# Stream Reasoning-Related Activities at KRR Uni Potsdam

Martin Gebser Philipp Obermeier Orkunt Sabuncu
Roland Kaminski Torsten Schaub

# Overview

# Outline

# Approach Overview

## Two Main Aspects

- Extension of (Reactive) ASP to allow for sliding-window-based reasoning: built-in support of data expiration
- Encoding technique to re-use learned conflict constraints by
  1. statically encode a task wrt. any window contents and
  2. dynamically map stream data to static encoding

## References


M. Gebser, T. Grote, R. Kaminski, P. Obermeier, O. Sabuncu, T. Schaub.
Stream reasoning with answer set programming: Preliminary report.
*KR'12*, 2012.

M. Gebser, T. Grote, R. Kaminski, P. Obermeier, O. Sabuncu, T. Schaub.
Answer set programming for stream reasoning.
*ASPOCP'12*, 2012.

# 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 length 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 length 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 length 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 length 2**!**

# Running Example

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

## Example Stream

$aababaab\ldots$ ✘

**Observation:** Only the two last readings are significant.

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

# Sliding the Window

## Stream Data (Expiration after 2 Steps)

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

### Reactive ASP Encoding

```
#program cumulative t.
#external read((a;b),t).          % set False after 2 steps
accept(t) :- read(a,(t;t-1)).
```

### Incremental Instantiation: $t = 1$

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

✔ Obsolete stream data is erased after expiration!

# Sliding the Window

## Stream Data (Expiration after 2 Steps)

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

## Reactive ASP Encoding

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

**% set False after 2 steps**

## Incremental Instantiation: $t = 1$

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

✔ Obsolete stream data is erased after expiration!

# Sliding the Window

## Stream Data (Expiration after 2 Steps)

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

## Reactive ASP Encoding

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

**% set False after 2 steps**

## Incremental Instantiation: $t = 1$

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

✔ Obsolete stream data is erased after expiration!

# Sliding the Window

## Stream Data (Expiration after 2 Steps)

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

## Reactive ASP Encoding

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

**% set False after 2 steps**

## Incremental Instantiation: $t = 2$

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

✔ Obsolete stream data is erased after expiration!

# Sliding the Window

## Stream Data (Expiration after 2 Steps)

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

## Reactive ASP Encoding

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

**% set False after 2 steps**

## Incremental Instantiation: $t = 3$

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

✔ Obsolete stream data is erased after expiration!

# Sliding the Window

## Stream Data (Expiration after 2 Steps)

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

## Reactive ASP Encoding

```
#program cumulative t.
#external read((a;b),t).          % set False after 2 steps
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 stream data is erased after expiration!

# 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

$aabaaab\ldots$ ✗

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

$1234567\ldots$

**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.

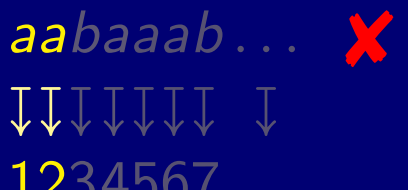
**Idea:** Introduce a free slot to disconnect present from past data**!**

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




**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 ...

**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 ...

**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


**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.

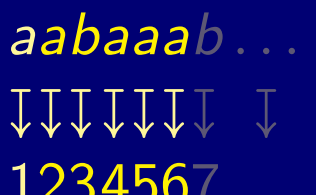
**Idea:** Introduce a free slot to disconnect present from past data**!**

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




**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 ...

**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 . . .

**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

$aabaaab\ldots$ ✘

↧↧↧↧↧↧↧ ↧

$1234501\ldots$

**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

$aabaaab\ldots$ ✘

↧↧↧↧↧↧↧ ↧

$1234501\ldots$

**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

```
next(T,(T+1) \ 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

```
next(T,(T+1) \ 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

```
next(T,(T+1) \ 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

```
next(T,(T+1) \ 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

```
next(T,(T+1) \ 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

```
next(T,(T+1) \ 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 (Expiration after 5 steps)

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

read ⇒ b_read

```
#program cumulative(t).
#external read((a;b),t).                % set False after 5 steps
:- read(a,t), not b_read(a,t \ 6).
```

b_read ⇒ read

```
#program volatile(t).
#external volatile(t).                  % set True for steps t to t+5
:- b_read(a,t \ 6), not read(a,t), volatile(t).
```

➥ Constraints expire when window progresses (by six steps)

# Online Data vs. Internal Representation

## Stream Data (Expiration after 5 steps)

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

## read ⇒ b_read

```
#program cumulative(t).
#external read((a;b),t).                % set False after 5 steps
:- read(a,t), not b_read(a,t \ 6).
```

## b_read ⇒ read

```
#program volatile(t).
#external volatile(t).                  % set True for steps t to t+5
:- b_read(a,t \ 6), not read(a,t), volatile(t).
```

➥ Constraints expire when window progresses (by six steps)

# Online Data vs. Internal Representation

## Stream Data (Expiration after 5 steps)

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

## read ⇒ b_read

```
#program cumulative(t).
#external read((a;b),t).                % set False after 5 steps
:- read(a,t), not b_read(a,t \ 6).
```

## b_read ⇒ read

```
#program volatile(t).
#external volatile(t).                  % set True for steps t to t+5
:- b_read(a,t \ 6), not read(a,t), volatile(t).
```

➥ Constraints expire when window progresses (by six steps)

# Online Data vs. Internal Representation

## Stream Data (Expiration after 5 steps)

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

## read ⇒ b_read

```
#program cumulative(t).
#external read((a;b),t).                % set False after 5 steps
:- read(a,t), not b_read(a,t \ 6).
```

## b_read ⇒ read

```
#program volatile(t).
#external volatile(t).                  % set True for steps t to t+5
:- b_read(a,t \ 6), not read(a,t), volatile(t).
```

➥ Constraints expire when window progresses (by six steps)

# Online Data vs. Internal Representation

## Stream Data (Expiration after 5 steps)

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

## read ⇒ b_read

```
#program cumulative(t).
#external read((a;b),t).                % set False after 5 steps
:- read(a,t), not b_read(a,t \ 6).
```

## b_read ⇒ read

```
#program volatile(t).
#external volatile(t).                  % set True for steps t to t+5
:- b_read(a,t \ 6), not read(a,t), volatile(t).
```

**Observation:** Dynamic parts confined to data and its mapping.

# Outline

# Motivation

- Claim ASP is an under-the-hood technology

  That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- Single-shot solving: *ground* | *solve*

  Multi-shot solving: *ground* | *solve*

  ➥ *continuously changing logic programs*

Agents, Assisted Living, Robotics, Planning, Query-answering, etc

*clingo* 4

# Motivation

- Claim ASP is an under-the-hood technology

  That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- Single-shot solving: *ground* | *solve*

  Multi-shot solving: *ground* | *solve*

  ➥ *continuously changing logic programs*

Agents, Assisted Living, Robotics, Planning, Query-answering, etc

*clingo* 4

# Motivation

- Claim ASP is an under-the-hood technology

  That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- Single-shot solving: *ground* | *solve*
- Multi-shot solving: *ground* | *solve*

  ➥ *continuously changing logic programs*

- Application areas

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc

- Implementation *clingo* 4

# Motivation

- Claim ASP is an under-the-hood technology

  That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- Single-shot solving: *ground* | *solve*
- Multi-shot solving: *ground* | *solve*
  - ➥ *continuously changing logic programs*
- Application areas

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc
- Implementation *clingo* 4

# Motivation

- Claim ASP is an under-the-hood technology

  That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- Single-shot solving: *ground* | *solve*
- Multi-shot solving: $ground^*$ | $solve^*$
  - ➥ *continuously changing logic programs*
- Application areas

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc
- Implementation *clingo* 4

# Motivation

- Claim ASP is an under-the-hood technology

  That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- Single-shot solving: $\mathit{ground} \mid \mathit{solve}$
- Multi-shot solving: $(\mathit{ground}^* \mid \mathit{solve}^*)^*$
  - ➥ *continuously changing logic programs*
- Application areas

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc
- Implementation *clingo* 4

# Motivation

- Claim ASP is an under-the-hood technology

  That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- Single-shot solving: *ground* | *solve*
- Multi-shot solving: ( *input* | *ground*$^*$ | *solve*$^*$ )$^*$
  - ➥ *continuously changing logic programs*
- Application areas

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc
- Implementation *clingo* 4

# Motivation

- Claim ASP is an under-the-hood technology

  That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- Single-shot solving: *ground* | *solve*
- Multi-shot solving: ( *input* | *ground*$^*$ | *solve*$^*$| *theory* )$^*$
  - ➥ *continuously changing logic programs*
- Application areas

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc
- Implementation *clingo* 4

# Motivation

- Claim ASP is an under-the-hood technology

  That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- Single-shot solving: *ground* | *solve*
- Multi-shot solving: ( *input* | *ground*$^*$ | *solve*$^*$| *theory* | . . . )$^*$
  - ➥ *continuously changing logic programs*
- Application areas

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc
- Implementation *clingo* 4

# Motivation

- Claim ASP is an under-the-hood technology

  That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- Single-shot solving: *ground* | *solve*
- Multi-shot solving: ( *input* | *ground*$^*$ | *solve*$^*$| *theory* | . . . )$^*$
  - ➥ *continuously changing logic programs*
- Application areas

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc
- Implementation *clingo* 4

# Motivation

- Claim ASP is an under-the-hood technology

  That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- Single-shot solving: *ground* | *solve*
- Multi-shot solving: ( *input* | *ground*$^*$ | *solve*$^*$| *theory* | . . . )$^*$
  - ➥ *continuously changing logic programs*
- Application areas

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc

- Implementation *clingo 4*

# Motivation

- Claim ASP is an under-the-hood technology

  That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- Single-shot solving: *ground* | *solve*
- Multi-shot solving: ( *input* | *ground*$^*$ | *solve*$^*$| *theory* | . . . )$^*$
  - ➥ *continuously changing logic programs*
- Application areas

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc
- Implementation *clingo* 4

# Clingo = ASP + Control

- ASP

```
#program <name> [ (<parameters>) ]
          #program play(t).
#external <atom> [ :  <body> ]
          #external mark(X,Y,P,t) :  field(X,Y), player(P).
```

- Control

Lua (www.lua.org)

```
          prg:solve(), prg:ground(parts), ...
```

Python (www.python.org)

```
          prg.solve(), prg.ground(parts), ...
```

- Integration

in ASP: embedded scripting language (#script)
in Lua/Python: library import (import gringo)

# Clingo = ASP + Control

- ASP
  - `#program <name> [ (<parameters>) ]`
    - Example `#program play(t).`
  - `#external <atom> [ : <body> ]`
    - Example `#external mark(X,Y,P,t) : field(X,Y), player(P).`
- Control
  - Lua (`www.lua.org`)
    `prg:solve(), prg:ground(parts), ...`
  - Python (`www.python.org`)
    `prg.solve(), prg.ground(parts), ...`
- Integration
  - in ASP: embedded scripting language (`#script`)
  - in Lua/Python: library import (`import gringo`)

# Clingo = ASP + Control

- ASP
  - `#program <name> [ (<parameters>) ]`
    - Example `#program play(t).`
  - `#external <atom> [ : <body> ]`
    - Example `#external mark(X,Y,P,t) : field(X,Y), player(P).`
- Control

  Lua (`www.lua.org`)

  `prg:solve(), prg:ground(parts), ...`

  Python (`www.python.org`)

  `prg.solve(), prg.ground(parts), ...`
- Integration

  in ASP: embedded scripting language (`#script`)

  in Lua/Python: library import (`import gringo`)

# Clingo = ASP + Control

- ASP
  - `#program <name> [ (<parameters>) ]`
    - Example `#program play(t).`
  - `#external <atom> [ : <body> ]`
    - Example `#external mark(X,Y,P,t) : field(X,Y), player(P).`
- Control
  - Lua (`www.lua.org`)
    - Example `prg:solve(), prg:ground(parts), ...`
  - Python (`www.python.org`)
    - Example `prg.solve(), prg.ground(parts), ...`
- Integration

  in ASP: embedded scripting language (`#script`)
  in Lua/Python: library import (`import gringo`)

# Clingo = ASP + Control

- ASP
  - `#program <name> [ (<parameters>) ]`
    - Example `#program play(t).`
  - `#external <atom> [ : <body> ]`
    - Example `#external mark(X,Y,P,t) : field(X,Y), player(P).`
- Control
  - Lua (`www.lua.org`)
    - Example `prg:solve()`, `prg:ground(parts)`, ...
  - Python (`www.python.org`)
    - Example `prg.solve()`, `prg.ground(parts)`, ...
- Integration

  in ASP: embedded scripting language (`#script`)
  in Lua/Python: library import (`import gringo`)

# Clingo = ASP + Control

- ASP
  - `#program <name> [ (<parameters>) ]`
    - Example `#program play(t).`
  - `#external <atom> [ : <body> ]`
    - Example `#external mark(X,Y,P,t) : field(X,Y), player(P).`
- Control
  - Lua (`www.lua.org`)
    - Example `prg:solve(), prg:ground(parts), ...`
  - Python (`www.python.org`)
    - Example `prg.solve(), prg.ground(parts), ...`
- Integration
  - in ASP: embedded scripting language (`#script`)
  - in Lua/Python: library import (`import gringo`)

# Vanilla *clingo*

- Emulating *clingo* in *clingo* 4

```
#script (python)
def main(prg):
    parts = []
    parts.append(("base", []))
    prg.ground(parts)
    prg.solve()
#end.
```

# Vanilla *clingo*

- Emulating *clingo* in *clingo* 4

```
#script (python)
def main(prg):
    parts = []
    parts.append(("base", []))
    prg.ground(parts)
    prg.solve()
#end.
```

# Vanilla *clingo*

- Emulating *clingo* in *clingo* 4

```
#script (python)
def main(prg):
    parts = []
    parts.append(("base", []))
    prg.ground(parts)
    prg.solve()
#end.
```

# Outline

# ROSoClingo

- ROSoClingo provides a highly capable reasoning framework for ROS by integrating the reactive answer set solver clingo
- Representation methodology based on reactive ASP
  - clingo can react to incoming requests, environment changes, and new sensory information
  - Exogenous events are modelled by clingo's external directives
  - Execution failures are directly incorporated in the encoding
- Single framework declaratively controlling robots to do complex action planning while adapting to new information and environment changes
- Available at `potassco.sourceforge.net`

# Outline

# Summary and Outlook

## Stream Reasoning Approach for ASP

- Extended (Reactive) ASP by built-in support of sliding windows
- Developed modeling approaches to reason over transient data with re-use of conflict constraint

## Clingo = Control + ASP

- Operative framework to continuously process ASP programs
- Interleaving of ASP grounding/solving with imperative control, among others, essential for stream reasoning

# Summary and Outlook

Applications (Hybris project) driving our future refinements and extensions


Warehouse logistics

Robocup logistics