Stream Reasoning with Answer Set Programming: Preliminary Report

M. Gebser and **T. Grote** and **R. Kaminski** and **P. Obermeier**[⋆] and **O. Sabuncu** and **T. Schaub**[*]
Universität Potsdam, Germany and [⋆]DERI Galway, Ireland

Abstract

The advance of Internet and Sensor technology has brought about new challenges evoked by the emergence of continuous data streams. While existing data stream management systems allow for high-throughput stream processing, they lack complex reasoning capacities. We address this shortcoming and elaborate upon an approach to knowledge-intense stream reasoning, based on Answer Set Programming (ASP). The emphasis thus shifts from rapid data processing towards complex reasoning. To accommodate this in ASP, we develop new techniques that allow us to formulate problem encodings dealing with emerging as well as expiring data in a seamless way. We thus provide novel language constructs and modeling approaches for specifying and reasoning with time-decaying logic programs.

Introduction

The advance of Internet and Sensor technology has brought about new challenges evoked by the emergence of continuous data streams, like web logs, mobile locations, or traffic data. While existing data stream management systems (Golab and Özsu 2010) allow for high-throughput stream processing, they lack complex reasoning capacities (Della Valle et al. 2009). We address this shortcoming and introduce an approach to knowledge-intense stream reasoning, based on Answer Set Programming (ASP; (Baral 2003)) as a prime tool for Knowledge Representation and Reasoning (KRR). The emphasis thus shifts from rapid data processing towards complex reasoning, as needed for instance in ambient assisted living, robotics, or scheduling.

However, the sheer amount and continuous flow of information produced by data streams precludes the direct application of ASP, simply because it is designed for singular reasoning from all available information. Unlike this, *"stream reasoning, instead, restricts processing to a certain window of concern, focusing on a subset of recent statements in the stream, while ignoring previous statements."* (Barbieri et al. 2010b). To accommodate this in ASP, we develop new techniques that allow us to formulate problem encodings dealing with emerging as well as expiring data in a seamless way.

[*]Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

To further illustrate this scenario, consider a continuous character stream over alphabet $\{a, b\}$ along with the task of continuously checking whether the stream at hand matches regular expression $(a|b)^*aa$. We represent the stream via atoms of the form `read(C,T)`, indicating that character `C` is at stream position `T`. As a first attempt, we may then encode the recognition of $(a|b)^*aa$ by the rule

```
accept :- read(a,T-1), read(a,T).
```

This rule can be seen as an "offline" encoding, which is correct for the initial segment of a stream of successive instances of predicate `read`, that is, up to the smallest i (if any) such that read(a,i−1) and read(a,i) hold. However, instances of `read` constitute an "online" data flow, and an `accept` decision has to be withdrawn when letter b is read, eg. in read(b,i+1). Clearly, solving such a problem with traditional ASP systems requires relaunching the system upon the arrival of each character. Although each time only the last two readings need to be taken into account, neither of the following ways to utilize standard ASP systems is satisfactory from a KRR viewpoint: (a) one may add further rules to explicitly identify outdated readings (in order not to reason about them) among the whole data; (b) an external component may filter readings and pass only the most recent ones on to the ASP system. Major drawbacks of (a) are the increasing size of input data over time and the more involved encoding, required for the sake of "garbage collection." Compared to this, (b) might appear tempting, but it relies on external filtering and thus fails to model the scenario at hand within the declarative realm of ASP.

To overcome the described insufficiencies, we propose an ASP-based approach to stream reasoning based on the sliding window model (cf. (Golab and Özsu 2010)). The idea is (i) to read an "offline" encoding just once and (ii) to keep only the n last entries of an "online" data stream. We accomplish this by extending our previous approach to reactive ASP (Gebser et al. 2011) by means for dealing with time-decaying program parts. In our example, this implies that instances of predicate `read` expire after two steps. Hence, when investigating the stream $abba$, only the atoms `read(b,3)` and `read(a,4)` are taken into account, while `read(a,1)` and `read(b,2)` have already expired and been disposed of. In fact, time-decaying data poses a major challenge to ASP given that fixed encodings must tolerate emerging as well as expiring data. While standard ASP solving deals with one problem instance at a time, we now face continuously changing instances. Furthermore, the applicability of traditional modeling techniques, eg. frame axioms (Lifschitz 2002), is in question since initial information expires. We address this by proposing novel language constructs that allow for specifying and reasoning with time-decaying logic programs in an effective way. Moreover, we develop modeling techniques that are robust enough to handle changing data without continuous reprocessing or increasing memory demands.

Background

We only provide a brief introduction to the syntax of logic programs with choice rules and integrity constraints, and refer the reader to (Simons, Niemelä, and Soininen 2002) for semantic issues. A *rule* is an expression of the form

$$h \leftarrow a_1, \dots, a_m, \mathit{not}\ a_{m+1}, \dots, \mathit{not}\ a_n \qquad (1)$$

where a_i, for $1 \leq m \leq n$, is an *atom* of the form $p(t_1, \dots, t_k)$, and $t_1, \dots, t_k$ are terms, viz. constants, variables, or functions. For a rule r as in (1), the *head* h of r is either an atom, a *cardinality constraint* of the form $l\,\{h_1, \dots, h_k\}\,u$ in which l, u are integers and $h_1, \dots, h_k$ are atoms, or the special symbol $\bot$. If h is a cardinality constraint, we call r a *choice rule*, and an *integrity constraint* if $h = \bot$. We denote the atoms occurring in h by $\mathit{head}(r)$, ie. $\mathit{head}(r) = \{h\}$ if h is an atom, $\mathit{head}(r) = \{h_1, \dots, h_k\}$ if $h = l\,\{h_1, \dots, h_k\}\,u$, and $\mathit{head}(r) = \emptyset$ if $h = \bot$. (We below skip $\bot$ when writing integrity constraints.) A *logic program* R is a set of rules of the form (1). By $\mathit{atom}(R)$, we denote the set of all atoms occurring in R, and $\mathit{head}(R) = \bigcup_{r \in R} \mathit{head}(r)$ is the collection of head atoms in R. The *ground instance* of R, denoted by $\mathit{grd}(R)$, is the set of all ground rules constructable from rules $r \in R$ by substituting every variable in r with some ground term composed of constants and function symbols from the (implicit) signature of R.

For capturing dynamic systems, we take advantage of *incremental logic programs* (Gebser et al. 2008), consisting of a triple (B, P, Q) of logic programs, among which P and Q contain a (single) parameter t ranging over the positive integers. In view of this, we sometimes denote P and Q by $P[t]$ and $Q[t]$. The base program B is meant to describe static knowledge, independent of parameter t. The role of P is to capture knowledge accumulating with increasing t (eg. a transition function in planning), whereas Q is specific for each value of t (eg. a query). Roughly speaking, we are interested in finding an answer set of the program $R_i = B \cup \bigcup_{1 \leq j \leq i} P[t/j] \cup Q[t/i]$ for some (minimum) integer $i \geq 1$. That is, the cumulative (and static) parts of R are meant to be progressively extended when i increases, while a query persists for just one step i. This is implemented in the incremental ASP solver *iclingo* (Gebser et al. 2008), providing directives like "**`#base`**," "**`#cumulative`**," and "**`#volatile`**" for fixing the roles of program parts. The reactive solver *oclingo* (Gebser et al. 2011) extends this functionality to incorporate external information, via "**`#external`**," from a controller, also distinguishing cumulative and volatile parts. However, stream data often stays in a sliding window for several steps before it can (and should) be discarded, so that it fits neither into the cumulative part P nor the query Q in a natural way. In order to address this shortcoming, we introduce the concept of time-decaying logic programs.

Time-Decaying Logic Programs

To provide a formal account of time-decaying logic programs, subject to emerging and expiring constituents, we rely on *module theory* (Oikarinen and Janhunen 2006) for capturing the continuous composition and decomposition of program parts. To this end, we further extend the incremental and reactive module theory developed in (Gebser et al. 2008; 2011). Also, we introduce directives for specifying the respective modules, leading to an extension of the pre-existing language of *oclingo* (Gebser et al. 2011).

A *time-decaying logic program* Q^l is a logic program Q annotated with a *life span* $l \in \mathbb{N} \cup \{\infty\}$; when $l = \infty$, we often write just Q below. The life span allows for steering the expiration of non-persistent program parts, also called *transients*. To support this flexibility in practice, we augment the *oclingo* language with new directives of the form

```
#volatile t [: l].
```

While `t` indicates the name written for the incremental parameter t in a (schematic) program $Q[t]$, the additional integer `l` gives the life span l of $Q^l[t]$. If `l` is omitted, as in the prior *oclingo* language, it is taken as 1, thus leading to $Q^1[t]$. Reconsidering the introductory example of recognizing $(a|b)^*aa$, the fact that only the last *two* readings deserve attention could now be captured by a time-decaying program $Q^2[t]$ specified as follows:

```
#volatile t : 2.
accept :- read(a,t), read(a,t+1).
```
(2)

For deciding acceptance at a stream position, $Q^2[t]$ involves `read(a,`t`+1)`. Such a reading is yet unavailable when $Q^2[t/i]$ is introduced at a stream position i, but `read(a,`i`)` and `read(a,`i`+1)` together trigger the rule to derive `accept` wrt. the reading at $i{+}1$, while $Q^2[t/i]$ expires once a reading at $i{+}2$ becomes available.

A *time-decaying incremental logic program* is a triple of the form $(B, P[t], \{Q_1^{l_1}[t], \dots, Q_m^{l_m}[t]\})$ in which $B, P[t], Q_1^{l_1}[t], \dots, Q_m^{l_m}[t]$ are time-decaying logic programs. Such an incremental program serves as "offline" encoding of an underlying dynamic system. Note that ordinary incremental logic programs $(B, P[t], Q[t])$ specialize the decaying case to $(B, P[t], \{Q^1[t]\})$.

A *time-decaying online progression*, representing a stream of lasting and transient program parts, is a sequence $(E_i[e_i], \{F_{1_i}^{l_{1_i}}, \dots, F_{m_i}^{l_{m_i}}\}[f_i])_{i \geq 1}$ of pairs in which $E_i, F_{1_i}^{l_{1_i}}, \dots, F_{m_i}^{l_{m_i}}$ are time-decaying logic programs and e_i, f_i are positive integers. The latter represent minimum values assumed for the incremental parameter t in an associated "offline" (incremental) logic program. Note that online progressions in the sense of (Gebser et al. 2011) capture the special case $(E_i[e_i], \{F_{1_i}\}[f_i])_{i\geq 1}$, where a transient F_{1_i} persists for arbitrarily many incremental steps and is superseded only by $F_{1_{i+1}}$. In order to generalize the previous setting, beyond "**#volatile.**" directives, we extended *oclingo*'s (external) controller component to additionally support the following:

```
#volatile : l.
```

As with (transient) incremental logic program parts, the integer `l` gives the life span l of a transient F^l.

Note that decaying the rules of an incremental program, as done in (2), does still not truly capture the sliding window idea that the data expires while the reasoning remains the same. Rather than expiring rules, for our introductory example, we better decay readings in view of the fact that all but the last *two* are irrelevant. For instance, a (time-decaying) online progression representing the stream $abba$ can be provided as follows:

```
#step 1.  #volatile : 2. read(a,1).
#step 2.  #volatile : 2. read(b,2).
#step 3.  #volatile : 2. read(b,3).
#step 4.  #volatile : 2. read(a,4).
```

For $1 \leq i \leq 4$, the value of f_i (and e_i) is given in a "**#step** i." directive, expressing that an underlying incremental program must have progressed to the position i of a reading in the stream. Furthermore, the life span $l_{1_i} = 2$ of transients $F_{1_i}^{l_{1_i}}$ is provided via "**#volatile** : 2." Accordingly, the online progression specified above is as follows:

$$\begin{aligned}\big(\,&(\emptyset[1], \{\{\texttt{read(a,1).}\}^2\}[1]),\\ &(\emptyset[2], \{\{\texttt{read(b,2).}\}^2\}[2]),\\ &(\emptyset[3], \{\{\texttt{read(b,3).}\}^2\}[3]),\\ &(\emptyset[4], \{\{\texttt{read(a,4).}\}^2\}[4])\,\big).\end{aligned} \tag{3}$$

In view of decaying data, rules stemming from the incremental program in (2) could now also be accumulated (when replacing "**#volatile** t : 2." by "**#cumulative** t."), while still preserving the intended meaning that only the last two readings are used for deciding acceptance.

Although the expiration of outdated data and/or rules may yield a working (standard) logic program at each incremental step, a step-wise redefinition of head atoms, as with `accept` in (2), is delicate in ASP and, in particular, for an incremental ASP system like *oclingo*. In fact, the possibility of integrating recent additions without exhaustively reprocessing the entire collection of (non-expired) data and rules requires incrementally gathered program parts to be "compositional." This condition can be expressed in terms of modules (Oikarinen and Janhunen 2006), as elaborated in the following.

For providing a clear interface between various program parts and guaranteeing their compositionality, we build upon the concept of a *module*, $\mathbb{P}$, being a triple (P, I, O) consisting of a ground program P and sets I, O of ground atoms such that $I \cap O = \emptyset$, $atom(P) \subseteq I \cup O$, and $head(P) \subseteq O$. The elements of I and O are called *input* and *output* atoms, also denoted by $I(\mathbb{P})$ and $O(\mathbb{P})$, respectively; similarly, we refer to P by $P(\mathbb{P})$. The *join* of two modules $\mathbb{P}$ and $\mathbb{Q}$, denoted by $\mathbb{P} \sqcup \mathbb{Q}$, is defined as the module

$$(\, P(\mathbb{P}) \cup P(\mathbb{Q}), (I(\mathbb{P}) \setminus O(\mathbb{Q})) \cup (I(\mathbb{Q}) \setminus O(\mathbb{P})), O(\mathbb{P}) \cup O(\mathbb{Q})\,)$$

provided that $O(\mathbb{P}) \cap O(\mathbb{Q}) = \emptyset$ and there is no strongly connected component in the positive dependency graph of $P(\mathbb{P}) \cup P(\mathbb{Q})$ that shares atoms with both $O(\mathbb{P})$ and $O(\mathbb{Q})$. A set A of atoms is an *answer set* of a module $\mathbb{P}$ if A is a (standard) answer set of $P(\mathbb{P}) \cup \{a \leftarrow \mid a \in I(\mathbb{P}) \cap A\}$; we denote the set of all answer sets of $\mathbb{P}$ by $AS(\mathbb{P})$. For two modules $\mathbb{P}$ and $\mathbb{Q}$, the *composition* of their answer sets is $AS(\mathbb{P}) \bowtie AS(\mathbb{Q}) = \{A_{\mathbb{P}} \cup A_{\mathbb{Q}} \mid A_{\mathbb{P}} \in AS(\mathbb{P}), A_{\mathbb{Q}} \in AS(\mathbb{Q}), A_{\mathbb{P}} \cap (I(\mathbb{Q}) \cup O(\mathbb{Q})) = A_{\mathbb{Q}} \cap (I(\mathbb{P}) \cup O(\mathbb{P}))\}$. The module theorem in (Oikarinen and Janhunen 2006) shows that the semantics of $\mathbb{P}$ and $\mathbb{Q}$ is *compositional* if their join is defined, ie. if $\mathbb{P} \sqcup \mathbb{Q}$ is *well-defined*, then $AS(\mathbb{P} \sqcup \mathbb{Q}) = AS(\mathbb{P}) \bowtie AS(\mathbb{Q})$. In ASP solving, compositionality eases adding new rules to a program, as it boils down to combining (without revising) the constraints characterizing answer sets.

For turning programs into modules, we associate a (non-ground) program and a set of (ground) input atoms with a module imposing certain restrictions on the induced ground program. To this end, for a ground program P and a set X of ground atoms, define $P|_X$ as

$$\begin{aligned}&\{h \leftarrow a_1, \dots, a_m, not\ a'_{m+1}, \dots, not\ a'_{n'} \mid\\ &\ h \leftarrow a_1, \dots, a_m, not\ a_{m+1}, \dots, not\ a_n \in P,\\ &\{a_1, \dots, a_m\} \subseteq X, \{a'_{m+1}, \dots, a'_{n'}\} = \{a_{m+1}, \dots, a_n\} \cap X\}.\end{aligned}$$

Note that $P|_X$ projects the bodies of rules in P to the atoms of X. If a body contains an atom outside X, either the corresponding rule or literal is removed, depending on whether the atom occurs positively or negatively. This allows us to associate (non-ground) programs with (ground) modules in the following way.

Definition 1. *Let P^l be a time-decaying logic program, I a set of ground atoms, and k an integer. For $X = I \cup head(grd(P))$ and $Y = I \cup head(grd(P)|_X)$, we define the module*

$$\mathbb{P}^l(I,k) = \begin{cases} (\,\emptyset, & I, head(grd(P)|_X)\,) \quad \text{if } l \leq k;\\ (\,grd(P)|_Y, & I, head(grd(P)|_X)\,) \quad \text{otherwise.}\end{cases}$$

The full ground instantiation, $grd(P)$, of P^l is projected onto inputs and atoms defined in $grd(P)$. The head atoms of this projection, viz. $head(grd(P)|_X)$, serve as output atoms and are used to simplify $grd(P)$, sparing only input and output atoms. If $k < l$, we thus get $P(\mathbb{P}^l(I,k)) = grd(P)|_Y$, while $P(\mathbb{P}^l(I,k)) = \emptyset$, obtained otherwise, reflects the expiration of P^l wrt. k. However, the input-output interface of $\mathbb{P}^l(I,k)$, $I(\mathbb{P}^l(I,k))$ and $O(\mathbb{P}^l(I,k))$, remains unaffected by expiration (thus prohibiting the redefinition of expired head atoms). Furthermore, when $l = \infty$, P^l can never expire, and we below write $\mathbb{P}(I)$ in this case as a shorthand for $\mathbb{P}^\infty(I,k)$.

We now turn to formalizing the modularity of time-decaying incremental logic programs and online progressions. For utilizing the import capacities of modules, we assume that any (non-ground) time-decaying logic program P^l has an associated set I_P of input atoms, used below to obtain ground programs and interfaces of modules $\mathbb{P}^l(I,k)$.

Definition 2. *We define a time-decaying online progression* $(E_i[e_i], \{F_{1_i}^{l_{1_i}}, \dots, F_{m_i}^{l_{m_i}}\}[f_i])_{i\geq 1}$ *as modular wrt. a time-decaying incremental logic program* $(B, P[t], \{Q_1^{l_1}[t], \dots, Q_m^{l_m}[t]\})$ *if the modules*

$$\begin{aligned}
\mathbb{Q}_0 &= \mathbb{B}(I_B)\\
\mathbb{P}_n &= \mathbb{Q}_{n-1} \sqcup \mathbb{P}[t/n](O(\mathbb{Q}_{n-1}) \cup I_{P[t/n]})\\
\mathbb{Q}_n &= \mathbb{P}_n \sqcup \big(\textstyle\bigsqcup_{1\leq g\leq m} \mathbb{Q}_g^{l_g}[t/n](O(\mathbb{P}_n) \cup I_{Q_g[t/n]}, k{-}n)\big)\\
\mathbb{F}_0 &= (\emptyset, \emptyset, \emptyset)\\
\mathbb{E}_n &= \mathbb{F}_{n-1} \sqcup \mathbb{E}_n(O(\mathbb{Q}_{e_n}) \cup I_{E_n})\\
\mathbb{F}_n &= \mathbb{E}_n \sqcup \big(\textstyle\bigsqcup_{\substack{1\leq h\leq m_n\\ \text{and } l_{h_n}\neq\infty}} \mathbb{F}_{h_n}^{l_{h_n}}(O(\mathbb{Q}_{f_n}) \cup I_{F_{h_n}}, k{-}f_n)\big)\\
\mathbb{R}_{j,k} &= \mathbb{Q}_k \sqcup \mathbb{F}_j \sqcup \big(\textstyle\bigsqcup_{\substack{1\leq h\leq m_j\\ \text{and } l_{h_j}=\infty}} \mathbb{F}_{h_j}(O(\mathbb{Q}_{f_j}) \cup I_{F_{h_j}})\big)
\end{aligned}$$

are well-defined for all $j, k \geq 1$ *with* $e_1, f_1, \dots, e_j, f_j \leq k$.

The module $\mathbb{R}_{j,k}$ represents the combination of the accumulated "offline" encoding $\mathbb{Q}_k$ (with horizon k) and data gathered in online progressions $\mathbb{F}_j$ and $\mathbb{F}_{h_j}^{(\infty)}$ up to element j. The definition requires modules generated upon applying an incremental program to an online progression as well as their joins to be well-defined. The latter guarantees a compositional semantics enabling an "additive" step-wise integration of modules. Expiration of transients $Q_g^{l_g}[t/n]$ (or $F_{h_n}^{l_{h_n}}$) is reflected by using $k{-}n$ (or $k{-}f_n$) for deriving a corresponding module. For instance, when l_g is 1, $k{-}n < 1$ only holds for n matching the current step k (and $n > k$ unused in $\mathbb{R}_{j,k}$), while the empty program is obtained for smaller n (cf. Definition 1). Furthermore, the special handling of singular transients $F_{h_j}^{(\infty)}$ in $\mathbb{R}_{j,k}$ admits their withdrawal when proceeding to the $j{+}1$-th element of an online progression; if withdrawal is unintended, the (lasting) event E_j allows for the accumulation of rules.

As a (negative) example, reconsider the time-decaying program $Q^2[t]$ from (2), and let $I_{Q[t]} = \{\texttt{read(a,}n\texttt{)} \mid n \in \{t, t{+}1\}\}$. Then, $(\emptyset, \emptyset, \{Q^2[t]\})$ cannot be combined modularly with streams; eg. for $\mathbb{R}_{j,2}$, the intersecting outputs $O(\mathbb{Q}^2[t/1](I_{Q[t/1]}, 2{-}1)) = O(\mathbb{Q}^2[t/2](I_{Q[t/2]}, 2{-}2)) = \{\texttt{accept}\}$ yield an undefined join. Note, however, that inputs like $I_{Q[t]}$ can be declared in the *oclingo* language as follows: `#external read(a,t;t+1).`

As an alternative incremental program for the recognition of $(a|b)^*aa$, let us consider the following specification:

```
#base.              #external read(a,1).
{ accept }.
#volatile t : 2. #external read(a,t+1).
:- read(a,t), read(a,t+1), not accept.
:- accept, not read(a,t).
```

Denoting the program part in-between "`#base.`" and "`#volatile t : 2.`" by B and the remaining part by $Q^2[t]$, the program induces modules of the form

$$\begin{aligned}
&\mathbb{B}(\{\texttt{read(a,1)}\}) = (B, \{\texttt{read(a,1)}\}, \{\texttt{accept}\})\\
&\mathbb{Q}^2[t](O(\mathbb{Q}_{t-1}) \cup \{\texttt{read(a,}t{+}1\texttt{)}\}) =\\
&\quad (Q, O(\mathbb{Q}_{t-1}) \cup \{\texttt{read(a,}t{+}1\texttt{)}\}, \emptyset)
\end{aligned}$$

where either $Q = \emptyset$ or $Q = Q[t/n]$, depending on whether $k{-}n < 2$ for $\mathbb{R}_{j,k}$ and $1 \leq n \leq k$. Observe that the atom `accept` is now subject to a choice rule in B in order to avoid (non-modular) redefinitions within $Q^2[t]$; rather, the synchronization of `accept` with stream readings is accomplished via integrity constraints. In fact, B and $Q^2[t]$ induce a well-defined sequence of (joined) modules, where $O = \{\texttt{accept}\}$, as part of $\mathbb{R}_{4,4}$:

$$\begin{aligned}
\mathbb{Q}_0 &= (B, \{\texttt{read(a,1)}\}, O)\\
\mathbb{Q}_1 &= (P(\mathbb{Q}_0), \{\texttt{read(a,2)}\} \cup I(\mathbb{Q}_0), O)\\
\mathbb{Q}_2 &= (P(\mathbb{Q}_1), \{\texttt{read(a,3)}\} \cup I(\mathbb{Q}_1), O)\\
\mathbb{Q}_3 &= (P(\mathbb{Q}_2) \cup Q[t/3], \{\texttt{read(a,4)}\} \cup I(\mathbb{Q}_2), O)\\
\mathbb{Q}_4 &= (P(\mathbb{Q}_3) \cup Q[t/4], \{\texttt{read(a,5)}\} \cup I(\mathbb{Q}_3), O).
\end{aligned}$$

The result $\mathbb{Q}_4$ can also be joined with the combined module

$$\begin{aligned}
\mathbb{F}_4 = (&\{\texttt{read(b,3).\ \ read(a,4).}\}, \{\texttt{accept}\},\\
&\{\texttt{read(a,1),read(b,2),read(b,3),read(a,4)}\})
\end{aligned}$$

obtained from the online progression in (3). In view of the integrity constraint "`:- accept, not read(a,3).`" in $Q[t/3]$, `accept` must not belong to an answer set of $\mathbb{R}_{4,4} = \mathbb{Q}_4 \sqcup \mathbb{F}_4$ such that the input atoms `read(a,2)`, `read(a,3)`, and `read(a,5)` of $\mathbb{R}_{4,4}$ are false. However, "`:- read(a,3), read(a,4), not accept.`" in $Q[t/3]$ would enforce `accept` to hold if the third reading was `read(a,3)`.

Modeling and Reasoning

In the extended version of this paper (Gebser et al. 2012), we present several extensive case studies illustrating particular features in modeling and reasoning with time-decaying logic programs and stream data. This includes modelings of the simple task to monitor consecutive user accesses, an overtaking scenario utilizing frame axioms, and the combinatorial problem of online job scheduling.

Related Work

Academic and commercial systems for stream processing include, on the one hand, high-throughput stream processors, like *Aurora* (Abadi et al. 2003), IBM's *System S* (Gedik et al. 2008), or *C-SPARQL* (Barbieri et al. 2009). All these systems essentially operate on the level of continuous conjunctive queries (without recursion). Although they fall short in terms of expressiveness, they are highly optimized for dealing with enormous amounts of stream data (as eg. in stock exchange). Rule-based stream reasoners, on the other hand, address this lack of expressiveness. For instance, Barbieri et al. (2010a) devise a reasoning engine with ontological background knowledge, which amounts to pure Datalog. Also, they employ a time-based sliding window by annotating incoming data with fixed expiration times. Interestingly, the stream reasoner *ETALIS* (Anicic et al. 2010) provides a declarative rule-based language for complex event processing (implemented in Prolog). ETALIS associates propositions with time intervals; complex events are formed via interval operators, like *meets* or *during*. Given that ETALIS relies on unification, it is interesting future work to see in how far its functionalities can be transferred to a grounding approach based on ASP. Finally, ASP was used in (Do, Loke, and Liu 2011) for a case study in processing OWL data streams. However, this proposal does not integrate the treatment of stream data into ASP, but rather calls anew an ASP solver (here *dlv* (Leone et al. 2006)) on each window. Unlike this, our approach handles time-decaying data (and programs) within the reasoning methodology of ASP, which distinguishes it from other approaches to stream reasoning.

Summary

We introduced the first genuine approach to stream reasoning in ASP. Our approach is of general purpose offering interesting prospects for implementing higher forms of dynamic reasoning, as in agent technology, belief revision and update, cognitive robots, forgetting, etc. Technically, the emergence and expiration of program parts presented a significant challenge to traditional ASP. We addressed this by starting from semantic principles and developing language extensions for specifying time-decaying program parts. Our approach is implemented within the reactive ASP solver *oclingo* (oclingo), which is a central component in the EU projects `www.easyreach-project.eu` and `www.strokeback.eu` relying on knowledge-intense stream reasoning in eHealth.

Acknowledgments. This work was partially funded by DFG under grant SCHA 550/8-1/2 and by the EU project EasyReach under AAL call 2009-2.

References

Abadi, D.; Carney, D.; Çetintemel, U.; Cherniack, M.; Convey, C.; Lee, S.; Stonebraker, M.; Tatbul, N.; and Zdonik, S. 2003. Aurora: A new model and architecture for data stream management. *VLDB Journal* 12(2):120–139.

Anicic, D.; Fodor, P.; Rudolph, S.; Stühmer, R.; Stojanovic, N.; and Studer, R. 2010. A rule-based language for complex event processing and reasoning. In *Proceedings of the Fourth International Conference on Web Reasoning and Rule Systems (RR'10)*, 42–57. Springer.

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

Barbieri, D.; Braga, D.; Ceri, S.; Della Valle, E.; and Grossniklaus, M. 2009. C-SPARQL: SPARQL for continuous querying. In *Proceedings of the Eighteenth International Conference on World Wide Web (WWW'09)*, 1061–1062. ACM Press.

Barbieri, D.; Braga, D.; Ceri, S.; Della Valle, E.; and Grossniklaus, M. 2010a. Incremental reasoning on streams and rich background knowledge. In *Proceedings of the Seventh Extended Semantic Web Conference (ESWC'10)*, 1–15. Springer.

Barbieri, D.; Braga, D.; Ceri, S.; Della Valle, E.; Huang, Y.; Tresp, V.; Rettinger, A.; and Wermser, H. 2010b. Deductive and inductive stream reasoning for semantic social media analytics. *IEEE Intelligent Systems* 25(6):32–41.

Della Valle, E.; Ceri, S.; van Harmelen, F.; and Fensel, D. 2009. It's a streaming world! reasoning upon rapidly changing information. *IEEE Intelligent Systems* 24(6):83–89.

Do, T.; Loke, S.; and Liu, F. 2011. Answer set programming for stream reasoning. In *Proceedings of the Twenty-fourth Canadian Conference on Artificial Intelligence (AI'11)*, 104–109. Springer.

Gebser, M.; Kaminski, R.; Kaufmann, B.; Ostrowski, M.; Schaub, T.; and Thiele, S. 2008. Engineering an incremental ASP solver. In *Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08)*, 190–205. Springer.

Gebser, M.; Grote, T.; Kaminski, R.; and Schaub, T. 2011. Reactive answer set programming. In *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, 54–66. Springer.

Gebser, M.; Grote, T.; Kaminski, R.; Obermeier, P.; Sabuncu, O.; and Schaub, T. 2012. Stream reasoning with answer set programming: Extended version. Unpublished draft. Available at (oclingo).

Gedik, B.; Andrade, H.; Wu, K.; Yu, P.; and Doo, M. 2008. SPADE: The System S declarative stream processing engine. In *Proceedings of the International Conference on Management of Data (SIGMOD'08)*, 1123–1134. ACM Press.

Golab, L., and Özsu, M. 2010. *Data Stream Management*. Morgan and Claypool Publishers.

Leone, N.; Pfeifer, G.; Faber, W.; Eiter, T.; Gottlob, G.; Perri, S.; and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic* 7(3):499–562.

Lifschitz, V. 2002. Answer set programming and plan generation. *Artificial Intelligence* 138(1-2):39–54.

oclingo. `http://www.cs.uni-potsdam.de/oclingo`.

Oikarinen, E., and Janhunen, T. 2006. Modular equivalence for normal logic programs. In *Proceedings of the Seventeenth European Conference on Artificial Intelligence (ECAI'06)*, 412–416. IOS Press.

Simons, P.; Niemelä, I.; and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence* 138(1-2):181–234.