Answer Set Programming for Stream Reasoning

Martin Gebser   Torsten Grote   Roland Kaminski
Philipp Obermeier   Orkunt Sabuncu   Torsten Schaub

University of Potsdam

Outline

Outline

Motivation

While traditional Answer Set Programming (ASP) methods aim at 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"* [BBC+10].

Data stream management systems (for high-throughput stream processing) [GÖ10] lack complex reasoning capacities [DCvF09], as required in application areas like

- ambient assisted living,
- dynamic scheduling,
- robotics,
- etc.

Goal: Close gap by enriching ASP with means for stream reasoning**!**

Motivation

While traditional Answer Set Programming (ASP) methods aim at 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"* [BBC+10].

Data stream management systems (for high-throughput stream processing) [GÖ10] lack complex reasoning capacities [DCvF09], as required in application areas like

- ambient assisted living,
- dynamic scheduling,
- robotics,
- etc.

Goal: Close gap by enriching ASP with means for stream reasoning!

Motivation

While traditional Answer Set Programming (ASP) methods aim at 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"* [BBC+10].

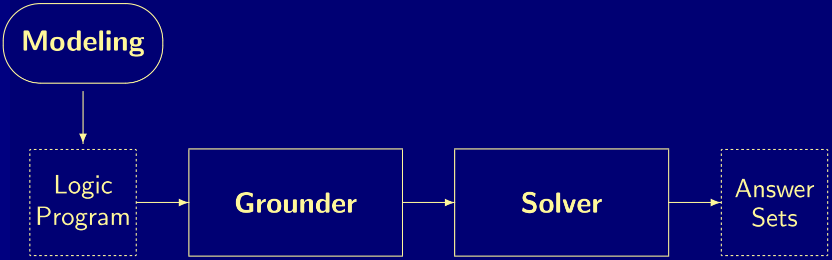
Data stream management systems (for high-throughput stream processing) [GÖ10] lack complex reasoning capacities [DCvF09], as required in application areas like

- ambient assisted living,
- dynamic scheduling,
- robotics,
- etc.

Goal: Close gap by enriching ASP with means for stream reasoning**!**

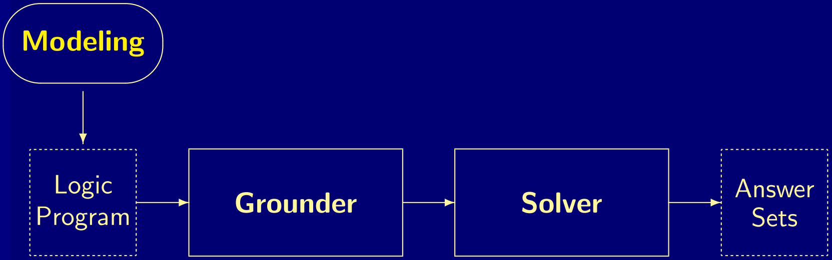
Setting the stage

Reactive ASP by `oclingo`

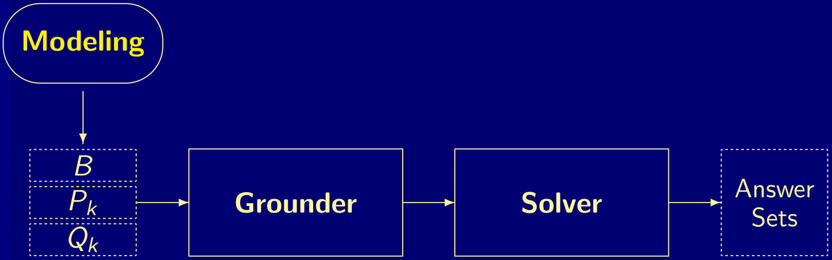


Setting the stage

Reactive ASP by `oclingo`

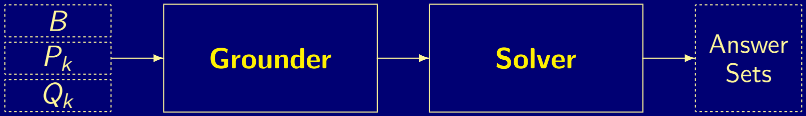


Reactive ASP by `oclingo`



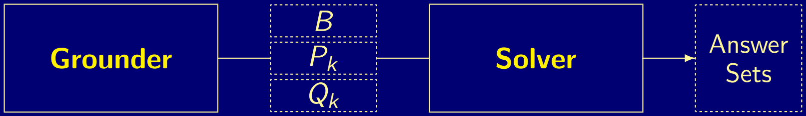
Setting the stage

Reactive ASP by `oclingo`



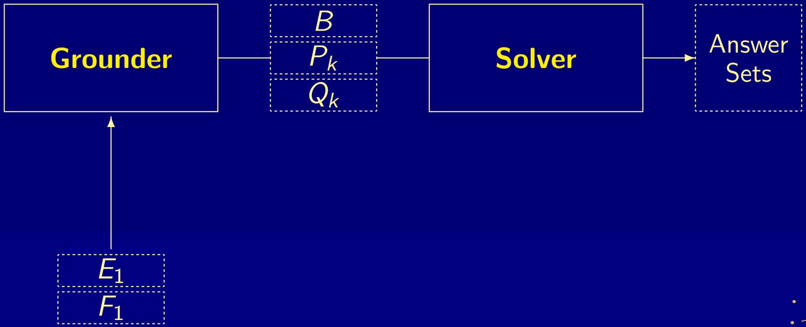
Setting the stage

Reactive ASP by `oclingo`



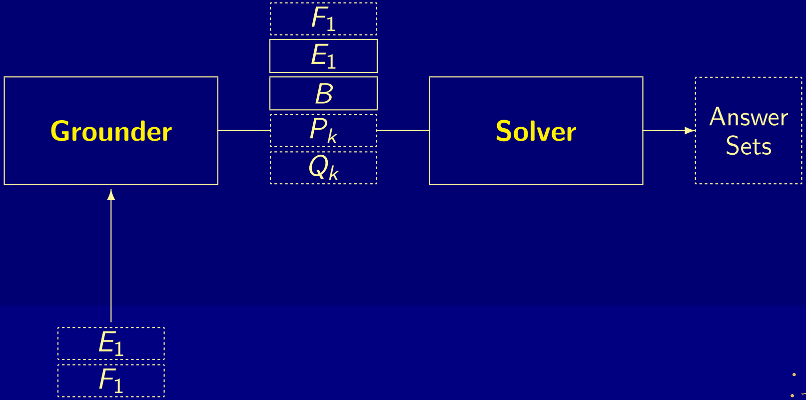
Setting the stage

Reactive ASP by oclingo



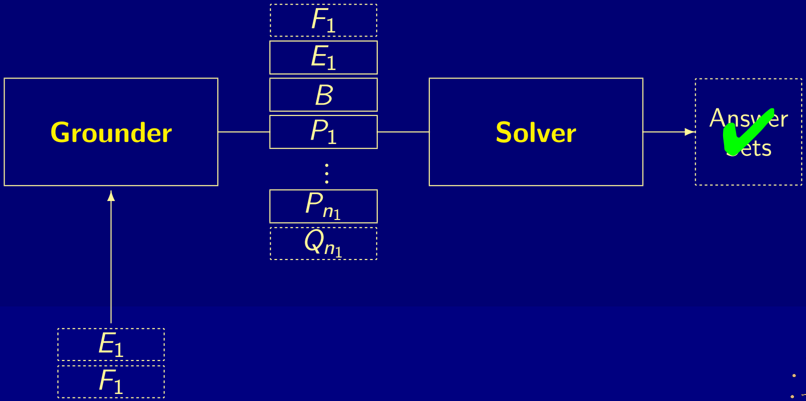
Setting the stage

Reactive ASP by oclingo



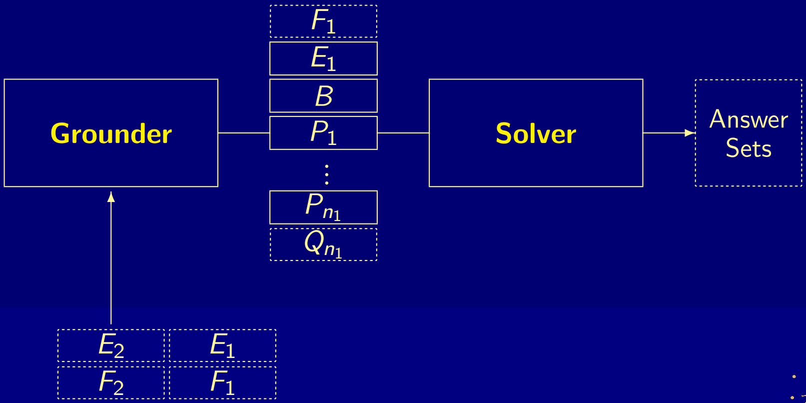
Setting the stage

Reactive ASP by `oclingo`



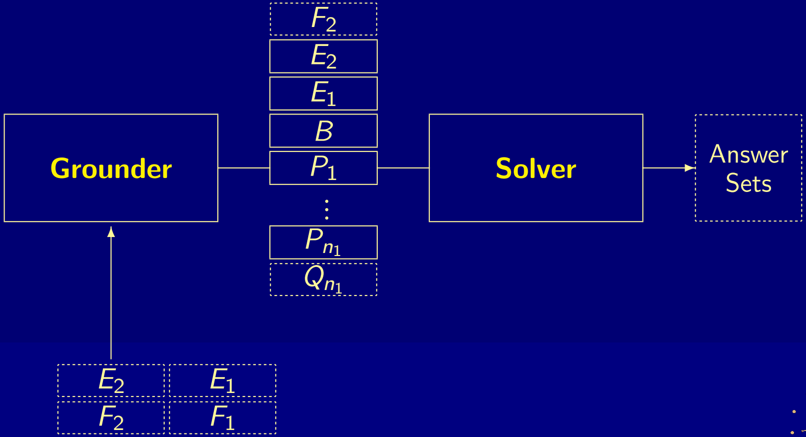
Setting the stage

Reactive ASP by `oclingo`



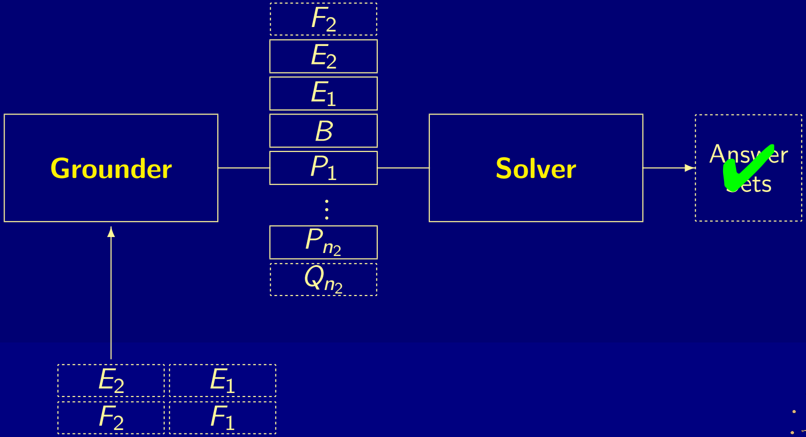
Setting the stage

Reactive ASP by oclingo

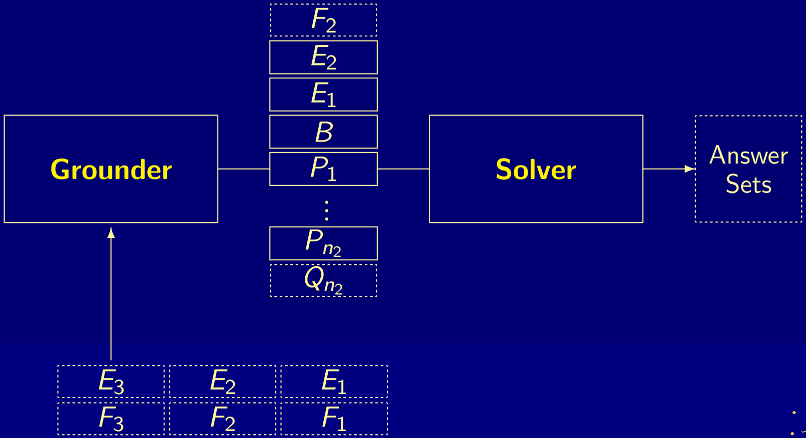


Setting the stage

Reactive ASP by oclingo

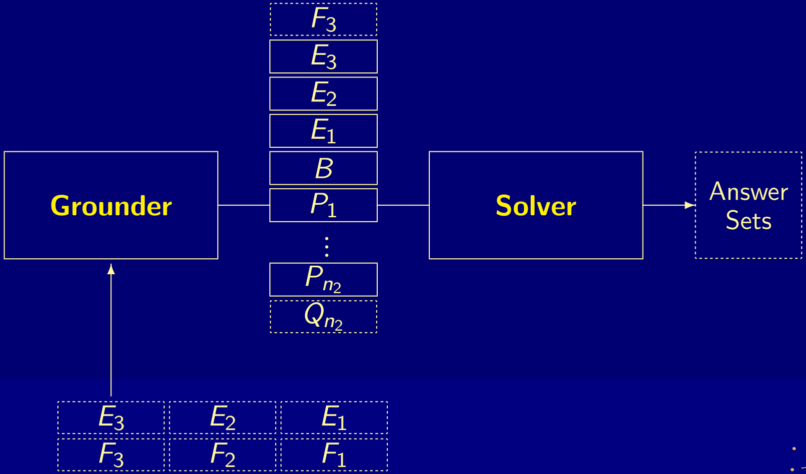


Reactive ASP by `oclingo`



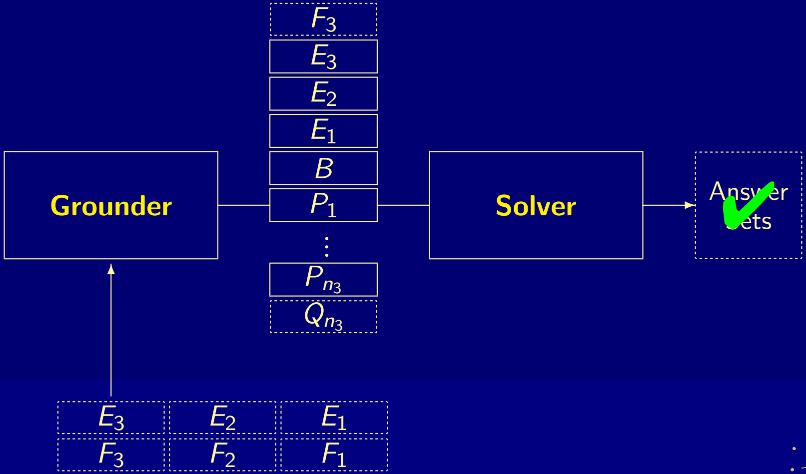
Setting the stage

Reactive ASP by oclingo



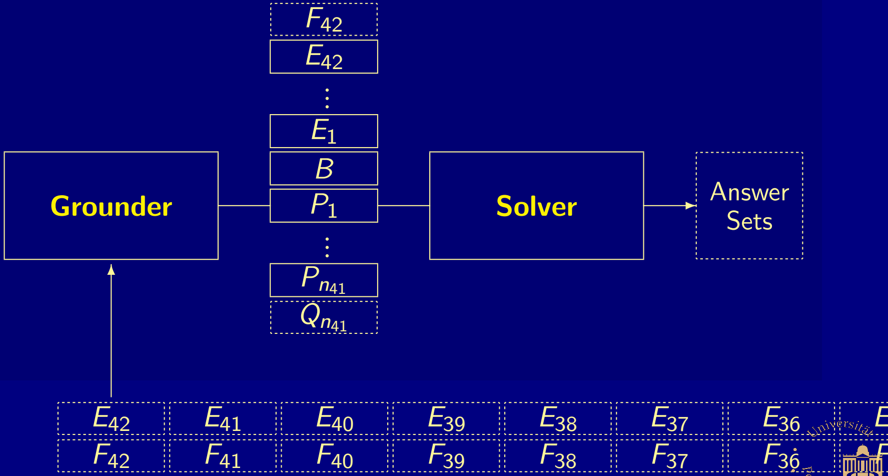
Setting the stage

Reactive ASP by `oclingo`



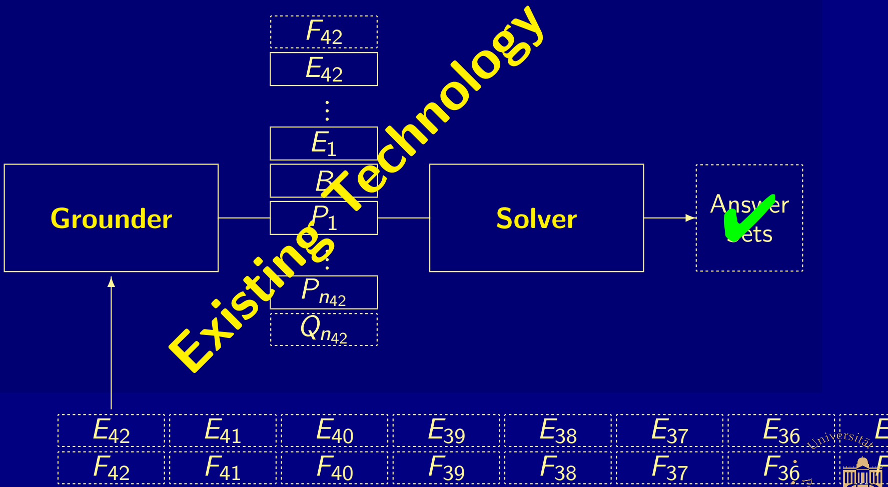
Setting the stage

Reactive ASP by oclingo



Setting the stage

Reactive ASP by `oclingo`



Outline

Running example

Consider the task of continuously matching stream prefixes against regular expression $(a|b)^*aa$.

Example Stream

$aabaaab\ldots$ ✘

Observation: Only the two last readings are significant.

➥ Restrict attention to sliding window of width 2**!**

Running example

Consider the task of continuously matching stream prefixes against regular expression $(a|b)^*aa$.

Example Stream

$aabaaab\ldots$ ✘

Observation: Only the two last readings are significant.

➥ Restrict attention to sliding window of width 2**!**

Consider the task of continuously matching stream prefixes against regular expression $(a|b)^*aa$.

Example Stream

$aabaaab\ldots$ ✔

Observation: Only the two last readings are significant.

➥ Restrict attention to sliding window of width 2**!**

Running example

Consider the task of continuously matching stream prefixes against regular expression $(a|b)^*aa$.

Example Stream

$aabaaab\ldots$ ✘

Observation: Only the two last readings are significant.

➥ Restrict attention to sliding window of width 2**!**

Running example

Consider the task of continuously matching stream prefixes against regular expression $(a|b)^*aa$.

Example Stream

$aabaaab\ldots$ ✘

Observation: Only the two last readings are significant.

➥ Restrict attention to sliding window of width 2**!**

Running example

Consider the task of continuously matching stream prefixes against regular expression $(a|b)^*aa$.

Example Stream

$aabaaab\ldots$ ✔

Observation: Only the two last readings are significant.

➥ Restrict attention to sliding window of width 2**!**

Running example

Consider the task of continuously matching stream prefixes against regular expression $(a|b)^*aa$.

Example Stream

$aabaaab\ldots$ ✔

Observation: Only the two last readings are significant.

➥ Restrict attention to sliding window of width 2!

Running example

Consider the task of continuously matching stream prefixes against regular expression $(a|b)^*aa$.

Example Stream

$aabaaab\ldots$ ✘

Observation: Only the two last readings are significant.

➥ Restrict attention to sliding window of width 2**!**

Running example

Consider the task of continuously matching stream prefixes against regular expression $(a|b)^*aa$.

Example Stream

$aabaaab\ldots$

Observation: Only the two last readings are significant.

➥ Restrict attention to sliding window of width 2!

Limitations of (foregoing) reactive ASP I

Stream Data

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

Reactive ASP Encoding

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

Incremental Instantiation: $t = 0$

```
accept(0) :- read(a,0), read(a,-1),
          not read(a,1), not read(b,1).
```

Limitations of (foregoing) reactive ASP I

Stream Data

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

Reactive ASP Encoding

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

Incremental Instantiation: $t = 0$

```
accept(0) :- read(a,0), read(a,-1),
         not read(a,1), not read(b,1).
```

Limitations of (foregoing) reactive ASP I

Stream Data

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

Reactive ASP Encoding

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

Incremental Instantiation: $t = 0$

```
accept(0) :- read(a,0), read(a,-1),
          not read(a,1), not read(b,1).
```

Limitations of (foregoing) reactive ASP I

Stream Data

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

Reactive ASP Encoding

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

Incremental Instantiation: $t = 1$

```
accept(1) :- read(a,1), read(a,0),
         not read(a,2), not read(b,2).
read(a,1).
```

Limitations of (foregoing) reactive ASP I

Stream Data

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

Reactive ASP Encoding

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

Incremental Instantiation: $t = 2$

```
accept(2) :- read(a,2), read(a,1),
             not read(a,3), not read(b,3).
read(a,1).   read(a,2).
```

Limitations of (foregoing) reactive ASP I

Stream Data

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

Reactive ASP Encoding

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

Incremental Instantiation: $t = 3$

```
accept(3) :- read(a,3), read(a,2),
             not read(a,4), not read(b,4).
read(a,1).   read(a,2).   read(b,3).
```

Limitations of (foregoing) reactive ASP I

Stream Data

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

Reactive ASP Encoding

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

Incremental Instantiation: $t = \ldots$

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

✘ Obsolete data is *not* erased!

Limitations of (foregoing) reactive ASP II

Stream Data

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

Reactive ASP Encoding

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

- ✘ Data must be replayed!
- Redefinitions (of head atoms) violate modularity assumption.

Limitations of (foregoing) reactive ASP II

Stream Data

```
#step 1.
#volatile.

read(a,1).
#endstep.
```

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

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

...

Reactive ASP Encoding

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

- ✗ Data must be replayed!
- Redefinitions (of head atoms) violate modularity assumption.

Limitations of (foregoing) reactive ASP II

Stream Data

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

Reactive ASP Encoding

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

- ✘ Data must be replayed!
- Redefinitions (of head atoms) violate modularity assumption.

Limitations of (foregoing) reactive ASP II

Stream Data

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

Reactive ASP Encoding

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

- ✘ Data must be replayed!
- Redefinitions (of head atoms) violate modularity assumption.

Limitations of (foregoing) reactive ASP II

Stream Data

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

Reactive ASP Encoding

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

- ✘ Data must be replayed!
- Redefinitions (of head atoms) violate modularity assumption.

Limitations of (foregoing) reactive ASP II

Stream Data

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

Reactive ASP Encoding

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

- ✘ Data must be replayed!
- Redefinitions (of head atoms) violate modularity assumption.

Closing the gap

Review

Neither permanent ("**#cumulative**.") nor singular ("**#volatile**.") consideration of online data is suitable for stream reasoning.

Preview

Extend `oclingo` language by "**#volatile** : l." directive for built-in support of expiration in l steps.

Closing the gap

Review

Neither permanent ("**#cumulative**.") nor singular ("**#volatile**.") consideration of online data is suitable for stream reasoning.

Preview

Extend `oclingo` language by "**#volatile** : l." directive for built-in support of expiration in l steps.

Closing the gap

Stream Data

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

Reactive ASP Encoding

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

Incremental Instantiation: $t = 1$

```
accept(1) :- read(a,1), read(a,0).
             read(a,1).
```

✔ Obsolete data is erased without necessitating replays!

Closing the gap

Stream Data

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

Reactive ASP Encoding

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

Incremental Instantiation: $t = 1$

```
accept(1) :- read(a,1), read(a,0).
             read(a,1).
```

✔ Obsolete data is erased without necessitating replays!

Closing the gap

Stream Data

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

Reactive ASP Encoding

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

Incremental Instantiation: $t = 1$

```
accept(1) :- read(a,1), read(a,0).
             read(a,1).
```

✔ Obsolete data is erased without necessitating replays!

Closing the gap

Stream Data

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

Reactive ASP Encoding

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

Incremental Instantiation: $t = 2$

```
accept(2) :- read(a,2), read(a,1).
             read(a,2). read(a,1).
```

✔ Obsolete data is erased without necessitating replays!

Closing the gap

Stream Data

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

Reactive ASP Encoding

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

Incremental Instantiation: $t = 3$

```
accept(3) :- read(a,3), read(a,2).
             read(b,3). read(a,2).
```

✔ Obsolete data is erased without necessitating replays!

Closing the gap

Stream Data

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

Reactive ASP Encoding

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

Incremental Instantiation: $t = \ldots$

```
accept(t) :- read(a,t), read(a,t-1).
             read(_,t). read(_,t-1).
```

✔ Obsolete data is erased without necessitating replays!

Outline

Recapitulation

We have seen how an reactive ASP encoding can be expanded relative to sliding window data by successively

1. generating new (ground) rules
2. defining new (ground) atoms.

✘ New propositions handicap the re-use of conflict constraints.

In what follows, we develop modeling approaches to combine online data with a static problem representation.

Idea: Encode problem wrt. any window contents and dynamically map stream data (in window) to internal representation!

Recapitulation

We have seen how an reactive ASP encoding can be expanded relative to sliding window data by successively

1. generating new (ground) rules
2. defining new (ground) atoms.

✘ New propositions handicap the re-use of conflict constraints.

In what follows, we develop modeling approaches to combine online data with a static problem representation.

Idea: Encode problem wrt. any window contents and dynamically map stream data (in window) to internal representation!

Modified running example

Consider the task of checking whether the last five readings (over alphabet $\{a, b\}$) from a stream include *aaa* as a subsequence.

Example Stream

$aabaaab\ldots$ ✘

$\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\ \downarrow$

$1234567\ldots$

Observation: Readings remain in window for five steps.

➥ Map stream positions to slots represented by remainders of 5**?**

✘ Circular subsequences may lead to false positives.

Idea: Introduce a free slot to disconnect present from past data**!**

Static problem representation captures windows of width 5.

Modified running example

Consider the task of checking whether the last five readings (over alphabet $\{a, b\}$) from a stream include *aaa* as a subsequence.

Example Stream

$aabaaab\ldots$ ✗

↧↧↧↧↧↧↧ ↧

$1234567\ldots$

Observation: Readings remain in window for five steps.

➥ Map stream positions to slots represented by remainders of 5**?**

✗ Circular subsequences may lead to false positives.

Idea: Introduce a free slot to disconnect present from past data**!**

Static problem representation captures windows of width 5.

Modified running example

Consider the task of checking whether the last five readings (over alphabet $\{a, b\}$) from a stream include *aaa* as a subsequence.

Example Stream

$aabaaab \ldots$ ✗

↧↧↧↧↧↧↧ ↧

$1234567 \ldots$

Observation: Readings remain in window for five steps.

➥ Map stream positions to slots represented by remainders of 5**?**

✗ Circular subsequences may lead to false positives.

Idea: Introduce a free slot to disconnect present from past data**!**

Static problem representation captures windows of width 5.

Modified running example

Consider the task of checking whether the last five readings (over alphabet $\{a, b\}$) from a stream include *aaa* as a subsequence.

Example Stream

$aabaaab\ldots$ ✗

$\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\ \downarrow$

$1234567\ldots$

Observation: Readings remain in window for five steps.

➥ Map stream positions to slots represented by remainders of 5**?**

✗ Circular subsequences may lead to false positives.

Idea: Introduce a free slot to disconnect present from past data**!**

Static problem representation captures windows of width 5.

Modified running example

Consider the task of checking whether the last five readings (over alphabet $\{a, b\}$) from a stream include *aaa* as a subsequence.

Example Stream

$aabaaab\ldots$ ✘

$\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow \;\; \downarrow$

$1234567\ldots$

Observation: Readings remain in window for five steps.

➥ Map stream positions to slots represented by remainders of 5**?**

✘ Circular subsequences may lead to false positives.

Idea: Introduce a free slot to disconnect present from past data**!**

Static problem representation captures windows of width 5.

Modified running example

Consider the task of checking whether the last five readings (over alphabet $\{a, b\}$) from a stream include aaa as a subsequence.

Example Stream

$aabaaab\ldots$ ✘

↧↧↧↧↧↧↧ ↧

$1234567\ldots$

Observation: Readings remain in window for five steps.

➥ Map stream positions to slots represented by remainders of 5**?**

✘ Circular subsequences may lead to false positives.

Idea: Introduce a free slot to disconnect present from past data**!**

Static problem representation captures windows of width 5.

Modified running example

Consider the task of checking whether the last five readings (over alphabet $\{a, b\}$) from a stream include aaa as a subsequence.

Example Stream

$aabaaab\ldots$ ✔

↧↧↧↧↧↧↧ ↧

$1234567\ldots$

Observation: Readings remain in window for five steps.

➥ Map stream positions to slots represented by remainders of 5**?**

✘ Circular subsequences may lead to false positives.

Idea: Introduce a free slot to disconnect present from past data**!**

Static problem representation captures windows of width 5.

Modified running example

Consider the task of checking whether the last five readings (over alphabet $\{a, b\}$) from a stream include *aaa* as a subsequence.

Example Stream

aabaaab . . . ✔

↧↧↧↧↧↧↧ ↧

1234567 . . .

Observation: Readings remain in window for five steps.

➥ Map stream positions to slots represented by remainders of 5**?**

✗ Circular subsequences may lead to false positives.

Idea: Introduce a free slot to disconnect present from past data**!**

Static problem representation captures windows of width 5.

Modified running example

Consider the task of checking whether the last five readings (over alphabet $\{a, b\}$) from a stream include aaa as a subsequence.

Example Stream

$aabaaab\ldots$ ✗

↧↧↧↧↧↧↧ ↧

$1234567\ldots$

Observation: Readings remain in window for five steps.

➥ Map stream positions to slots represented by remainders of 5?

✗ Circular subsequences may lead to false positives.

Idea: Introduce a free slot to disconnect present from past data!

Static problem representation captures windows of width 5.

Modified running example

Consider the task of checking whether the last five readings (over alphabet $\{a, b\}$) from a stream include aaa as a subsequence.

Example Stream

$aabaaab\ldots$

↧↧↧↧↧↧↧ ↧

$1234012\ldots$

Observation: Readings remain in window for five steps.

➥ Map stream positions to slots represented by remainders of 5?

✘ Circular subsequences may lead to false positives.

Idea: Introduce a free slot to disconnect present from past data!

Static problem representation captures windows of width 5.

Modified running example

Consider the task of checking whether the last five readings (over alphabet $\{a, b\}$) from a stream include aaa as a subsequence.

Example Stream

$aabaaab\ldots$ ✗

$\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\ \downarrow$

$1234012\ldots$

Observation: Readings remain in window for five steps.

➥ Map stream positions to slots represented by remainders of 5**?**

✘ Circular subsequences may lead to false positives.

Idea: Introduce a free slot to disconnect present from past data**!**

Static problem representation captures windows of width 5.

Universität Potsdam

Modified running example

Consider the task of checking whether the last five readings (over alphabet $\{a, b\}$) from a stream include aaa as a subsequence.

Example Stream

$aabaaab\ldots$ ✘

↧↧↧↧↧↧↧ ↧

$1234501\ldots$

Observation: Readings remain in window for five steps.

➥ Map stream positions to slots represented by remainders of 5?

✘ Circular subsequences may lead to false positives.

Idea: Introduce a free slot to disconnect present from past data!

✔ Static problem representation captures windows of width 5.

Modified running example

Consider the task of checking whether the last five readings (over alphabet $\{a, b\}$) from a stream include aaa as a subsequence.

Example Stream

$aabaaab\ldots$

$\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\ \downarrow$

$1234501\ldots$

Observation: Readings remain in window for five steps.

➥ Map stream positions to slots represented by remainders of 5**?**

✘ Circular subsequences may lead to false positives.

Idea: Introduce a free slot to disconnect present from past data**!**

✔ Static problem representation captures windows of width 5.

Static "free slot" approach

Reactive ASP Encoding

```
#base.
next(T,(T+1) #mod 6) :- T := 0..5.
{ b_read(a,T) } :- next(T,_).
single(T) :- b_read(a,T).
double(T) :- b_read(a,T), single(S), next(S,T).
accept    :- b_read(a,T), double(S), next(S,T).
```

- Static program part is instantiated once (initially).
- Successive slots are determined via modulo-6 arithmetic.
- Internal representation of readings is generated by choice rules.
- Subsequences *aaa* are traced wrt. internal representation.

➥ Dynamic parts must map readings to internal representation!

Static “free slot” approach

Reactive ASP Encoding

```
#base.
next(T,(T+1) #mod 6) :- T := 0..5.
{ b_read(a,T) } :- next(T,_).
single(T) :- b_read(a,T).
double(T) :- b_read(a,T), single(S), next(S,T).
accept    :- b_read(a,T), double(S), next(S,T).
```

- Static program part is instantiated once (initially).
- Successive slots are determined via modulo-6 arithmetic.
- Internal representation of readings is generated by choice rules.
- Subsequences *aaa* are traced wrt. internal representation.

➥ Dynamic parts must map readings to internal representation!

Static "free slot" approach

Reactive ASP Encoding

```
#base.
next(T,(T+1) #mod 6) :- T := 0..5.
{ b_read(a,T) } :- next(T,_).
single(T) :- b_read(a,T).
double(T) :- b_read(a,T), single(S), next(S,T).
accept    :- b_read(a,T), double(S), next(S,T).
```

Ground Instantiation

```
next(0,1).   next(3,4).
next(1,2).   next(4,5).
next(2,3).   next(5,0).
```

Static “free slot” approach

Reactive ASP Encoding

```
#base.
next(T,(T+1) #mod 6) :- T := 0..5.
{ b_read(a,T) } :- next(T,_).
single(T) :- b_read(a,T).
double(T) :- b_read(a,T), single(S), next(S,T).
accept    :- b_read(a,T), double(S), next(S,T).
```

- Static program part is instantiated once (initially).
- Successive slots are determined via modulo-6 arithmetic.
- Internal representation of readings is generated by choice rules.
- Subsequences *aaa* are traced wrt. internal representation.

➥ Dynamic parts must map readings to internal representation!

Static "free slot" approach

Reactive ASP Encoding

```
#base.
next(T,(T+1) #mod 6) :- T := 0..5.
{ b_read(a,T) } :- next(T,_).
single(T) :- b_read(a,T).
double(T) :- b_read(a,T), single(S), next(S,T).
accept    :- b_read(a,T), double(S), next(S,T).
```

- Static program part is instantiated once (initially).
- Successive slots are determined via modulo-6 arithmetic.
- Internal representation of readings is generated by choice rules.
- Subsequences *aaa* are traced wrt. internal representation.

➥ Dynamic parts must map readings to internal representation!

Static "free slot" approach

Reactive ASP Encoding

```
#base.
next(T,(T+1) #mod 6) :- T := 0..5.
{ b_read(a,T) } :- next(T,_).
single(T) :- b_read(a,T).
double(T) :- b_read(a,T), single(S), next(S,T).
accept    :- b_read(a,T), double(S), next(S,T).
```

- Static program part is instantiated once (initially).
- Successive slots are determined via modulo-6 arithmetic.
- Internal representation of readings is generated by choice rules.
- Subsequences *aaa* are traced wrt. internal representation.

➥ Dynamic parts must map readings to internal representation!

Online data vs. internal representation

Stream Data

```
#step 1.            #step 2.            #step 3.
#volatile : 5.      #volatile : 5.      #volatile : 5.
read(a,1).          read(a,2).          read(b,3).      ...
#endstep.           #endstep.           #endstep.
```

read ⇒ b_read

```
#cumulative t.            #external read(a;b,t).
:- read(a,t), not b_read(a,t #mod 6).
```

b_read ⇒ read

```
#volatile t : 6.
:- b_read(a,t #mod 6), not read(a,t).
```

➥ Constraints expire when window progresses (by six steps).

Online data vs. internal representation

Stream Data

```
#step 1.          #step 2.          #step 3.
#volatile : 5.    #volatile : 5.    #volatile : 5.
read(a,1).        read(a,2).        read(b,3).        ...
#endstep.         #endstep.         #endstep.
```

read ⇒ b_read

```
#cumulative t.          #external read(a;b,t).
:- read(a,t), not b_read(a,t #mod 6).
```

b_read ⇒ read

```
#volatile t : 6.
:- b_read(a,t #mod 6), not read(a,t).
```

➥ Constraints expire when window progresses (by six steps).

Online data vs. internal representation

Stream Data

```
#step 1.          #step 2.          #step 3.
#volatile : 5.    #volatile : 5.    #volatile : 5.
read(a,1).        read(a,2).        read(b,3).        ...
#endstep.         #endstep.         #endstep.
```

read ⇒ b_read

```
#cumulative t.            #external read(a;b,t).
:- read(a,t), not b_read(a,t #mod 6).
```

b_read ⇒ read

```
#volatile t : 6.
:- b_read(a,t #mod 6), not read(a,t).
```

➥ Constraints expire when window progresses (by six steps).

Online data vs. internal representation

Stream Data

```
#step 1.          #step 2.          #step 3.
#volatile : 5.    #volatile : 5.    #volatile : 5.
read(a,1).        read(a,2).        read(b,3).        ...
#endstep.         #endstep.         #endstep.
```

read ⇒ b_read

```
#cumulative t.          #external read(a;b,t).
:- read(a,t), not b_read(a,t #mod 6).
```

b_read ⇒ read

```
#volatile t : 6.        #iinit -4.
:- b_read(a,(t+6) #mod 6), not read(a,t).
```

➥ Window is filled at step 1 to avoid guesses (over b_read).

Online data vs. internal representation

Stream Data

```
#step 1.          #step 2.          #step 3.
#volatile : 5.    #volatile : 5.    #volatile : 5.
read(a,1).        read(a,2).        read(b,3).        ...
#endstep.         #endstep.         #endstep.
```

read ⇒ b_read

```
#cumulative t.          #external read(a;b,t).
:- read(a,t), not b_read(a,t #mod 6).
```

b_read ⇒ read

```
#volatile t : 6.        #iinit -4.
:- b_read(a,(t+6) #mod 6), not read(a,t).
```

Observation: Dynamic parts confined to data and its mapping.

Remodified running example

Consider the task of checking whether the last five readings (over an arbitrary alphabet) include at least three occurrences of letter *a*.

➥ We may use frame axioms [Lif02] in the static program part.

Reactive ASP Encoding with Frame Axioms

```
#base.
next(T,(T+1) #mod 5) :- T := 0..4.
{ b_read(a,T) } :- next(T,_).
single(T) :- b_read(a,T).
double(T) :- b_read(a,T), single(S), next(S,T).
accept    :- b_read(a,T), double(S), next(S,T).
single(T) :- single(S), next(S,T).
double(T) :- double(S), next(S,T).
```

Observation: Frame axioms propagate into the past (via `next`).

Idea: Introduce a predicate to disconnect present from the past!

Remodified running example

Consider the task of checking whether the last five readings (over an arbitrary alphabet) include at least three occurrences of letter *a*.

➥ We may use frame axioms [Lif02] in the static program part.

Reactive ASP Encoding with Frame Axioms

```
#base.
next(T,(T+1) #mod 5) :- T := 0..4.
{ b_read(a,T) } :- next(T,_).
single(T) :- b_read(a,T).
double(T) :- b_read(a,T), single(S), next(S,T).
accept    :- b_read(a,T), double(S), next(S,T).
single(T) :- single(S), next(S,T).
double(T) :- double(S), next(S,T).
```

Observation: Frame axioms propagate into the past (via `next`).

Idea: Introduce a predicate to disconnect present from the past!

Remodified running example

Consider the task of checking whether the last five readings (over an arbitrary alphabet) include at least three occurrences of letter *a*.

➥ We may use frame axioms [Lif02] in the static program part.

Reactive ASP Encoding with Frame Axioms

```
#base.
next(T,(T+1) #mod 5) :- T := 0..4.
{ b_read(a,T) } :- next(T,_).
single(T) :- b_read(a,T).
double(T) :- b_read(a,T), single(S), next(S,T).
accept    :- b_read(a,T), double(S), next(S,T).
single(T) :- single(S), next(S,T).
double(T) :- double(S), next(S,T).
```

Observation: Frame axioms propagate into the past (via `next`).

Idea: Introduce a predicate to disconnect present from the past!

Remodified running example

Consider the task of checking whether the last five readings (over an arbitrary alphabet) include at least three occurrences of letter *a*.

➥ We may use frame axioms [Lif02] in the static program part.

Reactive ASP Encoding with Frame Axioms

```
#base.
next(T,(T+1) #mod 5) :- T := 0..4.
{ b_read(a,T) } :- next(T,_).
single(T) :- b_read(a,T).
double(T) :- b_read(a,T), single(S), next(S,T).
accept    :- b_read(a,T), double(S), next(S,T).
single(T) :- single(S), next(S,T).
double(T) :- double(S), next(S,T).
```

Observation: Frame axioms propagate into the past (via `next`).

Idea: Introduce a predicate to disconnect present from the past!

Remodified running example

Consider the task of checking whether the last five readings (over an arbitrary alphabet) include at least three occurrences of letter *a*.

➥ We may use frame axioms [Lif02] in the static program part.

Reactive ASP Encoding with Frame Axioms

```
#base.
next(T,(T+1) #mod 5) :- T := 0..4.
{ b_read(a,T) } :- next(T,_).
single(T) :- b_read(a,T).
double(T) :- b_read(a,T), single(S), next(S,T).
accept    :- b_read(a,T), double(S), next(S,T).
single(T) :- single(S), next(S,T).
double(T) :- double(S), next(S,T).
```

Observation: Frame axioms propagate into the past (via `next`).

Idea: Introduce a predicate to disconnect present from the past!

Static "last slot" approach

Reactive ASP Encoding with Frame Axioms

```
#base.                    slot(0..4).
next(T,(T+1) #mod 5) :- T := 0..4.
{ b_read(a,T) } :- next(T,_).
single(T) :- b_read(a,T).
double(T) :- b_read(a,T), single(S), next(S,T).
accept    :- b_read(a,T), double(S), next(S,T).
single(T) :- single(S), next(S,T).
double(T) :- double(S), next(S,T).
{ now(T) : slot(T) } 1.
#volatile t.
:- not now(t #mod 5).
```

➥ Propagation beyond the current slot is suppressed (via `now`).

Static "last slot" approach

Reactive ASP Encoding with Frame Axioms

```
#base.                           slot(0..4).
next(T,(T+1) #mod 5) :- slot(T), not now(T).
{ b_read(a,T) } :- slot(T).
single(T) :- b_read(a,T).
double(T) :- b_read(a,T), single(S), next(S,T).
accept    :- b_read(a,T), double(S), next(S,T).
single(T) :- single(S), next(S,T).
double(T) :- double(S), next(S,T).
{ now(T) : slot(T) } 1.
#volatile t.
:- not now(t #mod 5).
```

➥ Propagation beyond the current slot is suppressed (via `now`).

Static "last slot" approach

Reactive ASP Encoding with Frame Axioms

```
#base.                         slot(0..4).
next(T,(T+1) #mod 5) :- slot(T), not now(T).
{ b_read(a,T) } :- slot(T).
single(T) :- b_read(a,T).
double(T) :- b_read(a,T), single(S), next(S,T).
accept    :- b_read(a,T), double(S), next(S,T).
single(T) :- single(S), next(S,T).
double(T) :- double(S), next(S,T).
{ now(T) : slot(T) } 1.
#volatile t.
:- not now(t #mod 5).
```

➥ Propagation beyond the current slot is suppressed (via `now`).

Outline

Summary

We have

1. extended `oclingo` by built-in support of sliding windows and
2. developed modeling approaches to reason over transient data.

To promote the re-use of conflict constraints, we proposed to

1. statically encode a task wrt. any window contents and
2. dynamically map stream data to its designated representation.

We demonstrated how to preserve the chronology of data via

1. a free slot in the static program part or
2. a predicate qualifying the current slot.

↪ Beyond simple illustration domains [GGK+12a, GGK+12b], the presented modeling approaches are of general applicability, especially to solve combinatorial problems wrt. stream data.

Summary

We have

1. extended `oclingo` by built-in support of sliding windows and
2. developed modeling approaches to reason over transient data.

To promote the re-use of conflict constraints, we proposed to

1. statically encode a task wrt. any window contents and
2. dynamically map stream data to its designated representation.

We demonstrated how to preserve the chronology of data via

1. a free slot in the static program part or
2. a predicate qualifying the current slot.

➥ Beyond simple illustration domains [GGK+12a, GGK+12b], the presented modeling approaches are of general applicability, especially to solve combinatorial problems wrt. stream data.

Outlook

While ASP offers interesting prospects for knowledge-intense stream reasoning, continuous settings impose particular challenges.

- Improved low-level support of data expiration is needed to avoid memory pollution.
- Yet missing sequential functionalities, such as optimization, must be supplied to incremental and reactive operation modes.
- Provision of handy high-level language constructs is desirable to facilitate the modeling of sliding window scenarios by users.
- Additional control directives, such as **#assert** and **#retract**, may be useful to flexibly (de)activate logic program parts.

This work is only a first step towards ASP-based stream reasoning.

➥ Realistic applications must be pioneered to furnish a deeper understanding and advanced system support of use cases.

Outlook

While ASP offers interesting prospects for knowledge-intense stream reasoning, continuous settings impose particular challenges.

- Improved low-level support of data expiration is needed to avoid memory pollution.
- Yet missing sequential functionalities, such as optimization, must be supplied to incremental and reactive operation modes.
- Provision of handy high-level language constructs is desirable to facilitate the modeling of sliding window scenarios by users.
- Additional control directives, such as **#assert** and **#retract**, may be useful to flexibly (de)activate logic program parts.

This work is only a first step towards ASP-based stream reasoning.

➥ Realistic applications must be pioneered to furnish a deeper understanding and advanced system support of use cases.

Outlook

While ASP offers interesting prospects for knowledge-intense stream reasoning, continuous settings impose particular challenges.

- Improved low-level support of data expiration is needed to avoid memory pollution.
- Yet missing sequential functionalities, such as optimization, must be supplied to incremental and reactive operation modes.
- Provision of handy high-level language constructs is desirable to facilitate the modeling of sliding window scenarios by users.
- Additional control directives, such as **#assert** and **#retract**, may be useful to flexibly (de)activate logic program parts.

This work is only a first step towards ASP-based stream reasoning.

➥ Realistic applications must be pioneered to furnish a deeper understanding and advanced system support of use cases.

References I

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

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

M. Gebser, T. Grote, R. Kaminski, P. Obermeier, O. Sabuncu, and T. Schaub.
Answer set programming for stream reasoning.
In M. Fink and Y. Lierler, editors, *Proceedings of the Fifth Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'12)*, 2012.

M. Gebser, T. Grote, R. Kaminski, P. Obermeier, O. Sabuncu, and T. Schaub.
Stream reasoning with answer set programming: Extended version.
Unpublished draft, 2012.
Available at `http://www.cs.uni-potsdam.de/oclingo`.

References II

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

M. Gebser, T. Grote, R. Kaminski, and T. Schaub.
Reactive answer set programming.
In J. Delgrande and W. Faber, editors, *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, volume 6645 of *Lecture Notes in Artificial Intelligence*, pages 54–66. Springer-Verlag, 2011.

M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele.
Engineering an incremental ASP solver.
In M. Garcia de la Banda and E. Pontelli, editors, *Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08)*, volume 5366 of *Lecture Notes in Computer Science*, pages 190–205. Springer-Verlag, 2008.

References III

- M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider.
  Potassco: The Potsdam answer set solving collection.
  AI Communications, 24(2):105–124, 2011.

- L. Golab and M. Özsu.
  Data Stream Management.
  Synthesis Lectures on Data Management. Morgan and Claypool Publishers, 2010.

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