.status(p) == unknown)
    for (alpha <- stillUnknown) {
        // Further code to perform assignment...
    }
}
val ph = PH(stratum(l), alpha)
if (ph.exists(r => ufVal(L, r))) {
  L.update(alpha, Label(in, (t, t)))
  for (beta <- ph.find(r => ufVal(L, r)).get.Bn) {
    if (L.status(beta) == unknown) L.update(beta, Label(
      out, (t, t)))
  }
  UpdateOrdAtom(alpha, in, L)
  addToUpdated(alpha, l)
  support.updateP(stratum(l), L)
  return Option(false)
}
else {
  L.update(alpha, Label(out, (t, t)))
  ph foreach {r =>
    r.Bp foreach {b =>
      if (L.status(b) == unknown) L.update(b, Label(out, (t, t)))
    }
  }
  UpdateOrdAtom(alpha, out, L)
  support.updateN(stratum(l), L)
  return Option(false)
}
Option(true)

def SetOpenOrdAtomsOut(l: Int, t: Int, L: Labels): Unit = {
  stratum(l).rules foreach {
    rule => (rule.B ++ Set(rule.h)) foreach {
      a => if (L.status(a.atom) == unknown) {
        L.update(a.atom, Label(out, (t, t)))
      }
    }
  }
}

def PushUp(l: Int, t: Int, L: Labels): Unit = {
  pushNow = Set()
  push = Set()
  for (i <- l+1 to n) {
    pushNow += PushNow(i, t)
    val p = Push(i, t, L)
    p.foreach(r => push += r..1)
  }
}

38
B  TimeWindowAtomOperators.scala

package lars.tms
import lars.core.ClosedInterval
import lars.core.semantics.formulas.{Atom, ExtendedAtom}
import lars.core.semantics.programs.extatoms._
import lars.core.semantics.streams.S
import lars.core.windowfn.time.TimeWindowFixedParams
import lars.tms.status.Labels
import lars.tms.status.Status.in

import scala.collection.parallel.mutable

/**
 * Created by et on 09.09.15.
 */
object TimeWindowAtomOperators extends WindowAtomOperators{
    case wfn: TimeWindowFixedParams =>
      val atp = q(omega, L)
      omega match {
        case wb: WBox => mapInAtoms(omega, fired, t, atp)
        case _ =>
          val N = wfn.x.u-wfn.x.l
          mapInAtoms(omega, fired, t+N, atp)
      }
  }

  def mapInAtoms(omega: WindowAtom, fired: Set[(WindowAtom, Int)], t: Int, atp: Set[Int]): Boolean = {
    if (!fired.contains((omega, t))) {
      if (atp.exists(t1 => t1 < t)) return true
    } else false
  }

  override def q(omega: WindowAtom, L: Labels): Set[Int] = {
    var res = Set[Int]()
    if (L.status(omega) == in) {
      for (i <- L.intervals(omega)) {
        res = res ++ i.toSeq.toSet
      }
    }
    res
  }
}
override def aR(wa: WindowAtom, interval: ClosedIntInterval, tp: Int): ClosedIntInterval = wa.wop.wfn match {
  case wfn: TimeWindowFixedParams match {
    wfn.x.u match {
      case 0 => new ClosedIntInterval(interval.lower, interval.upper)
      case n: Int => new ClosedIntInterval(tp - n, tp)
    }
  }
}

  case wfn: TimeWindowFixedParams match {
    var result: Option[ClosedIntInterval] = None
    var t = tStar
    val Nl = wfn.x.l
    val Nu = wfn.x.u
    var at: Option[AtAtom] = None
    wa.nested.find({
      case a: AtAtom => {
        t = a.t
        at = Option(a)
      };
      true
    };
    case _ => false
  })
  wa match {
    case wb: WBox => {
      l match {
        case l => {
          for (t1 <- math.max(0, t - Nl) to t+Nu) {
            if (!D.v(t1).contains(wa.atom)) return None
          }
          result = Option(new ClosedIntInterval(t, t))
        case _ => {
          if (L.intervals(wa).contains(new ClosedIntInterval(math.max(0, t-Nl), t+Nu)))
            result = Option(new ClosedIntInterval(t, t))
        }
      }
      case _ => result = Option(new ClosedIntInterval(t-Nu, t+Nl))
    }
  }
  if (at.isDefined) {
    val tmp = L.intervals(at.get).filter(e => e.lower <= result.get.upper || e.upper >= result.get.lower)
    if (tmp.nonEmpty) {
      val min = math.min(tmp.head.lower, result.get.lower)
      val max = math.max(tmp.head.upper, result.get.upper)
    }
  }
}
result = Option(new ClosedInterval(min, max))
}

result

override def SOut(wa: WindowAtom, t: Int): ClosedInterval =
wa.wop.wfn match {
  case wfn: TimeWindowFixedParams =>
    wa match {
      case wb: WBox =>
        val NI = wfn.x.l
        val Nu = wfn.x.u
        new ClosedInterval(t-Nu, t+NI)
      case _ => new ClosedInterval(t, t)
    }
  }
}

References


