Stream Reasoning with Answer Set Programming: Extended Version*

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, as needed for instance in ambient assisted living, robotics, or scheduling. 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.

1 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 elaborate upon 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; (Lifschitz et al. 2008)). 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.

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

\[\text{accept} := \text{read}(a, T-1), \text{read}(a, T)\]

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 \(\text{read}(a, i-1)\) and \(\text{read}(a, i)\) hold. However, instances of read constitute an

* This paper extends a preliminary short version (Gebser et al. 2012) appearing at KR’12.
“online” data flow, and an accept decision has to be withdrawn when letter \( b \) is read, eg. in \( \text{read}(b, i+1) \). Clearly, solving such a problem with traditional ASP systems, like \( dlv \) (Leone et al. 2006) or \( clingo \) (Gebser et al. 2011b), 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. 2011a) 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 \( \text{read}(b, 3) \) and \( \text{read}(a, 4) \) are taken into account, while \( \text{read}(a, 1) \) and \( \text{read}(b, 2) \) 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.

2 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 et al. 2002) for details on semantics.\(^1\) A rule is an expression of the form

\[
\text{h} \leftarrow a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n.
\]

where \( a_i \), for \( 1 \leq m \leq n \), is an atom of the form \( p(t_1, \ldots, t_k) \), and \( t_1, \ldots, t_k \) 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 \( l \{ h_1, \ldots, h_k \} u \) in which \( l, u \) are integers and \( h_1, \ldots, h_k \) 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 head \((r)\), ie. head \((r)\) = \( \{h\} \) if \( h \) is an atom, head \((r)\) = \( \{h_1, \ldots, h_k\}\) if \( h = l \{h_1, \ldots, h_k\} u \), and 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 atom \((R)\), we denote the set of all atoms occurring in \( R \), and head \((R)\) = \( \bigcup_{r \in R} \text{head}(r) \) is the collection of head atoms in \( R \). The ground instance of \( R \), denoted by \( \text{grd}(R) \), is the set of all ground rules constructable

\(^1\) For simplicity, we here limit the attention to cardinality constraints occurring in heads of rules. In practice, cardinality as well as weight constraints can occur likewise in rule heads and bodies.
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 \( R \).\(^2\)

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 formal semantics is given in terms of a dedicated module theory. The reactive solver oclingo (Gebser et al. 2011a) 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.

3 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; Gebser et al. 2011a). Also, we introduce directives for specifying the respective modules, leading to an extension of the preexisting language of oclingo (Gebser et al. 2011a).

A time-decaying logic program \( Q^T \) 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

\[
\text{#volatile } t : \{1\}.
\]

While \( t \) indicates the name written for the incremental parameter \( t \) in a (schematic) program \( Q[t] \), the additional integer \( i \) gives the life span \( l \) of \( Q^T[t] \). If \( i \) 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:

\[
\text{#volatile } t : 2. \text{ accept} :- \text{read}(a,t), \text{read}(a,t+1).
\]

For deciding acceptance at a stream position, \( Q^2[t] \) involves \( \text{read}(a,t+1) \). Such a reading is yet unavailable when \( Q^2[t/i] \) is introduced at a stream position \( i \), but \( \text{read}(a,i) \) and \( \text{read}(a,i+1) \) together trigger the rule to derive \text{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 \((B, P[t], \{Q^1[i][t], \ldots, Q^m[i][t]\})\) in which \( B, P[t], Q^1[i][t], \ldots, Q^m[i][t] \) are time-decaying logic programs. Such

\(^2\) We also assume some familiarity with built-ins of grounders like gringo (Gebser et al. 2011c), eg. ‘\( *\)’, ‘\( !=\)’, and arithmetic functions, which are evaluated upon instantiation.
an incremental program serves as “offline” encoding of an underlying dynamic system. While ordinary incremental logic programs \((B, P[t], Q[t])\) specialize the decaying case to \((B, P’t, \{Q’[t]\})\), the life spans \(l_1, \ldots, l_m\) can diverge from 1 and one another.

A time-decaying online progression, representing a stream of lasting and transient program parts, is a sequence \(\{E_i[e_i], \{F^1_i, \ldots, F^m_i\} [f_i]\}_{i \geq 1}\) of pairs in which \(E_i, F^1_i, \ldots, F^m_i\) 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. 2011a) capture the special case \((E_i[e_i], \{F_i\} [f_i])_{i \geq 1}\), where a transient \(F_i\) persists for arbitrarily many incremental steps and is superseded only by \(F_{i+1}\). In order to generalize the previous setting, beyond ”\#volatile.” directives, we extended oclingo’s (external) controller component to additionally support the following:

\[
\text{\#volatile : 1.}
\]

As with (transient) incremental logic program parts, 1 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:

\[
\begin{align*}
\text{\#step 1.} & \quad \text{\#volatile : 2.} & \text{read}(a,1). \\
\text{\#step 2.} & \quad \text{\#volatile : 2.} & \text{read}(b,2). \\
\text{\#step 3.} & \quad \text{\#volatile : 2.} & \text{read}(b,3). \\
\text{\#step 4.} & \quad \text{\#volatile : 2.} & \text{read}(a,4).
\end{align*}
\]

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 \(l_{i+1} = 2\) of transients \(F^l_{i+1}\) is provided via “\#volatile : 2.” Accordingly, the online progression specified above is as follows:

\[
\begin{align*}
&\quad (\{\emptyset[1]\}, \{\{\text{read}(a,1).\}^2\}[1]), (\emptyset[2], \{\{\text{read}(b,2).\}^2\}[2]), \\
&\quad (\emptyset[3], \{\{\text{read}(b,3).\}^2\}[3]), (\emptyset[4], \{\{\text{read}(a,4).\}^2\}[4])\).
\end{align*}
\]

(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. Also note that the possibility of associating stream data with a life span (not fixed to 1) is essential to provide “automatic” reasoning support for sliding windows. If this possibility were unavailable, either the whole window contents would need to be provided as transient online input at each step, thus “replaying” part of the data when windows overlap, or rules referring to persistently added data would have to be deactivated once the data “expires.” While the former workaround incurs redundancy, the latter approach basically means to simulate expiration via input atoms (see below) switching off outdated rules. As such workarounds reintroduce the drawbacks of standard ASP systems (explained in the introductory section), the extension of oclingo’s language and reasoning capacities to include life spans properly increases the expressiveness of reactive ASP.

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 gath-
Stream Reasoning with Answer Set Programming: Extended Version

5

ered 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, \( P \), being a triple \((P, I, O)\) consisting of a ground program \( P \) and sets \( I, O \) of ground atoms such that \( I \cap O = \emptyset \), \( \text{atoms}(P) \subseteq I \cup O \), and \( \text{head}(P) \subseteq O \). The elements of \( I \) and \( O \) are called input and output atoms, also denoted by \( I(P) \) and \( O(P) \), respectively; similarly, we refer to \( P \) by \( P(P) \). The join of two modules \( P \) and \( Q \), denoted by \( P \sqcup Q \), is defined as the module

\[
(P(P) \cup P(Q), (I(P) \setminus O(Q)) \cup (I(Q) \setminus O(P)), O(P) \cup O(Q))
\]

provided that \( O(P) \cap O(Q) = \emptyset \) and there is no strongly connected component in the positive dependency graph of \( P(P) \cup P(Q) \) that shares atoms with both \( O(P) \) and \( O(Q) \).

A set \( A \) of atoms is an answer set of a module \( P \) if \( A \) is a (standard) answer set of \( P(P) \cup \{ a \leftarrow a \in I(P) \cap A \} \); we denote the set of all answer sets of \( P \) by \( \text{AS}(P) \). For two modules \( P \) and \( Q \), the composition of their answer sets is \( \text{AS}(P) \times \text{AS}(Q) = \{ A_P \cup A_Q \mid A_P \in \text{AS}(P), A_Q \in \text{AS}(Q) \} \). The module theorem in (Oikarinen and Janhunen 2006) shows that the semantics of \( A \) modules is compositional if their join is defined, i.e., if \( P \sqcup Q \) is well-defined, then \( \text{AS}(P \sqcup Q) = \text{AS}(P) \times \text{AS}(Q) \). In ASP solving, compositionality eases adding new rules to a program, as it boils down to combining (without revising) the constraints characterizing answer sets. The solving component of \textit{oclingo} exploits this to successively integrate rules without large overhead; in particular, strongly connected components are only calculated locally once a new program part is added. As a consequence, the compliance of models computed by \textit{oclingo} with answer sets of \( P(P) \sqcup P(Q) \) relies on \( P \sqcup Q \) to be well-defined. Otherwise, \textit{oclingo}'s lightweight incremental processing cannot guarantee meaningful outcomes, and relaunching an ASP system from scratch would be required instead.

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

\[
\{ h \leftarrow a_1, \ldots, a_m, \text{not } a_{m+1}, \ldots, \text{not } a_n \mid h \leftarrow a_1, \ldots, a_m, \text{not } a_{m+1}, \ldots, \text{not } a_n \in P, \\
\{ a_1, \ldots, a_m \} \subseteq X, \{ a_{m+1}, \ldots, a_n \} \cap X \}
\]

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 \( P^i \) be a time-decaying logic program, \( I \) a set of ground atoms, and \( k \) an integer. For \( X = I \cup \text{head}(\text{grd}(P)) \) and \( Y = I \cup \text{head}(\text{grd}(P)|_X) \), we define the module

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

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

**Definition 2**

We define a time-decaying online progression \( (E_1[e_1], \{F_{1j}^{t_1}, \ldots, F_{mj}^{t_m}\}[f_i])_{i \geq 1} \) as modular wrt. a time-decaying incremental logic program \( (B, P[t], \{Q_1[t], \ldots, Q_n[t]\}) \) if the modules

\[
\begin{align*}
Q_0 &= B(I_B) \\
P_n &= Q_{n-1} \cup P[t/n](O(Q_{n-1}) \cup I_{P[t/n]}) \\
Q_n &= P_n \cup \left( \bigcup_{1 \leq j \leq m} Q_j^{t/n}(O(P_n) \cup I_{Q_j[t/n]}, k-n) \right) \\
F_0 &= (\emptyset, \emptyset, \emptyset) \\
E_n &= F_{n-1} \cup E_n(O(Q_{e_n}) \cup I_{E_n}) \\
F_n &= E_n \cup \left( \bigcup_{1 \leq h \leq m_n} F_{h, n}^{t/n} \left( O(Q_{f_n}) \cup I_{F_{h, n}}, k-f_n \right) \right) \\
R_{j,k} &= Q_k \cup F_j \cup \left( \bigcup_{1 \leq l \leq m_j} F_{l, j}^{t/n} \left( O(Q_{l, j}) \cup I_{F_{l, j}} \right) \right)
\end{align*}
\]

are well-defined for all \( j, k \geq 1 \) such that \( e_1, f_1, \ldots, e_j, f_j \leq k \).

The module \( R_{j,k} \) represents the combination of the accumulated “offline” encoding \( Q_k \) (with horizon \( k \)) and data gathered in online progressions \( F_j \) and \( F_{h, j}^{(\infty)} \) 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 \( Q_j^{t/n} \) or \( F_{h, j}^{t/n} \) is reflected by using \( k-n \) (or \( k-f_n \)) for deriving a corresponding module. For instance, when \( I_j = 1 \), \( k-n < 1 \) only holds for \( n \) matching the current step \( k \) (and \( n > k \) unused in \( R_{j,k} \)), while the empty program is obtained for smaller \( n \) (cf. Definition 1). Furthermore, the special handling of singular transients \( F_{h, j}^{(\infty)} \) in \( R_{j,k} \) admits their withdrawal when proceeding to the \( j+1 \)-th element of an online progression; if withdrawal is unintended, the (last) 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 \( I_{Q_2} = \{ \text{read}(a, t) \mid t \in \{ t+1 \} \} \). Then, \( (\emptyset, \emptyset, \{Q_2[t]\}) \) cannot be combined modularly with streams; eg. for \( R_{j,2} \), the intersecting outputs \( O(Q_2[t/1](I_{Q_2[t/1]}, 2-1)) = O(Q_2[t/2](I_{Q_2[t/2]}, 2-2)) = \{\text{accept}\} \) yield an undefined join. Note, however, that inputs like \( I_{Q_2} \) can be declared in the ocingo language as follows:

```ocingo
#external read(a,t;t+1).
```

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

```ocingo
#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

\[
\begin{align*}
B(\{\text{read}(a,1)\}) &= (B, \{\text{read}(a,1)\}, \{\text{accept}\}) \\
Q_2[t](O(Q_{t-1}) \cup \{\text{read}(a,t+1)\}) &= (Q, O(Q_{t-1}) \cup \{\text{read}(a,t+1)\}, \emptyset)
\end{align*}
\]

where either \( Q = \emptyset \) or \( Q = Q[t/n] \), depending on whether \( k-n < 2 \) for \( R_{j,k} \) 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 as part of $\mathbb{R}_{4,4}$:

\begin{align*}
Q_0 &= \{B, \{\text{read}(a, 1)\}, \{\text{accept}\}\} \\
Q_1 &= \{P(Q_0), \{\text{read}(a, 2)\} \cup I(Q_0), \{\text{accept}\}\} \\
Q_2 &= \{P(Q_1), \{\text{read}(a, 3)\} \cup I(Q_1), \{\text{accept}\}\} \\
Q_3 &= \{P(Q_2) \cup Q[t/3], \{\text{read}(a, 4)\} \cup I(Q_2), \{\text{accept}\}\} \\
Q_4 &= \{P(Q_3) \cup Q[t/4], \{\text{read}(a, 5)\} \cup I(Q_3), \{\text{accept}\}\}.
\end{align*}

The result $Q_4$ can also be joined with the combined module

$$F_4 = \{(\{\text{read}(b, 3), \{\text{read}(a, 4)\}, \{\text{accept}\}, \\
\{\text{read}(a, 1), \text{read}(b, 2), \text{read}(b, 3), \text{read}(a, 4)\})\}$$

obtained from the online progression in (3). In view of “:- accept, not read(a, 3).”
in $Q[t/3]$, accept must not belong to an answer set of $\mathbb{R}_{4,4} = Q_4 \cup F_4$ such that
the input atoms $\text{read}(a, 2)$, $\text{read}(a, 3)$, and $\text{read}(a, 5)$ 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). We further elaborate upon the modeling of (modular)
time-decaying incremental logic programs and online progressions in the next section.

## 4 Modeling and Reasoning

The case studies provided below illustrate particular features in modeling and reasoning
with time-decaying logic programs and stream data. We begin with modelings of the simple
task to monitor consecutive user accesses, proceed with an overtaking scenario utilizing
frame axioms, and then turn to the combinatorial problem of online job scheduling. The
Corresponding encodings can be downloaded at (oclingo).

### 4.1 Access Control

Our first scenario considers users attempting to access some service, for instance, by log-
ing in via a website. Access attempts can be denied or granted, eg. depending on the sup-
plied password, and a user account is (temporarily) closed in case of three access denials in
a row. An (initial) stream segment of access attempts is shown in Listing 1. It provides data
about three users, called alice, bob, and claud. As specified by “#volatile : 3.”
(for each step), the life span of access data is limited to three incremental steps, aiming at
an (automatic) reopening of closed user accounts after some waiting period has elapsed.
Furthermore, we assume that time stamps provided in the third argument of facts over
access/3 deviate from the number $i$ in “#step i.” by at most 2; that is, the terms used
in transient facts are coupled to the step number (in an underlying incremental program).

\begin{table}

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>bob</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>claud</td>
<td>[3]</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

For instance, observe that the three denied accesses by bob logged in the second and fourth
step are consecutive in view of the time stamps 3, 2, and 4 provided as argument values,
eg. in access(bob, denied, 3) expiring at step 5.

Our first “offline” encoding is shown in Listing 2. To keep the concrete sliding window
Listing 1. Stream of user accesses with life span of 3 steps

```
#step 1. volatile : 3.
access(alice, granted, 1).
#step 2. volatile : 3.
access(alice, denied, 3).
access(bob, denied, 3).
#step 3. volatile : 3.
access(alice, denied, 2).
access(claude, granted, 5).
#step 4. volatile : 3.
access(alice, denied, 2).
access(bob, denied, 2).
access(bob, denied, 4).
access(claude, denied, 2).
access(claude, denied, 3).
access(claude, denied, 4).
#step 5. volatile : 3.
access(alice, denied, 4).
access(bob, denied, 2).
access(claude, denied, 3).
access(claude, denied, 4).
#step 6. volatile : 3.
access(alice, denied, 6).
#step 7. volatile : 3.
access(alice, denied, 8).
#step 8. volatile : 3.
access(alice, denied, 7).
```

width, matching the life span of stream transients, adjustable, the constant \textit{window} is introduced in Line 1 (and set to default value 3). Similarly, the maximum deviation of time stamps from incremental step numbers and the threshold of consecutive denied accesses at which an account is closed are represented by the constants \textit{offset} and \textit{denial}. After introducing the aforementioned users and the possible outcomes of their access attempts via facts, the static \#base part includes in Line 6 a choice rule on whether the status of a user account is \textit{closed}, while it remains \textit{open} otherwise. In fact, the non-static parts of the incremental program solve the task to identify the current status of the account of a user \textit{U} and represent it by including either \textit{account(U,closed)} or \textit{account(U,open)} in an answer set, yet without redefining \textit{account} over steps. This makes sure that step-wise (ground) incremental program parts are compositional, i.e. that their joins are well-defined according to Definition 2.

The encoding in Listing 2 (mainly) relies on accumulating rules given below \#cumulative \textit{t}." (in Line 9), thus resembling (incremental) planning encodings (cf. (Gebser et al. 2008)) based on a history of actions. In order to react to external inputs, the \#external directive in Line 10 declares (undefined) atoms as input that can be provided by the environment, which is a stream of user access attempts here. Note that ground instances of \textit{access/3} with time stamp \textit{t+offset} (where \textit{offset} is the maximum deviation from \textit{t}) are introduced in an incremental step \textit{t}; in this way, ground rules are prepared for (forward) shifted access data arriving early. The rules in Line 11–13 implement the counting of consecutive denied access attempts per user, up to the threshold given by \textit{denial}; if this threshold is reached (wrt. non-expired access data), the account of the respective user is temporarily closed. The “positive” conclusion from \textit{denial} many denied access attempts to closing an account is encoded via the integrity constraint in Line 15, while more care is needed in concluding the opposite: the right decision whether to leave an account open can be made only after inspecting the whole contents of the current window. To this end, the rule in Line 14 passes information about the threshold being reached on to later steps, and the query in Line 18, refuting an account to be closed if there were no three consecutive denied access attempts by a user, is included only for the (currently) last incremental step.

For the stream segment in Listing 1, in the fourth incremental step, \textit{denied(bob,3,4)} is derived in view of (transient) facts \textit{access(bob,denied,3)}, \textit{access(bob,denied,2)}, and \textit{access(bob,denied,4)}. In view of the integrity
Listing 2. Cumulative access control encoding

1  #const window=3. #const offset=2. #const denial=3. #iinit 1-offset.
2  
3  #base.
4  user(bob;alice;claude). % some users
5  signal(denied;granted). % some signals
6  { account(U,closed) : user(U) }.
7  account(U,open) :- user(U), not account(U,closed).
8  
9  #cumulative t.
10  #external access(U,S,t+offset) : user(U) : signal(S).
11  denied(U,1, t) :- access(U,denied,t+offset).
12  denied(U,N+1, t) :- access(U,denied,t+offset),
13       denied(U,N,t-1), N < denial.
14  denied(U,denial,t) :- denied(U,denial,t-1).
15  :- denied(U,denial,t), not account(U,closed).
16  
17  #volatile t.
18  :- account(U,closed), not denied(U,denial,t).

Listing 3. Volatile access control encoding

1  #const retain=window+2*offset.
2  #volatile t : retain.
3  #external access(U,S,t+offset) : user(U) : signal(S).
4  denied(U,1, t) :- access(U,denied,t+offset).
5  denied(U,N+1, t) :- access(U,denied,t+offset),
6       denied(U,N,t-1), N < denial.
7  denied(U,denial,t) :- denied(U,denial,t-1).
8  :- denied(U,denial,t), not account(U,closed).
9  :- account(U,closed), not denied(U,denial,T) : T := (t,retain)+1..t.

Constraint in Line 15 of Listing 2, this enforces account(bob,closed) to hold. Since access(bob,denied,3) expires at step 5, we can then no longer derive denied(bob,3,4), and denied(bob,3,5) does not hold either; the query in Line 18 thus enforces account(bob,closed) to be false in the fifth incremental step. Similarly, we have that account(claude,closed) or account(alice,closed) hold at step 5, 6, and 8, respectively, but are enforced to be false at any other step. In fact, albeit the semantics of modules principally admits input atoms to assume “arbitrary” truth values (coherent with a module’s rules), oclingo disambiguates this situation by fixing (undefined) input atoms to be false. Furthermore, one may have noticed that, given the values of constants in Listing 2, the consideration of atoms over access/3 starts at t+offset = 3 for t = 1. However, rather than filling up initial contents via additional rules in the #base program part, we added the “#iinit” directive to oclingo for initializing a window wrt. the “past.” By using 1-offset = −1, once the first element of an online progression is processed, ground rules treating access data with time stamps 1, 2, and 3 are readily available, which is exactly the range admitted at the first step. Likewise, at a step like 6, the range of admissible time stamps starts at 4, and it increases at later steps; that is, inputs with time stamp 3, declared at step 1, are not supposed to be provided anymore in stream data. To enable permanent program simplifications setting such atoms to false, an online progression can include directives of the form “#forget i.” within its elements, and they are provided in practice to dispose of elapsed input atoms (cf. (oclingo)).

While, for simplicity, our first encoding accumulated rules (simplified only in view of #forget directives) over expiring data, we now turn to an alternative approach in which outdated rules expire along with stream data. This second encoding, shown in Listing 3, performs the same task as the previous one, and its first part, including constant declarations and #base (up to Line 7), is as before. However, all remaining rules belong to time-
decaying programs of fixed life span, making use of the observation that a time stamp \( i \) is introduced at step \( t = i - \text{offset} \) and last referenced at step \( i + \text{offset} + \text{window} - 1 \). In view of this, the (shortest) sufficient life span is calculated in Line 8 of Listing 3 and associated with a \#volatile program part including all rules below Line 9. In order to avoid using another \#volatile block of life span 1, as included in Listing 2, the basic idea is to count denied access attempts downstream, so that the oldest non-expired program part at a step has an overview of the whole current window contents. This approach is implemented by the rules in Line 12–15, where counting relies on \text{denied}/3 instances from the next step, serving as (additional) inputs declared in Line 11. However, since the incremental parameter \( t \) is included in each head atom, we still have that the atoms defined in different steps do not overlap, so that incremental program parts stay compositional. To make the right decisions about whether to close accounts in view of the outcomes of consecutive denial counting, the integrity constraint in Line 16 can be used unchanged (cf. Line 15 in Listing 2). Unlike this, the integrity constraint in Line 17, enforcing atoms of the form \text{account}(U, \text{closed}) to be false, now checks renouncement of the denial threshold over all (possibly) non-expired preceding program parts; in this way, the oldest program part is guaranteed to be among the investigated ones.

Both encodings presented before have the drawback that a logic program modeling the task at hand is (partially) replaced at each step. While this is tolerable in the simple access scenario, for more complex problems, it means that the internal problem representation of a solving component changes significantly when processing stream data, so that a potential reuse of learned constraints is limited. In order to re-enable constraint learning, we next demonstrate modeling approaches allowing for the preservation of a (static) problem representation in view of fixed window capacity.

At the beginning (up to Line 5), our static access control encoding, shown in Listing 4, is similar to the cumulative approach (cf. Listing 2), while the \#base part is significantly extended in the sequel. In fact, the constant \text{modulo}, calculated in Line 6, takes into account that at most \text{window} many consecutive elements of an online progression are jointly available (preceding ones are expired) and that terms representing time stamps may deviate by up to \text{offset} (both positively and negatively) from incremental step numbers. Given this, \text{window}+2*\text{offset} slots are sufficient to accommodate all distinct time stamps provided as argument values in instances of \text{access}/3 belonging to a window, and one additional slot is added in Line 6 as separator between the largest and the smallest (possibly) referenced time stamp. The available slots are then arranged in a cycle (via modulo arithmetic) in Line 7, and the counting of denied access attempts per user, implemented in Line 9–11, traverses consecutive slots according to instances of predicate \text{next}/2. Importantly, counting does not rely on transient external inputs, ie. instances of \text{access}/3, but instead refers to instances of \text{baseaccess}/3, provided by the choice rule in Line 8; in this way, the atoms and rules in the \#base program part can be reused in determining the status of user accounts wrt. transient access data from a stream.

To nonetheless get decisions on closing accounts right (via the rules in Line 12–13), synchronization between instances of \text{baseaccess}/3 and (transient) facts over \text{access}/3 is implemented in the \#cumulative and \#volatile parts in Listing 4. Beyond declaring inputs in the same way as before (cf. Line 10 in Listing 2), for any non-expired fact of the form \text{access}(U, \text{denied}, t + \text{offset}) given in an online progression, the integrity constraint in Line 17–18 enforces the corresponding instance of \text{baseaccess}/3, determined by calculating "\((t+\text{offset}) \mod \text{modulo}\)" to hold. Note that this constraint does not need to be decayed explicitly (yet it could be moved to the \#volatile block below) because elapsed input atoms render its instances ineffective anyway. On the other hand,
instances of the integrity constraint in Line 21–22, enforcing baseaccess/3 counterparts of non-provided facts of the form access(U,denied,\(t+\)offset) to be false, must be discharged once the sliding window progresses (by modulo many steps). For instance, when offset = 2 and modulo = 8, input atoms of the form access(U,denied,3), introduced at the first step, map to baseaccess(U,denied,3), and the same applies to instances of access(U,denied,11), becoming available at the ninth incremental step. Since the smallest time stamp that can still be mentioned by non-expired inputs at the ninth step is 5 (as facts provided up to step 6 are expired), the transition of ground integrity constraints (here mapping time stamp 11 instead of 3 to slot 3) is transparent. In addition, as unavailable inputs with time stamp 4 enforce atoms of the form baseaccess(U,denied,4) to be false (via instances of the integrity constraint in Line 21–22 provided at step 2), denial counting (in Line 9–11) cannot proceed beyond atoms baseaccess(U,denied,3), representing latest stream data at the ninth step. Hence, denial many consecutive instances of baseaccess(U,denied,T), needed to close the account of a user U, correspond to respective facts over access/3 in the current window. Finally, for avoiding initial guesses over baseaccess/3 wrt. (non-existing) denied accesses lying in the past, "#iinit 2-modulo." is included in Line 6. Then, positive values are obtained for "(T+modulo) #mod modulo" calculated in Line 21, temporarily enforcing instances of baseaccess/3 that match access/3 instances with non-positive time stamps to be false.

4.2 Overtaking Maneuver Recognition

Our second scenario deals with recognizing the completion of overtaking maneuvers by a car, eg. for signaling it to the driver. The recognition follows the transitions of the automaton in Figure 1.\(^3\) Starting from state ∅, representing that a maneuver has not yet been initiated, sensor information about being “behind,” “nextto,” or “infront” of another car enables transitions to corresponding states B, N, and F. As indicated by the output “OT!” in F, an overtaking maneuver is completed when F is reached from ∅ via a sequence of “behind,” “nextto,” and “infront” signals. Additional ε transitions model the progression from one time point to the next in the absence of signals: while such transitions are neutral in the states ∅, B, and N, the final state F is abandoned (after outputting “OT!”). For

\(^3\) Unlike in this simple example scenario, transition systems are usually described compactly in terms of state variables and operators on them, eg. defined via action languages. The automaton induced by a compact description can be of exponential size, and an explicit representation like Figure 1 is often inaccessible in practice.
instance, the automaton in Figure 1 admits the following trajectory (indicating a state at time point \( i \) by “@\( i \)” and providing signals in-between states):

\[
(∅@0, \text{behind}, B@1, ε, B@2, \text{nextto}, N@3, \text{infront}, F@4, ε,
∅@5, \text{nextto}, ∅@6, \text{behind}, B@7, \text{nextto}, N@8, ε, N@9).
\]

Here, an overtaking maneuver is completed when \( F \) is reached at time 4, and \( ε \) transitions preserve \( B \) and \( N \), but not \( F \). In the following, we consider overtaking maneuvers that are completed in at most 6 steps; that is, for a given time point \( i \), the automaton in Figure 1 is assumed to start from ∅ at time \( i−6 \).

Similar to the static access control encoding in Listing 4, our encoding of overtaking maneuver recognition, shown in Listing 5, uses modulo arithmetic to map time stamps in stream data to a corresponding slot of atoms and rules provided in the #base program part. In more detail, transient (external) facts of the form \( \text{at}(P,C,T) \) (in which \( P \) is behind, nextto, or infront and \( C \) refers to a red, blue, or green car) are matched with corresponding instances of baseat/3 by means of the #cumulative and #volatile parts in Line 20–24. As before, these parts implement a transparent shift from steps \( i \) to \( i+\text{modulo} \), provided that transient stream data is given in “#volatile : modulo.” blocks. Then, the rules in Line 12–16 model transitions based on signals, and the one in Line 17–18 implements \( ε \) transitions (making use of projection in Line 10) as specified by the automaton in Figure 1. In particular, the completion of an overtaking maneuver in the current step, indicated by deriving infront as state for a car \( C \) and a time slot \( S \) via the rule in Line 15–16, relies on now(S) (explained below). Also note that state ∅ is left implicit, i.e. it applies to a car \( C \) and a slot \( T \) if baseat \( (P,C,T) \) does not hold for any relative position \( P \), and that the frame axioms represented in Line 17–18 do not apply to infront states.

While a next/2 predicate had also been defined in Listing 4 to arrange the time slots of the #base program in a cycle, the corresponding rule in Line 7 of Listing 5 relies on the absence of now(T) for linking a time slot \( T \) to “\((T+1) \ #\text{mod} \ #\text{modulo}. \)” in fact, instances of now/1 are provided by the choice rule in Line 8 and synchronized with the incremental parameter \( t \) via the integrity constraint “:- not now(t #\text{mod} #\text{modulo}.).” of life span 1 (cf. Line 25–26). (By using integrity constraints rather than rules for continuous synchronization, we guarantee the compositionality of incremental program parts.) Unlike the previous approach to access counting (introducing an empty slot), making the current time slot explicit enables the linearization of a time slot cycle also in the presence of frame axioms, which could propagate into the “past” otherwise. In fact, if prerequisites regarding now/1 were dropped in Line 7 and 15, one could, beginning with \( \text{at}(\text{behind}, C, 7) \) as input and its corresponding atom baseat \( (\text{behind}, C, 1) \), derive state \( (\text{infront}, C, 4) \) at step 7 for a car \( C \) subject to the trajectory given above. Such a conclusion is clearly unintended, and the technique in Line 7–8 and 25–26, using now/1 to linearize a time slot
Listing 5. Static overtaking maneuver recognition encoding

```prolog
#const modulo=6. #iinit 2-modulo.
#base.
position(behind;nextto;infront). % relative positions
car(red;blue;green). % some cars
time(0..modulo-1). % time slots

next(T,(T+1) mod modulo) :- time(T), not now(T).
{ now(T) : time(T) } 1.
{ baseat(P,C,T) : position(P) : car(C) : time(T) }.
baseat(C,T) :- baseat(_,C,T).
state(behind, C,T) :- baseat(behind,C,T).
state(nextto, C,S) :- baseat(nextto,C,S), next(T,S),
{ state(behind,C,T), state(nextto,C,T) }.
state(infront,C,S) :- baseat(infront,C,S), now(S),
next(T,S), state(nextto,C,T).
state(P, C,S) :- state(P,C,T), P != infront,
next(T,S), not baseat(C,S).

#cumulative t.
#external at(P,C,t) : position(P) : car(C).
{- at(P,C,t), not baseat(P,C,t mod modulo).}
#volatile t : modulo.
{- baseat(P,C,(t+modulo) mod modulo), not at(P,C,t).}
#volatile t.
{- not now(t mod modulo).}
```

cycle, provides a general solution for this issue. Finally, “#iinit 2-modulo.” is again included (in Line 1) to avoid initial guesses over baseat/3 wrt. (non-existing) past signals.

### 4.3 Online Job Scheduling

After inspecting straightforward stream data evaluation tasks, we now turn to the problem of job scheduling, where job requests of different durations must be scheduled to machines without overlapping one another. Unlike in offline job scheduling (Brucker 2007), where all requests are known in advance, we here assume a stream of job requests, provided via (transient) facts job(I,M,D,T) in which I is a job ID, M is a machine, D is a duration, and T is the arrival time of a request. In addition, we assume a deadline T+max_step, for some integer constant max_step, by which the execution of a job I submitted at step T must be completed. For instance, an (initial) segment of a job request stream can be as follows:

```
#step 1 : 0.
job(1,1,1,1).  job(2,1,5,1).  job(3,1,5,1).  job(4,1,5,1).  job(5,1,5,1).
#step 21 : 0.
job(1,1,5,21). job(2,1,5,21).  job(3,1,5,21).  job(4,1,5,21).
```

That is, five jobs with ID 1 to 5 (of durations 1 and 5) are submitted at step 1 and ought to be completed on machine 1 within the deadline 1+max_step = 21 (taking max_step = 20). Four more jobs with ID 1 to 4 of duration 5, submitted at step 21, also need to be executed on machine 1. As a matter of fact, a schedule to finish all jobs within their deadlines must first launch the five jobs submitted at step 1, thus occupying machine 1 at time points up to 21, before the other jobs can use machine 1 from time point 22 to 41. However, when a time-decaying logic program does not admit any answer set at some step (ie. there is no schedule meeting all deadlines), the default behavior of oclingo is to increase the incremental step counter until an answer set is obtained. This behavior would lead to the expiration of pending job requests, so that a schedule generated in turn lacks part of the submitted jobs. Since such (partial) schedules are unintended here, we take advantage of
Listing 6. Static online job scheduling encoding

1. #const max_duration = 5. #const max_jobid = 5. #const num_machines = 5.
2. #const max_step = 20. #const modulo = 2*max_step+1. #init 2-modulo.
3. #base.
4. duration(1..max_duration). jobid(1..max_jobid). machine(1..num_machines).
5. next(T, (T+1) #mod modulo) :- T := 0..modulo-1.
6. \{
7.  basejob(I,M,D,T) : jobid(I) : machine(M) : duration(D) : next(T,_) \}.
8. 1 \{ jobstart(I,T, (T..T+max_step+1-D) #mod modulo) \} 1 :- basejob(I,_,D,T).
12. :- occupy(M,I1,T1,S), occupy(M,I2,T2,S), (I1,T1) < (I2,T2).
13. \#cumulative t.
14. \#external job(I,M,D,t) : jobid(I) : machine(M) : duration(D).
15. :- job(I,M,D,t), not basejob(I,M,D,t) #mod modulo).
16. \#volatile t : modulo.
17. :- basejob(I,M,D,t+modulo #mod modulo), not job(I,M,D,t).

the enriched directive “#step i : \delta.” to express that increases of the step counter must not exceed \(i+\delta\), regardless of whether an answer set has been obtained at step \(i+\delta\) (or some greater step). In fact, since \(\delta = 0\) is used above, oclingo does not increase the step counter beyond \(i\), but rather returns “unsatisfiable” as result if there is no answer set.

In Line 1–4, our (static) job scheduling encoding, shown in Listing 6, defines the viable durations, IDs of jobs requested per step, and the available machines in terms of corresponding constants. Furthermore, deadlines for the completion of jobs are obtained by adding max_step (set to 20 in Line 2) to request submission times. As a consequence, jobs with submission times \(i \leq j\) such that \(j \leq i+\text{max\_step}\) may need to be scheduled jointly, and the minimum window size modulo required to accommodate the (maximum) completion times of jointly submitted jobs is calculated accordingly in Line 2. In fact, the time slots of the #base program part are in Line 5 arranged in a cycle (similar to access counting in Listing 4). The technique applied in Line 15–19 to map job requests given by transient (external) facts over job/4 to corresponding instances of basejob/4, provided by the choice rule in Line 7, remains the same as in previous static encodings (cf. Listing 4 and 5). But note that job IDs can be shared between jobs submitted at different steps, so that pairs \((I,T)\) of an ID \(I\) and a slot \(T\) identify job requests uniquely in the sequel.

The rules in Line 8–13 of the #base program accomplish the non-overlapping scheduling of submitted jobs such that they are completed within their deadlines. In fact, the choice rule in Line 8 expresses that a job of duration \(D\) with submission time slot \(T\) must be launched such that its execution finishes at slot ”\((T+\text{max\_step}) \mod \text{modulo}\)” (at the latest). Given the slots at which jobs are started, the rules in Line 10–12 propagate the occupation of machines wrt. durations \(D\), and the integrity constraint in Line 13 makes sure that the same machine is not occupied by distinct jobs at the same time slot. For instance, the following atoms of an answer set represent starting times such that the jobs requested in the stream segment given above do not overlap and are processed within their deadlines:

\[
\text{jobstart}(1,1,1) \quad \text{jobstart}(2,1,2) \quad \text{jobstart}(3,1,7) \quad \text{jobstart}(4,1,12) \\
\text{jobstart}(5,1,17) \quad \text{jobstart}(1,21,22) \quad \text{jobstart}(2,21,27) \quad \text{jobstart}(3,21,32) \quad \text{jobstart}(4,21,37)
\]

Note that the execution of the five jobs submitted at step 1 is finished at their common deadline 21, and the same applies wrt. the deadline 41 of jobs submitted at step 21. Since machine 1 is occupied at all time slots, executing all jobs within their deadlines would no longer be feasible if a further job request, such as job(5,1,1,21), were submitted. In
such a case, oclingo will output “unsatisfiable” and wait for new online input, which may shift the window and relax the next query due to the expiration of some job requests.

A future extension of oclingo regards optimization (via #minimize/#maximize) wrt. online data, given that solutions violating as few (soft) constraints as possible may be more helpful than reporting unsatisfiability. In either mode of operation, the static representation of problems over a window of fixed size, illustrated in Listing 4, 5, and 6, enables the reuse of constraints learned upon solving a query for answering further queries asked later on.

Although oclingo is still in a prototypical state, we performed some preliminary experiments in order to give an indication of the impact of different encoding variants. As the first two example scenarios model pure data evaluation tasks (not requiring search), experiments with them did not exhibit significant runtimes, and we thus focus on results for online job scheduling. In particular, we assess oclingo on the static encoding in Listing 6 as well as a cumulative variant (analog to the cumulative access control encoding in Listing 2); we further consider the standard ASP solver clingo, processing each query independently via relaunching wrt. the current window contents. Table 2(a) provides average runtimes of the investigated configurations in seconds, grouped by satisfiable (\(\#S\)) and unsatisfiable (\(\#U\)) queries, for 12 randomly generated data streams with 200 elements each. These streams vary in the values used for constants, eg. \(\text{max_jobid} = 5\), \(\text{max_duration} = 3\), \(\text{num_machines} = 5\), and \(\text{max_step} = 15\) with the three “5x3x5\_15\_n” streams, and the respective numbers of satisfiable (\(\#S\)) and unsatisfiable (\(\#U\)) queries. First of all, we observe that the current prototype version of oclingo cannot yet compete with clingo. The reason for this is that oclingo’s underlying grounding and solving components were not designed with expiration in mind, so that they currently still remember the names of expired atoms (while irrelevant constraints referring to them are truly deleted). The resulting overhead in each step explains the advantage of relaunching clingo from scratch. When comparing oclingo’s performance wrt. encoding variants, the static encoding appears to be generally more effective than its cumulative counterpart, albeit some unsatisfiable queries stemming from the last three example streams in Table 2(a) are solved faster using the latter.

The plot in Figure 2(b) provides a more fine-grained picture by displaying runtimes for individual queries from stream “5x3x5\_15\_1,” where small bars on the x-axis indicate unsatisfiable queries. While the static encoding yields a greater setup time of oclingo at the very beginning, it afterwards dominates the cumulative encoding variant, which requires the instantiation and integration of rules representing new job requests at each step. Unlike this, the static encoding merely maps input atoms to their representations in the #base part, thus also solving each query wrt. the same (static) set of atoms. As a consequence, after initial unsatisfiable queries (yielding spikes in all configurations’ runtimes), oclingo with the static encoding is sometimes able to outperform clingo for successive queries...
remaining unsatisfiable. In fact, when the initial reasons for unsatisfiability remain in the window, follow-up queries are rather easy given the previously learned constraints, and we observed that some of these queries could actually be solved without any guessing.

5 Related Work

Academic and commercial systems for stream processing include, on the one hand, high-throughput stream processors, like Aurora (Zdonik 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. Predicates in “time-dependent answer set programs” (Fayruzov et al. 2010) may include a distinguished time argument, comparable to parameter t of (time-decaying) incremental logic program parts; the proposed solving method aims at the iterative saturation of the collection of reachable states (ie. interpretations of “time-dependent predicates”) and is not reactive. Finally, ASP was used in (Do et al. 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.

6 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. In turn, we concentrated on elaborating upon a novel modeling methodology that allows us to deal with continuously changing programs in an effective way. Our approach is implemented within the reactive ASP solver oclingo, freely available at (oclingo). Although our approach allows for harnessing modern conflict-driven learning and for avoiding redundant reprocessing, our preliminary experiments still revealed the need for re-engineering the underlying ASP systems in view of the new reasoning scenarios. The consolidation of oclingo’s implementation and the addition of yet missing features, such as reactive optimization, are subjects to ongoing and future work. Real-world applications of ASP-based stream reasoning are currently developed in the EU projects www.easyreach-project.eu and www.strokeback.eu, relying on knowledge-intense stream processing in eHealth.

Acknowledgments This work was partially funded by the German Science Foundation (DFG) under grant SCHA 550/8-1/2 and by the European Commission within the EasyReach project (www.easyreach-project.eu) under AAL call 2009-2.
References


