Answer Set Programming for Stream Reasoning[*]

M. Gebser[1], T. Grote[1], R. Kaminski[1], P. Obermeier[2], O. Sabuncu[1], and T. Schaub[1][**]

[1] Universität Potsdam, Germany
[2] DERI Galway, Ireland

Abstract. The advance of Internet and Sensor technology has brought about new challenges evoked by the emergence of continuous data streams. Beyond rapid data processing, application areas like ambient assisted living, robotics, or dynamic scheduling involve complex reasoning tasks. We address such scenarios and elaborate upon approaches to knowledge-intense stream reasoning, based on Answer Set Programming (ASP). While traditional ASP methods are devised for singular problem solving, we develop new techniques to formulate and process problems dealing with emerging as well as expiring data in a seamless way.

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 online measurements. While existing data stream management systems [4] allow for high-throughput stream processing, they lack complex reasoning capacities [5]. We address this shortcoming and elaborate upon approaches to knowledge-intense stream reasoning, based on Answer Set Programming (ASP; [6]) as a prime tool for Knowledge Representation and Reasoning (KRR; [7]). The emphasis thus shifts from rapid data processing towards complex reasoning, as required in application areas like ambient assisted living, robotics, or dynamic scheduling.

In contrast to traditional ASP methods, which are devised for singular problem solving, *"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"* [8]. To accommodate this in ASP, we develop new techniques to formulate and process problems dealing with emerging as well as expiring data in a seamless way. Our modeling approaches rely on the novel concept of time-decaying logic programs [1], where logic program parts are associated with life spans to steer their emergence as well as expiration upon continuous reasoning. Time-decaying logic programs are implemented as a recent extension of the reactive ASP system *oclingo* [9], using the ASP grounder *gringo* [10] for the recurrent composition of a static "offline" encoding with dynamic "online" data into queries to the ASP solver *clasp* [11].

While *oclingo* makes powerful ASP technology accessible for stream reasoning, its continuous query formulation and processing impose particular modeling challenges.

---

[*] This paper complements a short KR'12 paper [1]; an extended draft [2] is available at [3].
[**] Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

First, re-grounding parts of an encoding wrt. new data shall be as economical as possible. Second, traditional modeling techniques, eg. frame axioms [12], need to be reconsidered in view of the expiration of obsolete program parts. Third, the re-use of propositional atoms and rules shall be extensive to benefit from conflict-driven learning (cf. [13]). We here tackle such issues in continuous reasoning over data from a sliding window [4]. In a nutshell, we propose to encode knowledge about any potential window contents offline, so that dynamic logic program parts can concentrate on activating readily available rules (via transient online data). Moreover, we show how default conclusions, eg. expressed by frame axioms, can be faithfully combined with transient data.

After providing the necessary background, we demonstrate modeling approaches on three toy domains. The underlying principles are, however, of general applicability and thus establish universal patterns for ASP-based reasoning over sliding windows.[3]

2 Background

We presuppose familiarity with (traditional) ASP input languages (cf. [14, 15]) and (extended) logic programs (cf. [16, 6]). They provide the basis of *incremental logic programs* [17], where additional keywords, "**#base**," "**#cumulative**," and "**#volatile**," allow for partitioning rules into a static, an accumulating, and a transient program part. The latter two usually refer to some constant `t`, standing for a step number. In fact, when gradually increasing the step number, starting from 1, ground instances of rules in a **#cumulative** block are successively generated and joined with ground rules from previous steps, whereas a **#volatile** block contributes instances of its rules for the current step only. Unlike accumulating and transient program parts, which are re-processed at each incremental step, the static part indicated by **#base** is instantiated just once, initially, so that its rules correspond to a "0th" **#cumulative** block.

Application areas of incremental logic programs include planning (cf. [18]) and finite model finding (cf. [19]). For instance, (a variant of) the well-known Yale shooting problem (cf. [20]) can be modeled by an incremental logic program as follows:

```
#base.
{ loaded }.
live(0).
#cumulative t.
{ shoot(t+1) }.
  ab(t) :- shoot(t), loaded.
live(t) :- live(t-1), not ab(t).
#volatile t.
:- live(t).
:- shoot(t+1).
```

The first answer set, containing `loaded`, `live(0)`, `live(1)`, `shoot(2)`, and `ab(2)`, is generated from the following ground rules at step 2:

```
% #base.
{ loaded }.
live(0).
```

---

[3] For formal details on time-decaying logic programs and a discussion of related work in stream processing, we refer the interested reader to [1, 2].

```
% #cumulative 1.                  % #cumulative 2.
{ shoot(2) }.                     { shoot(3) }.
  ab(1) :- shoot(1), loaded.        ab(2) :- shoot(2), loaded.
live(1) :- live(0), not ab(1).   live(2) :- live(1), not ab(2).
% #volatile 2.
:- live(2).
:- shoot(3).
```

Observe that the static **#base** part is augmented with rules from the **#cumulative** block for step 1 and 2, whereas **#volatile** rules are included for step 2 only.

The *reactive* ASP system *oclingo* extends offline incremental logic programs by functionalities to incorporate external online information from a controller, also distinguishing accumulating and transient external inputs. For example, consider a character stream over alphabet $\{a, b\}$ along with the task of continuously checking whether a stream prefix at hand matches regular expression $(a|b)^*aa$. To provide the stream prefix aab, the controller component of *oclingo* can successively pass facts as follows:

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

The number i in "**#step** i." directives informs *oclingo* about the minimum step number up to which an underlying offline encoding must be instantiated in order to incorporate external information (given after a **#step** directive) in a meaningful way. In fact, the following offline encoding builds on the assumption that values i in atoms of the form read(a,i) or read(b,i) are aligned with characters' stream positions:

```
#iinit 0.
#cumulative t.
#external read(a,t+1).  #external read(b,t+1).
accept(t) :- read(a,t), read(a,t-1), not read(a;b,t+1).
```

The (undefined) atoms appearing after the keyword "**#external**" are declared as inputs to **#cumulative** blocks; that is, they are protected from program simplifications until they become defined (by external rules from the controller). Observe that any instance of the predicate $read/2$ is defined externally. In particular, future inputs that can be provided at (schematic) step t+1 are used to cancel an obsolete rule defining accept(t), once it does no longer refer to the last position of a stream prefix. Given that inputs are expected from step $1 = $ t+1 on, the directive "**#iinit** 0." specifies t $= 0$ as starting value to instantiate **#cumulative** blocks for, so that read(a,1) and read(b,1) are initially declared as external inputs. In view of this, the answer set obtained for stream prefix aa at step 2 includes accept(2) (generated via "accept(2) :- read(a,2), read(a,1), not read(a,3), not read(b,3)."), while accept(3) does not belong to the answer set for aab at step 3. Note that acceptance at different stream positions is indicated by distinct instances of accept/1; this is important to comply with modularity conditions (cf. [1, 2]) presupposed by *oclingo*, which essentially object to the (re)definition of (ground) head atoms at different steps.

Although the presented reactive ASP encoding correctly accepts stream prefixes matching $(a|b)^*aa$, the fact that external instances of $read/2$ are accumulated over time is a major handicap, incurring memory pollution upon running *oclingo* for a (quasi)

unlimited number of steps.[4] To circumvent this, external inputs could be made transient, as in the following alternative sequence of facts from *oclingo*'s controller component:

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

Note that the first two readings, represented by `read(a,1,1)` and `read(a,2,2)` as well as `read(a,1,2)` and `read(a,2,3)`, are provided by facts twice, where the respective (last) stream position is included as additional argument to avoid redefinitions. In view of this, the previous offline encoding could be replaced by the following one:

```
#cumulative t.
#external read(a,t-1;t,t).   #external read(b,t-1;t,t).
accept(t) :- read(a,t-1,t), read(a,t,t).
```

While the automatic expiration of transient inputs after each inquiry from the controller (along with the possible use of "`#forget i.`" directives) omits a blow-up in space as well as an explicit cancelation of outdated rules, it leads to the new problem that the whole window contents (two readings in this case) must be passed in each inquiry from the controller. This delegates the bookkeeping of sliding window contents to the controller, which then works around limitations of reactive ASP as introduced in [9].

Arguably, neither accumulating inputs (that are no longer inspected) over time nor replaying window contents (the width of the window many times) is acceptable in performing continuous ASP-based stream reasoning. To overcome the preexisting limitations, we introduced *time-decaying logic programs* [1] that allow for associating arbitrary *life spans* (rather than just 1) with transient program parts. Such life spans are given by integers `l` in directives of the form "`#volatile : l.`" (for online data) or "`#volatile t : l.`" (for offline encoding parts). With our example, stream readings can now be conveniently passed in `#volatile` blocks of life span 2 as follows:

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

In view of automatic expiration in two steps (eg. at step 3 for `read(a,1)` provided at step 1), the following stripped-down offline encoding correctly handles stream prefixes:

```
#cumulative t.
#external read(a,t).   #external read(b,t).
accept(t) :- read(a,t-1), read(a,t).
```

Note that the embedding of encoding rules in a `#cumulative` block builds on automatic rule simplifications relative to expired inputs. As an (equivalent) alternative, one could use "`#volatile t : 2.`" to discard outdated rules via internal assumption literals (cf. [17]). However, we next investigate more adept uses of blocks for offline rules.

3   Modeling and Reasoning

The case studies provided below aim at illustrating particular features in modeling and reasoning with time-decaying logic programs and stream data. For the sake of clarity,

---

[4] Disposal of elapsed input atoms that are yet undefined, such as `read(b,1)`, `read(b,2)`, and `read(a,3)` wrt. stream prefix *aab*, can be accomplished via "`#forget i.`" directives. Ground rules mentioning such atoms are in turn "automatically" simplified by *clasp* (cf. [17]).

Listing 1. Stream of user accesses with life span of 3 steps

```
1   #step 1. #volatile : 3.
2   access(alice,granted,1).
3   #step 2. #volatile : 3.
4   access(alice,denied,3).
5   access(bob,denied,3).
6   #step 3. #volatile : 3.
7   access(alice,denied,2).
8   access(claude,granted,5).
9   #step 4. #volatile : 3.
10  access(bob,denied,2). access(bob,denied,4).
11  access(claude,denied,2).
12  #step 5. #volatile : 3.
13  access(alice,denied,4).
14  access(claude,denied,3). access(claude,denied,4).
15  #step 6. #volatile : 3.
16  access(alice,denied,6).
17  #step 7. #volatile : 3.
18  access(alice,denied,8).
19  #step 8. #volatile : 3.
20  access(alice,denied,7).
```

we concentrate on toy domains rather than any actual target application. 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 [3].

3.1 Access Control

Our first scenario considers users attempting to access some service, for instance, by logging in via a website. Access attempts can be denied or granted, eg. depending on the supplied password, and a user account is (temporarily) closed in case of three access denials in a row. A stream segment of access attempts is shown in Listing 1. It provides data about three users, alice, bob, and claude. As specified by "**#volatile : 3.**" (for each step), the life span of access data is limited to three incremental steps (which may be correlated to some period of real time), aiming at an (automatic) reopening of closed user accounts after some waiting period has elapsed. We further assume that time stamps in the third argument of facts over access/3 deviate from 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 logic program). Given the segment in Listing 1, the following table summarizes non-expired logged accesses per step, where granted accesses are enclosed in brackets and sequences of three consecutive denials are underlined:

i	1	2	3	4	5	6	7	8
alice	[1]	[1] 3	[1] 2 3	2 3	2 4	4 6	4 6 8	6 7 8
bob		3	3	2 3 4	2 4	2 4		
claude			[5]	2 [5]	2 3 4 [5]	2 3 4	3 4	

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.

Listing 2. Cumulative access control encoding

```
1  #const window=3. #const offset=2. #const denial=3. #iinit 1-offset.

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).

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).

17 #volatile t.
18 :- account(U,closed), not denied(U,denial,t).
```

Our first offline encoding is shown in Listing 2. To keep the sliding window width, matching the life span of stream transients, adjustable, the constant `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 `offset` and `denial`. After introducing the three 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 `closed`, while it remains `open` otherwise. In fact, the dynamic parts of the incremental program solve the task to identify the current status of the account of a user `U` and represent it by including either `account(U,closed)` or `account(U,open)` in an answer set, yet without redefining `account`/2 over steps. This makes sure that step-wise (ground) incremental program parts are modularly composable (cf. [1, 2]), which is required for meaningful closed-world reasoning by *oclingo* in reactive settings.

The encoding in Listing 2 (mainly) relies on accumulating rules given below "`#cumulative t.`" (in Line 9), resembling incremental planning encodings (cf. [17]) based on a history of actions. In order to react to external inputs, the `#external` directive in Line 10 declares (undefined) atoms as inputs that can be provided by the environment, ie. the controller component of *oclingo*. Note that instances of `access`/3 with time stamp `t+offset` (where `offset` is the maximum deviation from `t`) are introduced at incremental step `t`; in this way, ground rules are prepared for 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 `denial`; if this threshold is reached (wrt. non-expired access data), the account of the respective user is temporarily closed. The "positive" conclusion from `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 window contents. 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, is included only for the (currently) last incremental step.

For the stream segment in Listing 1, in the fourth incremental step, we have that `denied(bob,3,4)` is derived in view of (transient) facts `access(bob,denied,3)`, `access(bob,denied,2)`, and `access(bob,denied,4)`. Due to the integrity 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. As one may have noticed, given the values of constants in Listing 2, the consideration of atoms over `access`/3 starts at `t+offset = 3` for `t = 1`. To still initialize the first (two) windows wrt. the "past," an `#iinit` directive is provided in Line 1. By using `1-offset = −1` as starting value for `t`, once the first external inputs are 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. Moreover, 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 anymore supposed to occur in stream data. To enable program simplifications by fixing such atoms to false, online data can be accompanied by "`#forget i.`" directives, and they are utilized in practice to dispose of elapsed input atoms (cf. [3]).

In addition to the cumulative encoding in Listing 2, we also devised a volatile variant (cf. extended draft [2]) in which outdated rules expire along with stream data. Both the cumulative and the volatile encoding have the drawback that the knowledge-intensive logic program part is (gradually) 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 like *clasp* changes significantly upon processing stream data, so that a potential re-use of learned constraints is limited. In order to reinforce conflict-driven learning, we next demonstrate a modeling approach 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 3, is similar to the cumulative approach (Listing 2), while the `#base` part is significantly extended in the sequel. In fact, the constant `modulo`, calculated in Line 6, takes into account that at most `window` many consecutive online inputs are jointly available (preceding ones are expired) and that terms representing time stamps may deviate by up to `offset` (both positively and negatively) from incremental step numbers. Given this, `window+2*offset` slots are sufficient to accommodate all distinct time stamps provided as argument values in instances of `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 the predicate `next`/2. Importantly, counting does not rely on transient external inputs, ie. `access`/3, but instead refers to instances of `baseaccess`/3, provided by the choice rule in Line 8; in

Listing 3. Static access control encoding

```
6   #const modulo=window+2*offset+1. #iinit 2-modulo.
7   next(T,(T+1) #mod modulo) :- T := 0..modulo-1.
8   { baseaccess(U,denied,T) : user(U) : next(T,_) }.
9   denied(U,1,  T) :- baseaccess(U,denied,T).
10  denied(U,N+1,S) :- baseaccess(U,denied,S), next(T,S),
11                     denied(U,N,T), N < denial.
12  account(U,closed) :- denied(U,denial,T).
13  account(U,open)   :- user(U), not account(U,closed).

15  #cumulative t.
16  #external access(U,S,t+offset) : user(U) : signal(S).
17  :- access(U,denied,T), T := t+offset,
18     not baseaccess(U,denied,T #mod modulo).

20  #volatile t : modulo.
21  :- baseaccess(U,denied,(T+modulo) #mod modulo),
22     not access(U,denied,T), T := t+offset.
```

this way, the atoms and rules in the **#base** program part can be re-used in determining the status of user accounts wrt. transient access data from a stream.

To get decisions on closing accounts right (via the rules in Line 12–13), synchronization between instances of baseaccess/3 and (transient) facts over access/3 is implemented in the **#cumulative** and **#volatile** parts in Listing 3. Beyond declaring inputs in the same way as before (cf. Line 10 in Listing 2), for any non-expired online input of the form access(U,denied,t+offset), the integrity constraint in Line 17–18 enforces the corresponding instance of baseaccess/3, calculated via "(t+offset) **#mod** modulo," to hold. Note that this constraint does not need to be expired explicitly (yet it could be moved to the **#volatile** block below) because elapsed input atoms render it ineffective anyway. However, 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 be mentioned by non-expired inputs at the ninth step is 5 (inputs given before step 7 are expired), the transition of integrity constraints (mapping time stamp 11 instead of 3 to slot 3) is transparent. In addition, as expired inputs with time stamp 4 enforce atoms of the form baseaccess(U,denied,4) to be false (via the integrity constraint in Line 21–22 instantiated for step 2), denial counting in Line 9–11 stops at 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, to avoid initial guesses over baseaccess/3 wrt. (non-existing) denied accesses lying in the past, "**#iinit** 2-modulo." is included in Line 6. Then, instances of baseaccess/3 for (positive) values "(T+modulo) **#mod** modulo," calculated in Line 21, are (temporarily) enforced to be false when they match access/3 instances with non-positive time stamps.

Fig. 1. Automaton for recognizing overtaking maneuvers
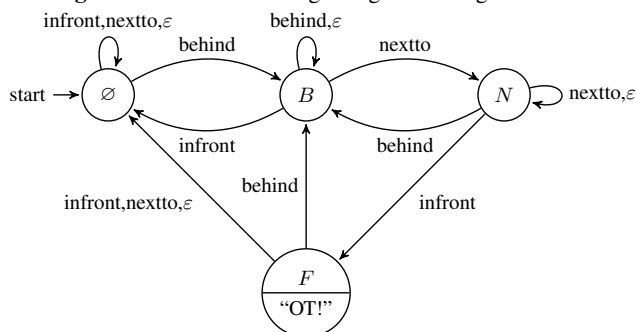
3.2 Overtaking Maneuver Recognition

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.[5] Starting from state $\varnothing$, 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 $\varnothing$ 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 states $\varnothing$, B, and N, the final state F is abandoned (after outputting "OT!"). For 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):

$$(\varnothing@0, \text{behind}, B@1, \varepsilon, B@2, \text{nextto}, N@3, \text{infront}, F@4, \varepsilon,$$
$$\varnothing@5, \text{nextto}, \varnothing@6, \text{behind}, B@7, \text{nextto}, N@8, \varepsilon, N@9).$$

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 $\varnothing$ at time $i-6$.

Similar to the static access control encoding in Listing 3, our encoding of overtaking maneuver recognition, shown in Listing 4, 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 `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 $i+$`modulo`, provided that transient stream data is given in

---

[5] 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 (cf. [21]). The automaton induced by a compact description can be of exponential size, and an explicit representation like in Figure 1 is often inaccessible in practice.

Listing 4. Static overtaking maneuver recognition encoding

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

7   next(T,(T+1) #mod modulo) :- time(T), not now(T).
8   { now(T) : time(T) } 1.
9   { baseat(P,C,T) : position(P) : car(C) : time(T) }.
10  baseat(C,T) :- baseat(_,C,T).

12  state(behind, C,T) :- baseat(behind,C,T).
13  state(nextto, C,S) :- baseat(nextto,C,S), next(T,S),
14                        1 { state(behind,C,T), state(nextto,C,T) }.
15  state(infront,C,S) :- baseat(infront,C,S), now(S),
16                        next(T,S), state(nextto,C,T).
17  state(P,       C,S) :- state(P,C,T), P != infront,
18                        next(T,S), not baseat(C,S).

20  #cumulative t.
21  #external at(P,C,t) : position(P) : car(C).
22  :- at(P,C,t), not baseat(P,C,t #mod modulo).
23  #volatile t : modulo.
24  :- baseat(P,C,(t+modulo) #mod modulo), not at(P,C,t).
25  #volatile t.
26  :- not now(t #mod modulo).
```

"**#volatile** : modulo." blocks. Then, the rules in Line 12–16 model state 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 $\varnothing$ is left implicit, ie. 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 3 to arrange the time slots of the **#base** program in a cycle, the corresponding rule in Line 7 of Listing 4 relies on the absence of now(T) for linking a time slot T to "(T+1) **#mod** modulo." In fact, instances of $now/1$ are provided by the choice rule in Line 8 and synchronized with the incremental step counter **t** via the integrity constraint ":- not now(t **#mod** modulo)." of life span 1 (cf. Line 25–26). 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 at(behind,C,7) as input and its corresponding atom baseat(behind,C,1), derive state(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 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.

3.3 Online Job Scheduling

After inspecting straightforward data evaluation tasks, we now turn to a combinatorial problem in which job requests of different durations must be scheduled to machines without overlapping one another. Unlike in offline job scheduling [22], where requests are known in advance, we here assume a stream of job requests, provided via (transient) facts `job(I,M,D,T)` such that `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.    #volatile : 21.                  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.   #volatile : 21.
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 $1+\texttt{max\_step} = 21$ (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. if 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 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 5, 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 $j \leq i+\texttt{max\_step}$ may need to be scheduled jointly, and the minimum window width `modulo` required to accommodate the (maximum) completion times of jointly submitted jobs is calculated accordingly in Line 2. Given the value of `modulo`, the time slots of the **#base** program part are in Line 5 again arranged in a cycle (similar to access counting in Listing 3). The technique applied in Line 15–19 to map job requests given by transient (external) facts over $\texttt{job}/4$ to corresponding instances of $\texttt{basejob}/4$, provided by the choice rule in Line 7, remains the same as in previous static encodings (cf. Listing 3 and 4). 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,

Listing 5. 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. #iinit 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.

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).

10  occupy(M,I,T,D,  S) :- basejob(I,M,D,T), jobstart(I,T,S).
11  occupy(M,I,T,D-1,S) :- occupy(M,I,T,D,R), next(R,S), D > 1.
12  occupy(M,I,T,S)     :- occupy(M,I,T,_,S).
13  :- occupy(M,I1,T1,S), occupy(M,I2,T2,S), (I1,T1) < (I2,T2).

15  #cumulative t.
16  #external job(I,M,D,t) : jobid(I) : machine(M) : duration(D).
17  :- job(I,M,D,t), not basejob(I,M,D,t #mod modulo).
18  #volatile t : modulo.
19  :- basejob(I,M,D,(t+modulo) #mod modulo), not job(I,M,D,t).
```

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 "$(T+max\_step)$ `#mod modulo`" (at the latest). Given the slots at which jobs are started, the rules in Line 10–12 propagate the occupation of machines M 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:

```
jobstart(1,1,1)
jobstart(2,1,2)    jobstart(1,21,22)
jobstart(3,1,7)    jobstart(2,21,27)
jobstart(4,1,12)   jobstart(3,21,32)
jobstart(5,1,17)   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 job request like `job(5,1,1,21)` were added. In such a case, *oclingo* outputs "unsatisfiable" and waits 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 just reporting unsatisfiability. In either mode of operation, the static representation of problems over a window of fixed width, illustrated in Listing 3, 4, and 5, enables the re-use 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 re-

Fig. 2. Experimental results for online job scheduling

(a) Comparison of encoding variants on data streams (b) Time plot for "5x3x5_15_1"

Stream	#S	#U	Static ∅S	Static ∅U	Cumulative ∅S	Cumulative ∅U	Relaunch ∅S	Relaunch ∅U
3x3x3_9_1	162	38	0.02	0.01	0.05	0.05	0.01	0.01
3x3x3_9_2	151	49	0.02	0.01	0.05	0.04	0.01	0.01
3x3x3_9_3	147	53	0.02	0.01	0.05	0.05	0.01	0.01
5x3x5_15_1	164	36	0.17	0.18	0.40	0.48	0.05	0.15
5x3x5_15_2	167	33	0.17	0.47	0.40	0.64	0.04	0.26
5x3x5_15_3	140	60	0.22	0.81	0.42	0.72	0.04	0.31
5x5x15_20_1	161	39	1.61	0.94	3.32	3.34	0.14	0.19
5x5x15_20_2	192	8	1.44	0.88	3.53	2.88	0.14	0.34
5x5x15_20_3	182	18	1.46	0.79	3.59	4.08	0.14	0.15
5x5x10_30_1	185	15	3.29	324.75	4.72	234.26	0.34	39.43
5x5x10_30_2	180	20	4.09	432.66	5.42	181.44	0.36	29.54
5x5x10_30_3	199	1	2.91	43.75	5.23	38.04	0.30	16.33
Total	2030	370	1.39	37.02	2.45	20.26	0.14	3.36
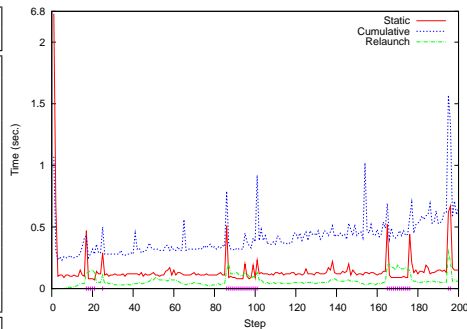
sults for online job scheduling. In particular, we assess *oclingo* on the static encoding in Listing 5 as well as a cumulative variant (analog to the cumulative access control encoding in Listing 2); we further consider the standard ASP system *clingo*, processing each query independently via relaunching 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, on 12 randomly generated data streams with 200 online inputs each. These streams vary in the values used for constants, eg. $max\_jobid = 5$, $max\_duration = 3$, $num\_machines = 5$, and $max\_step = 15$ 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 low-level 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 unrolling the horizons of 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.

4   Discussion

We have devised novel modeling approaches for continuous stream reasoning based on ASP, utilizing time-decaying logic programs to capture sliding window data in a natural way. While such data is transient and subject to routine expiration, we provided techniques to encode knowledge about potential window contents statically. This reduces the dynamic tasks of re-grounding and integrating rules of an offline encoding in view of new data to matching inputs to a corresponding internal representation. As a consequence, reasoning also concentrates on a fixed propositional representation (whose parts are selectively activated wrt. actual window contents), which enables the re-use of constraints gathered by conflict-driven learning. Although we illustrated modeling principles, including an approach to propagate frame axioms along time slot cycles, on toy domains only, the basic ideas are of general applicability. This offers interesting prospects for implementing knowledge-intense forms of stream reasoning, as required in application areas like ambient assisted living, robotics, or dynamic scheduling.

Our approach to ASP-based stream reasoning is prototypically implemented as an extension of the reactive ASP system *oclingo*, and preliminary experiments clearly show the need of improved low-level support of data expiration. In fact, we plan to combine the process of redesigning *oclingo* with the addition of yet missing functionalities, such as optimization in incremental and reactive settings. Future work also regards the consolidation of existing and the addition of further directives to steer incremental grounding and solving. For instance, beyond step-wise **#cumulative** and **#volatile** directives, we envisage **#assert** and **#retract** statements, as offered by Prolog (cf. [23]), to selectively (de)activate logic program parts. As with traditional ASP methods, the objective of future extensions is to combine high-level declarative modeling with powerful reasoning technology, automating both grounding and search. The investigation of sliding window scenarios performed here provides a first step towards gearing ASP to continuous reasoning tasks. Presupposing appropriate technology support, we think that many dynamic domains may benefit from ASP-based reasoning.

References

1. Gebser, M., Grote, T., Kaminski, R., Obermeier, P., Sabuncu, O., Schaub, T.: Stream reasoning with answer set programming: Preliminary report. In Eiter, T., McIlraith, S., eds.: Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning (KR'12), AAAI Press (2012) 613–617

2. Gebser, M., Grote, T., Kaminski, R., Obermeier, P., Sabuncu, O., Schaub, T.: Stream reasoning with answer set programming: Extended version. Unpublished (2012), available at [3]
3. oclingo: `http://www.cs.uni-potsdam.de/oclingo`
4. Golab, L., Özsu, M.: Data Stream Management. Synthesis Lectures on Data Management, Morgan and Claypool Publishers (2010)
5. Della Valle, E., Ceri, S., van Harmelen, F., Fensel, D.: It's a streaming world! reasoning upon rapidly changing information. IEEE Intelligent Systems **24**(6) (2009) 83–89
6. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
7. Lifschitz, V., van Harmelen, F., Porter, B., eds.: Handbook of Knowledge Representation. Elsevier Science (2008)
8. Barbieri, D., Braga, D., Ceri, S., Della Valle, E., Huang, Y., Tresp, V., Rettinger, A., Wermser, H.: Deductive and inductive stream reasoning for semantic social media analytics. IEEE Intelligent Systems **25**(6) (2010) 32–41
9. Gebser, M., Grote, T., Kaminski, R., Schaub, T.: Reactive answer set programming. [24] 54–66
10. Gebser, M., Kaminski, R., König, A., Schaub, T.: Advances in gringo series 3. [24] 345–351
11. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In Veloso, M., ed.: Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07), AAAI Press/MIT Press (2007) 386–392
12. Lifschitz, V.: Answer set programming and plan generation. Artificial Intelligence **138**(1-2) (2002) 39–54
13. Biere, A., Heule, M., van Maaren, H., Walsh, T., eds.: Handbook of Satisfiability. Volume 185 of Frontiers in Artificial Intelligence and Applications, IOS Press (2009)
14. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`. Available at `http://potassco.sourceforge.net`
15. Syrjänen, T.: Lparse 1.0 user's manual. Available at `http://www.tcs.hut.fi/Software/smodels`
16. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
17. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: Engineering an incremental ASP solver. In Garcia de la Banda, M., Pontelli, E., eds.: Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08). Volume 5366 of Lecture Notes in Computer Science, Springer-Verlag (2008) 190–205
18. Nau, D., Ghallab, M., Traverso, P.: Automated Planning: Theory and Practice. Morgan Kaufmann Publishers (2004)
19. Gebser, M., Sabuncu, O., Schaub, T.: An incremental answer set programming based system for finite model computation. AI Communications **24**(2) (2011) 195–212
20. Baker, A.: A simple solution to the Yale shooting problem. In Brachman, R., Levesque, H., Reiter, R., eds.: Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR'89), Morgan Kaufmann Publishers (1989) 11–20
21. Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., Turner, H.: Nonmonotonic causal theories. Artificial Intelligence **153**(1-2) (2004) 49–104
22. Brucker, P.: Scheduling Algorithms. Springer-Verlag (2007)
23. Kowalski, R.: Algorithm = logic + control. Communications of the ACM **22**(7) (1979) 424–436
24. Delgrande, J., Faber, W., eds.: Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11). Volume 6645 of Lecture Notes in Artificial Intelligence, Springer-Verlag (2011)