

Stream Runtime Verification

César Sánchez

IMDEA Software Institute, Spain

Felipe Gorostiaga Laura Bozzelli

Bernd Finkbeiner, Peter Faymonville, Hazem Torfah (et al.)

Martin Leucker, Daniel Thoma, Torben Sheffel, Malte Schmitz and A. Schramm (et al.)

Ben D'Angelo, Henny B. Sipma, Sriram Sankaranarayanan, Zohar Manna

Introduction

Introduction

Main goal of Stream Runtime Verification:

To express rich monitors easily

Introduction



Introduction



Introduction



Introduction



Introduction



Introduction



Introduction



Introduction



Introduction

Main goal of Stream Runtime Verification:

To express rich monitors easily

Introduction

Main goal of **Stream Runtime Verification**:

*To express **rich** monitors **easily***

- ▶ **for** **outline** runtime verification
 - ▶ **both** **online** and offline
 - ▶ **non intrusively**
-
- ▶ **Expressive**: extend monitoring to computing richer outcomes (beyond YES/NO)
 - ▶ **User friendly**: engineers use (and prefer) the language

Temporal Logics (and calculi, regular expressions, etc) tend to be cumbersome in practice for engineers

Motivation (user-friendly)

Example: “ $\Box_p F$ (F holds with probability at least p) ”

Motivation (user-friendly)

Example: “ $\Box_p F$ (F holds with probability at least p)”

In Eagle:

$$\begin{aligned} \min A(\text{Form } F, \text{float } p, \text{int } f, \text{int } t) = \\ (\bigcirc \text{Empty}() \wedge ((F \wedge (1 - \frac{f}{t}) \geq p) \vee (\neg F \wedge (1 - \frac{f+1}{t} \geq p))) \\ (\bigcirc \neg \text{Empty}() \wedge ((F \rightarrow \bigcirc A(F, p, f, t+1)) \\ \wedge (\neg F \rightarrow \bigcirc A(F, p, f+1, t+1)))) \end{aligned}$$

Motivation (user-friendly)

Example: “ $\Box_p F$ (F holds with probability at least p) ”

In Eagle:

$$\begin{aligned} \min A(\text{Form } F, \text{float } p, \text{int } f, \text{int } t) = \\ (\bigcirc \text{Empty}() \wedge ((F \wedge (1 - \frac{f}{t}) \geq p) \vee (\neg F \wedge (1 - \frac{f+1}{t} \geq p))) \\ (\bigcirc \neg \text{Empty}() \wedge ((F \rightarrow \bigcirc A(F, p, f, t+1)) \\ \wedge (\neg F \rightarrow \bigcirc A(F, p, f+1, t+1)))) \end{aligned}$$

In stream runtime verification:

```
output int    total    := total[1,0] + 1
output int    countF   := countF[1,0] + (if F then 1 else 0)
output bool   BoxFp    :=  $\frac{\text{countF}}{\text{total}} \geq p$ 
```

History of Stream Runtime Verification

B. D'Angelo, S. Sankaranarayanan, César Sánchez, W. Robinson, B. Finkbeiner, H. Sipma, S. Mehrotra, Z. Manna: *LOLA: Runtime Monitoring of Synchronous Systems*. TIME 2005

A. Pnueli, A. Zaks: *PSL Model Checking and Run-Time Verification Via Testers*. FM 2006

L. Pike, A. Goodloe, R. Morisset, S. Niller: *Copilot: A Hard Real-Time Runtime Monitor*. RV 2010

T. Reinbacher, K. Rozier, J. Schumann: *Temporal-Logic Based Runtime Observer Pairs for System Health Management of Real-Time Systems*. TACAS 2014

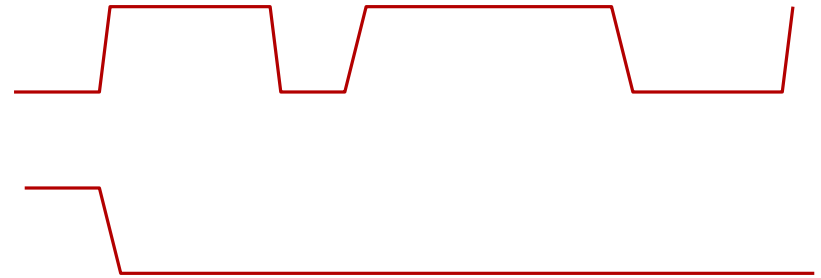
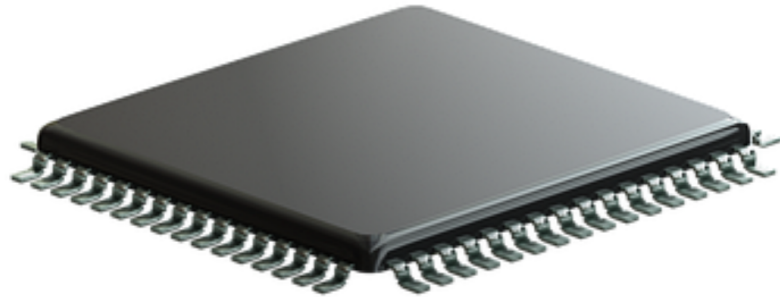
P. Faymonville, B. Finkbeiner, S. Schirmer, H. Torfah: *A Stream-Based Specification Language for Network Monitoring*. RV 2016

F. Adolf, P. Faymonville, B. Finkbeiner, S. Schirmer, C. Torens: *Stream Runtime Monitoring on UAS*. RV 2017

M. Leucker, C. Sánchez, T. Scheffel, M. Schmitz, A. Schramm: *TeSSLa: Runtime Verification of Non-synchronized Real-Time Streams*. SAC 2018

History of Stream Runtime Verification

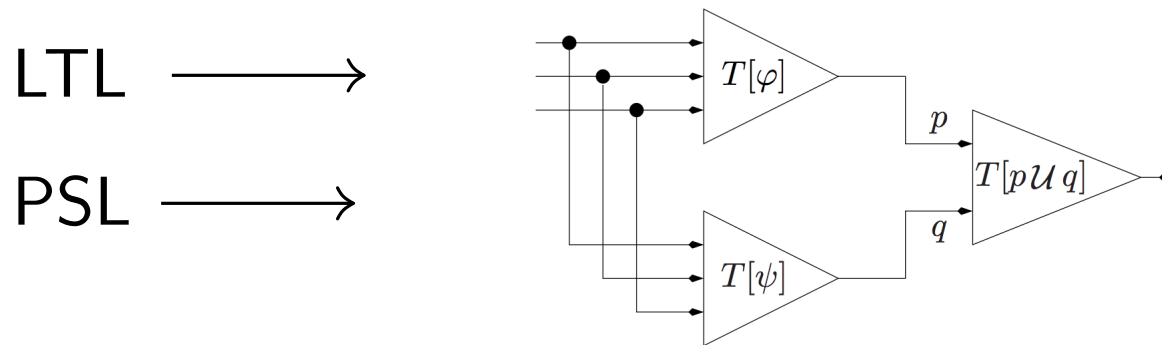
B. D'Angelo, S. Sankaranarayanan, César Sánchez, W. Robinson, B. Finkbeiner, H. Sipma, S. Mehrotra, Z. Manna: *LOLA: Runtime Monitoring of Synchronous Systems*. TIME 2005



History of Stream Runtime Verification

B. D'Angelo, S. Sankaranarayanan, César Sánchez, W. Robinson, B. Finkbeiner, H. Sipma, S. Mehrotra, Z. Manna: *LOLA: Runtime Monitoring of Synchronous Systems*. TIME 2005

A. Pnueli, A. Zaks: *PSL Model Checking and Run-Time Verification Via Testers*. FM 2006



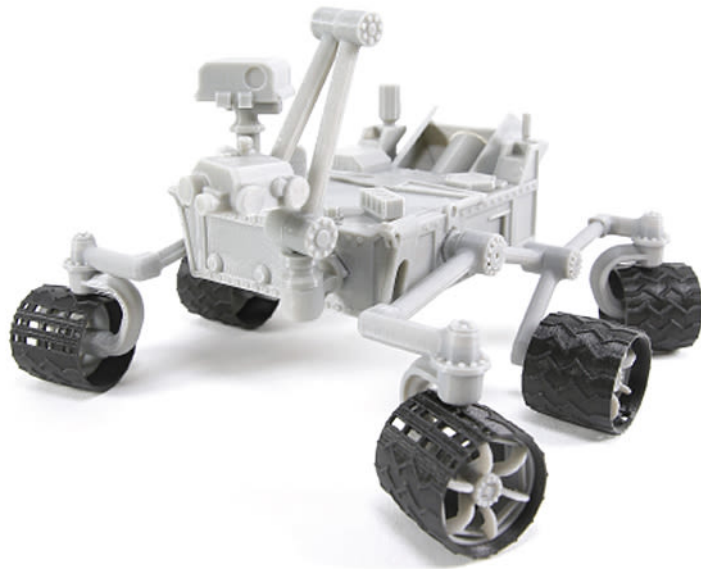
History of Stream Runtime Verification

B. D'Angelo, S. Sankaranarayanan, César Sánchez, W. Robinson, B. Finkbeiner, H. Sipma, S. Mehrotra, Z. Manna: *LOLA: Runtime Monitoring of Synchronous Systems*. TIME 2005

A. Pnueli, A. Zaks: *PSL Model Checking and Run-Time Verification Via Testers*. FM 2006

L. Pike, A. Goodloe, R. Morisset, S. Niller: *Copilot: A Hard Real-Time Runtime Monitor*. RV 2010

T. Reinbacher, K. Rozier, J. Schumann: *Temporal-Logic Based Runtime Observer Pairs for System Health Management of Real-Time Systems*. TACAS 2014



History of Stream Runtime Verification

B. D'Angelo, S. Sankaranarayanan, César Sánchez, W. Robinson, B. Finkbeiner, H. Sipma, S. Mehrotra, Z. Manna: *LOLA: Runtime Monitoring of Synchronous Systems*. TIME 2005

A. Pnueli, A. Zaks: *PSL Model Checking and Run-Time Verification Via Testers*. FM 2006

L. Pike, A. Goodloe, R. Morisset, S. Niller: *Copilot: A Hard Real-Time Runtime Monitor*. RV 2010

T. Reinbacher, K. Rozier, J. Schumann: *Temporal-Logic Based Runtime Observer Pairs for System Health Management of Real-Time Systems*. TACAS 2014

P. Faymonville, B. Finkbeiner, S. Schirmer, H. Torfah: *A Stream-Based Specification Language for Network Monitoring*. RV 2016



History of Stream Runtime Verification



F. Adolf, P. Faymonville, B. Finkbeiner, S. Schirmer, C. Torens: *Stream Runtime Monitoring on UAS*. RV 2017

History of Stream Runtime Verification

B. D'Angelo, S. Sankaranarayanan, César Sánchez, W. Robinson, B. Finkbeiner, H. Sipma, S. Mehrotra, Z. Manna: *LOLA: Runtime Monitoring of Synchronous Systems*. TIME 2005

A. Pnueli, A. Zaks: *PSL Model Checking and Run-Time Verification Via Testers*. FM 2006

L. Pike, A. Goodloe, R. Morisset, S. Niller: *Copilot: A Hard Real-Time Runtime Monitor*. RV 2010

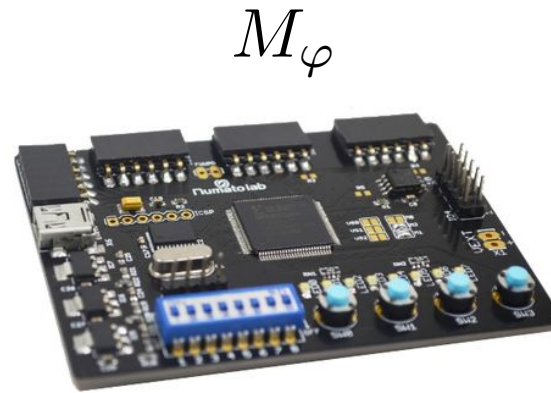
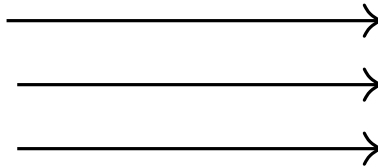
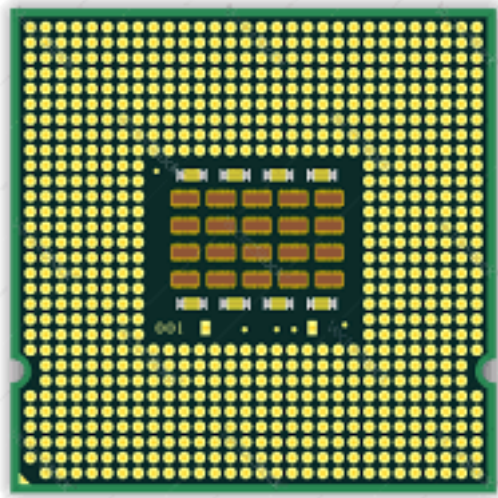
T. Reinbacher, K. Rozier, J. Schumann: *Temporal-Logic Based Runtime Observer Pairs for System Health Management of Real-Time Systems*. TACAS 2014

P. Faymonville, B. Finkbeiner, S. Schirmer, H. Torfah: *A Stream-Based Specification Language for Network Monitoring*. RV 2016

F. Adolf, P. Faymonville, B. Finkbeiner, S. Schirmer, C. Torens: *Stream Runtime Monitoring on UAS*. RV 2017

L. Bozzelli, C. Sánchez: *Foundations of Boolean Stream Runtime Verification* RV 2014

History of Stream Runtime Verification



M. Leucker, C. Sánchez, T. Scheffel, M. Schmitz, A. Schramm: *TeSSLa: Runtime Verification of Non-synchronized Real-Time Streams*. SAC 2018

Motivation (expressivity)

Consider the following LTL specs:

$$\Box p$$

Motivation (expressivity)

Consider the following LTL specs:

$$\Box p \qquad s := p \wedge s[1, \text{true}]$$

Motivation (expressivity)

Consider the following LTL specs:

$$\Box p \qquad s := p \wedge s[1, \text{true}]$$

$$\Box p$$

Motivation (expressivity)

Consider the following LTL specs:

$$\Box p \qquad s := p \wedge s[1, \text{true}]$$

$$\Box p \qquad s := p \wedge s[-1, \text{true}]$$

Motivation (expressivity)

Consider the following LTL specs:

$$\Box p \qquad s := p \wedge s[1, \text{true}]$$

$$\Box p \qquad s := p \wedge s[-1, \text{true}]$$

$$\Diamond p$$

Motivation (expressivity)

Consider the following LTL specs:

$$\Box p \qquad s := p \wedge s[1, \text{true}]$$

$$\Box p \qquad s := p \wedge s[-1, \text{true}]$$

$$\Diamond p \qquad s := p \vee s[1, \text{false}]$$

Motivation (expressivity)

Consider the following LTL specs:

$$\Box p \qquad s := p \wedge s[1, \text{true}]$$

$$\Box p \qquad s := p \wedge s[-1, \text{true}]$$

$$\Diamond p \qquad s := p \vee s[1, \text{false}]$$

$$\Diamond p$$

Motivation (expressivity)

Consider the following LTL specs:

$$\Box p \qquad s := p \wedge s[1, \text{true}]$$

$$\Box p \qquad s := p \wedge s[-1, \text{true}]$$

$$\Diamond p \qquad s := p \vee s[1, \text{false}]$$

$$\Diamond p \qquad s := p \vee s[-1, \text{false}]$$

Motivation (expressivity)

Consider the following LTL specs:

$$\Box p \qquad s := p \wedge s[1, \text{true}]$$

$$\Box p \qquad s := p \wedge s[-1, \text{true}]$$

$$\Diamond p \qquad s := p \vee s[1, \text{false}]$$

$$\Diamond p \qquad s := p \vee s[-1, \text{false}]$$

$$p \mathcal{U} q$$

Motivation (expressivity)

Consider the following LTL specs:

$$\Box p \qquad s := p \wedge s[1, \text{true}]$$

$$\Box p \qquad s := p \wedge s[-1, \text{true}]$$

$$\Diamond p \qquad s := p \vee s[1, \text{false}]$$

$$\Diamond p \qquad s := p \vee s[-1, \text{false}]$$

$$p \mathcal{U} q \qquad s := q \vee (p \wedge s[1, \text{false}])$$

Motivation (expressivity)

Consider the following LTL specs:

$$\Box p \qquad s := p \wedge s[1, \text{true}]$$

$$\Box p \qquad s := p \wedge s[-1, \text{true}]$$

$$\Diamond p \qquad s := p \vee s[1, \text{false}]$$

$$\Diamond p \qquad s := p \vee s[-1, \text{false}]$$

$$p \mathcal{U} q \qquad s := q \vee (p \wedge s[1, \text{false}])$$

$$p \mathcal{W} q$$

Motivation (expressivity)

Consider the following LTL specs:

$$\Box p \qquad s := p \wedge s[1, \text{true}]$$

$$\Box p \qquad s := p \wedge s[-1, \text{true}]$$

$$\Diamond p \qquad s := p \vee s[1, \text{false}]$$

$$\Diamond p \qquad s := p \vee s[-1, \text{false}]$$

$$p \mathcal{U} q \qquad s := q \vee (p \wedge s[1, \text{false}])$$

$$p \mathcal{W} q \qquad s := q \vee (p \wedge s[1, \text{true}])$$

Motivation (expressivity)

Consider the following LTL specs:

$$\Box p \qquad s := p \wedge s[1, \text{true}]$$

$$\Box p \qquad s := p \wedge s[-1, \text{true}]$$

$$\Diamond p \qquad s := p \vee s[1, \text{false}]$$

$$\Diamond p \qquad s := p \vee s[-1, \text{false}]$$

$$p \mathcal{U} q \qquad s := q \vee (p \wedge s[1, \text{false}])$$

$$p \mathcal{W} q \qquad s := q \vee (p \wedge s[1, \text{true}])$$

$$\bigcirc p \qquad s := p[1, \text{false}]$$

Motivation (expressivity)

Consider the following LTL specs:

Why restrict to Booleans?

$\Box p$

$s := p \wedge s[1, \text{true}]$

$\Box p$

$s := p \wedge s[-1, \text{true}]$

$\Diamond p$

$s := p \vee s[1, \text{false}]$

$\Diamond p$

$s := p \vee s[-1, \text{false}]$

$p \mathcal{U} q$

$s := q \vee (p \wedge s[1, \text{false}])$

$p \mathcal{W} q$

$s := q \vee (p \wedge s[1, \text{true}])$

$\bigcirc p$

$s := p[1, \text{false}]$

Separation of concerns

A **runtime verification** algorithm deals with two aspects:

- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

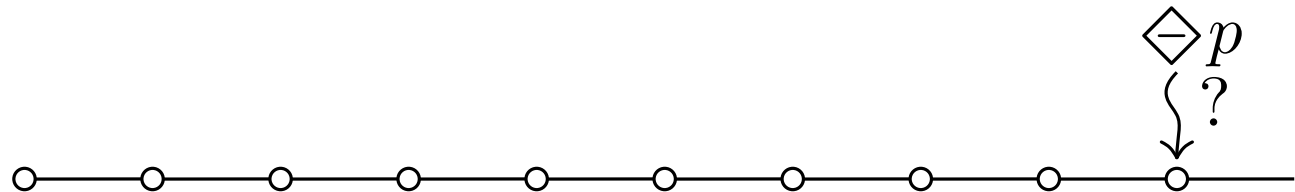
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$\Diamond p$



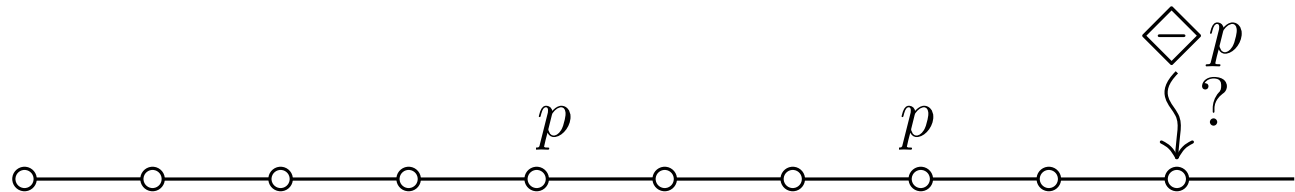
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$\Diamond p$



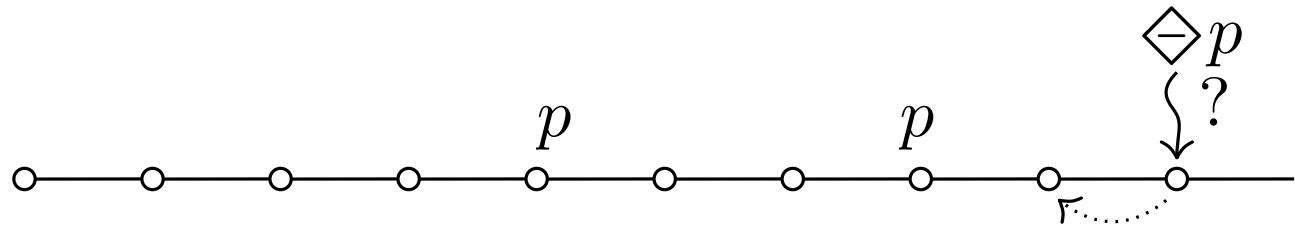
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$\Diamond p$



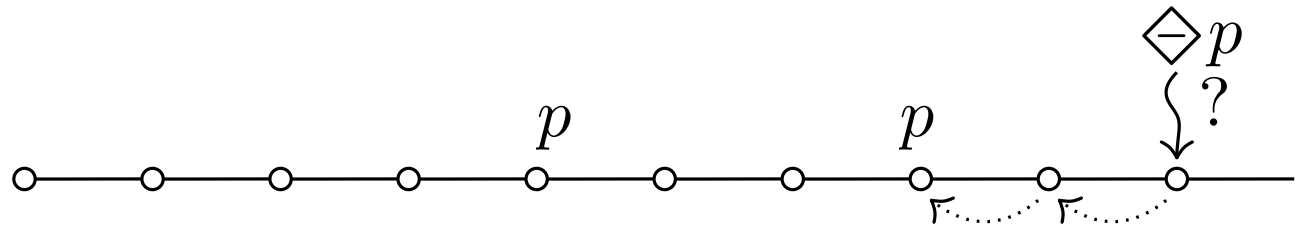
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$\Diamond p$



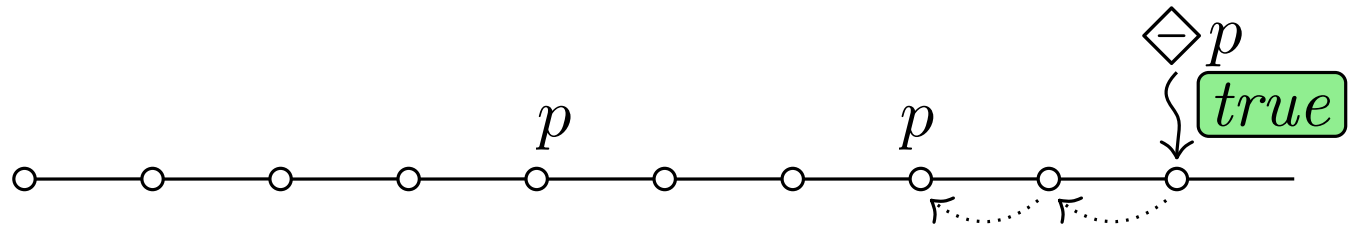
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$\Diamond p$



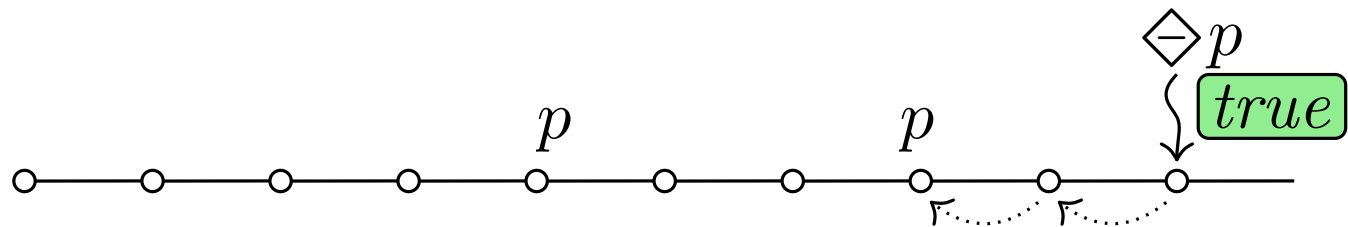
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$\Diamond p$



$s := p \vee s[-1, \text{false}]$

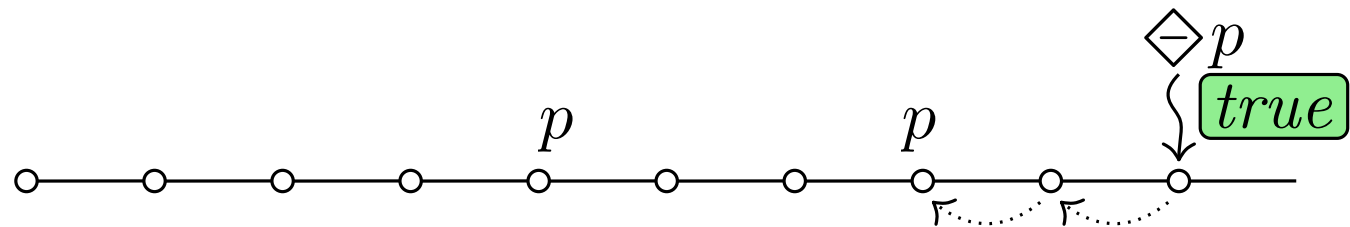
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

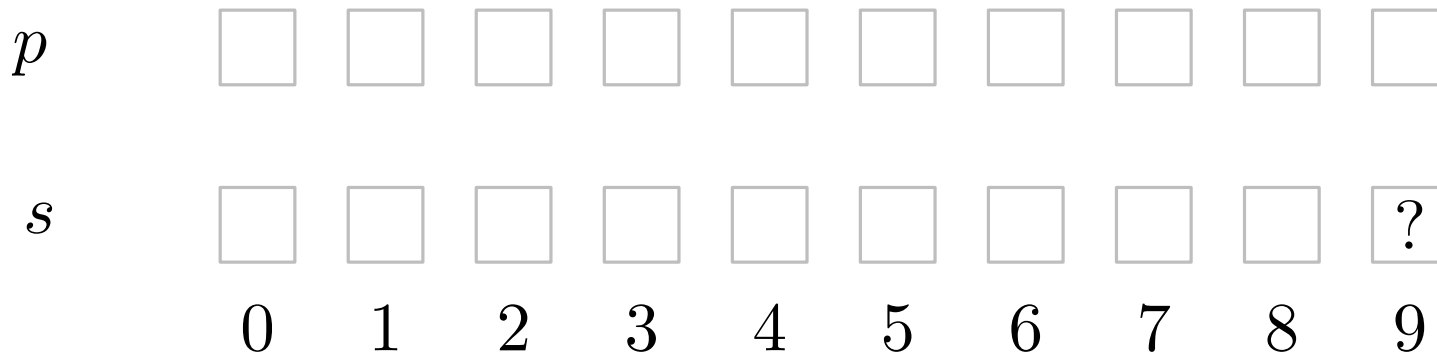
- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$\Diamond p$



$s := p \vee s[-1, \text{false}]$



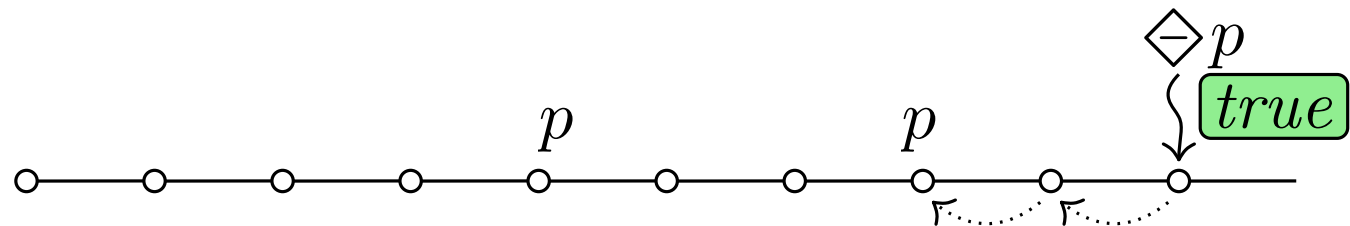
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

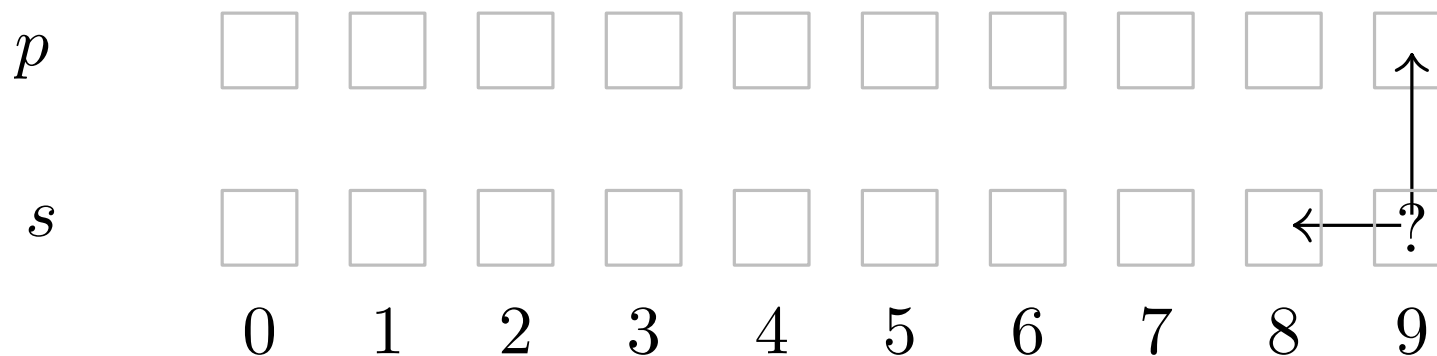
- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$\Diamond p$



$s := p \vee s[-1, \text{false}]$



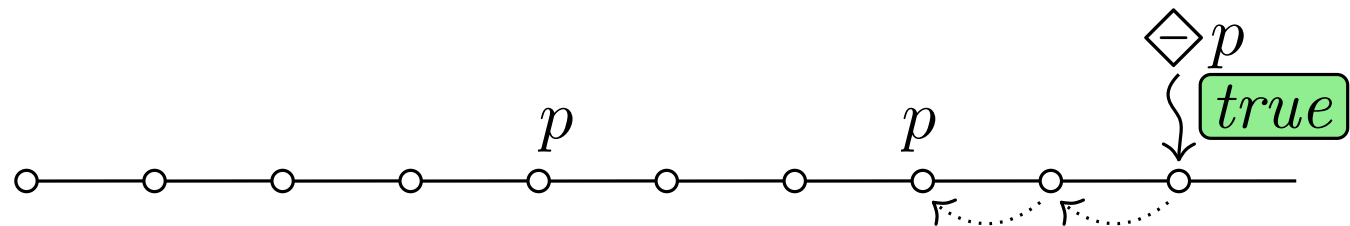
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

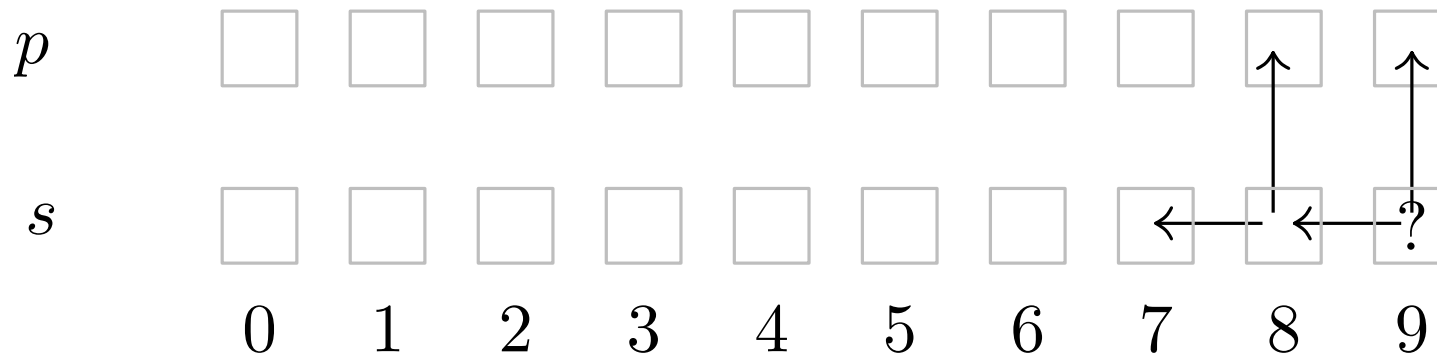
- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$\Diamond p$



$s := p \vee s[-1, \text{false}]$



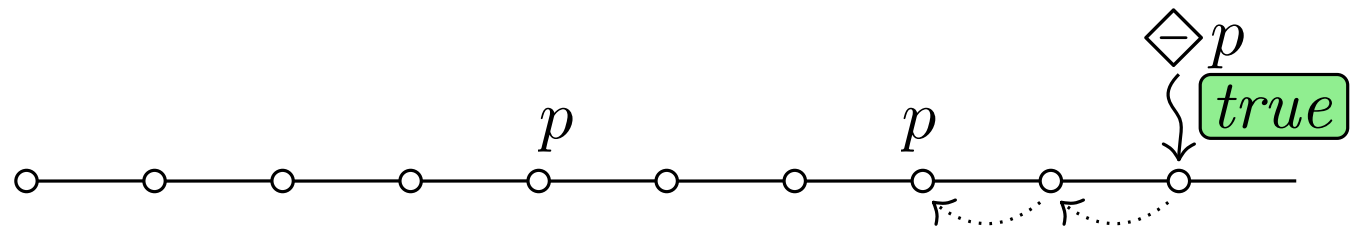
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

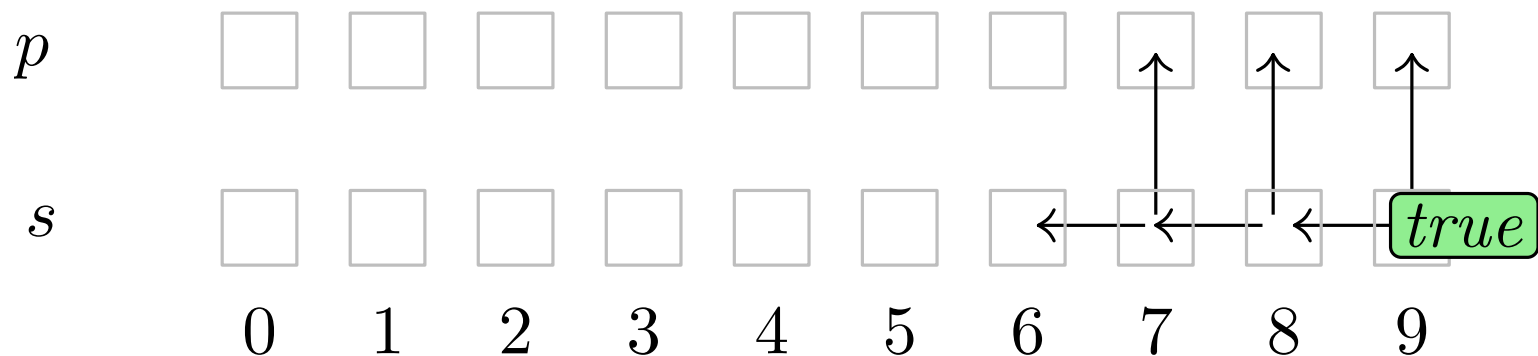
- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$\Diamond p$



$s := p \vee s[-1, \text{false}]$



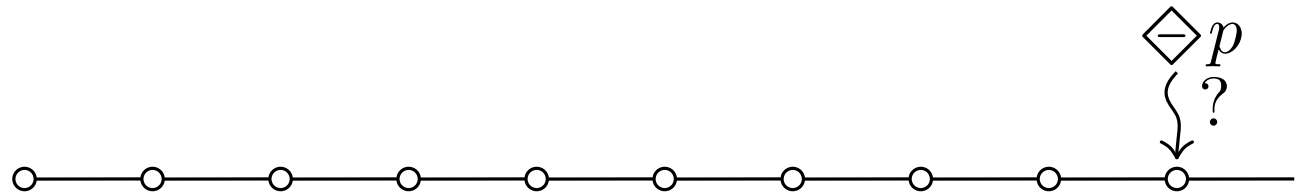
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$\Diamond p$



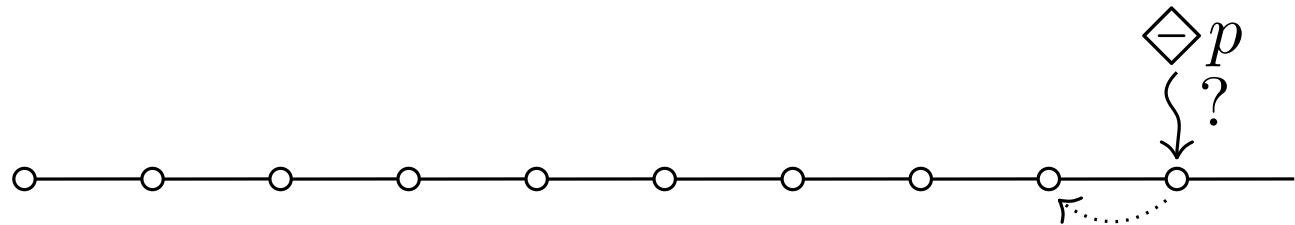
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$\Diamond p$



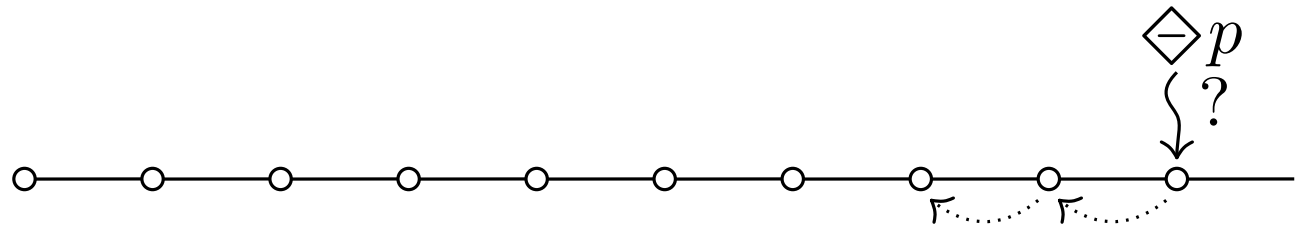
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$\Diamond p$



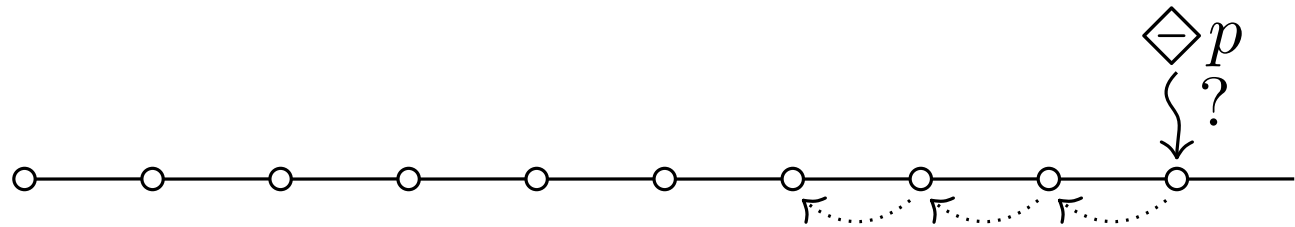
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$\Diamond p$



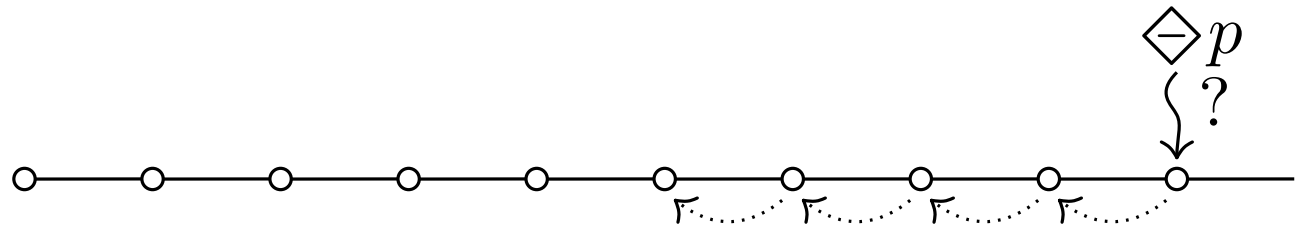
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$\Diamond p$



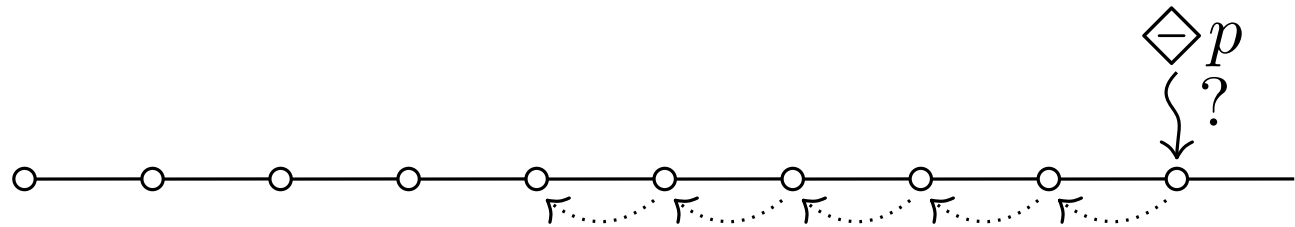
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$\Diamond p$



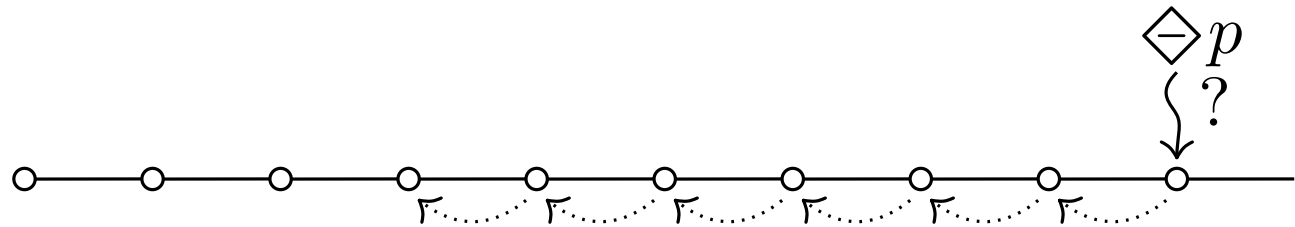
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$\Diamond p$



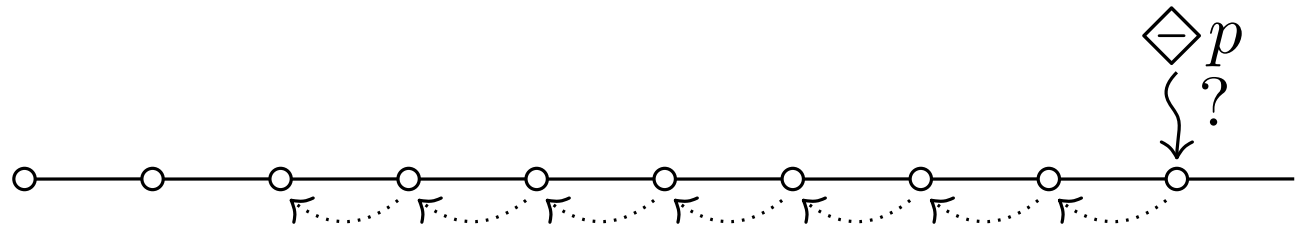
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$\Diamond p$



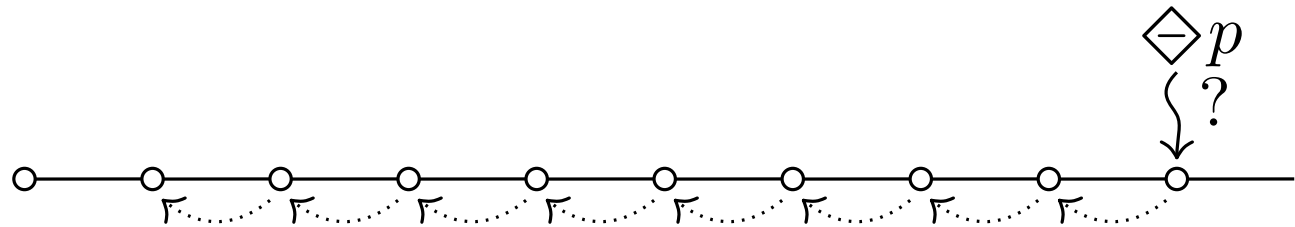
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$\Diamond p$



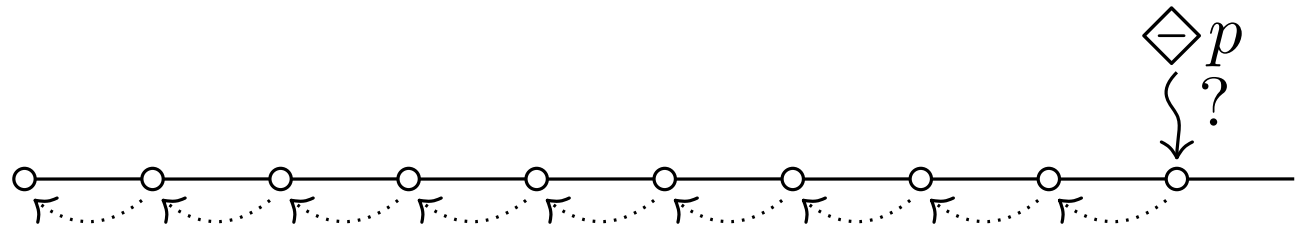
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$\Diamond p$



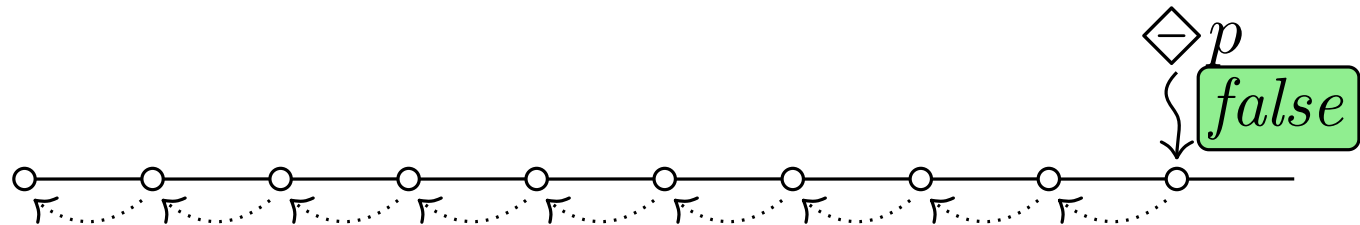
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$\Diamond p$



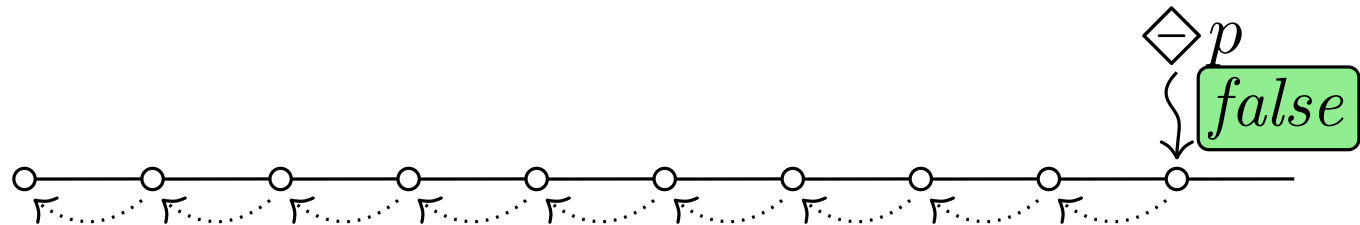
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$\Diamond p$



$s := p \vee s[-1, \text{false}]$

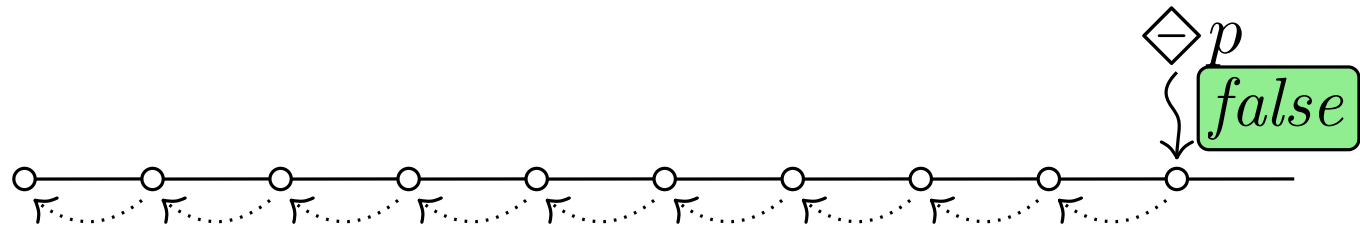
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

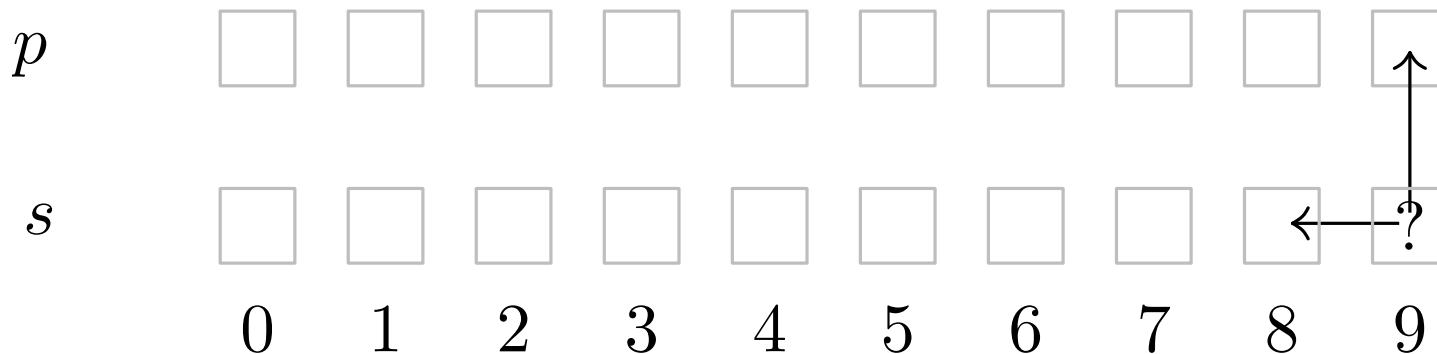
- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$$\Diamond p$$



$$s := p \vee s[-1, \text{false}]$$



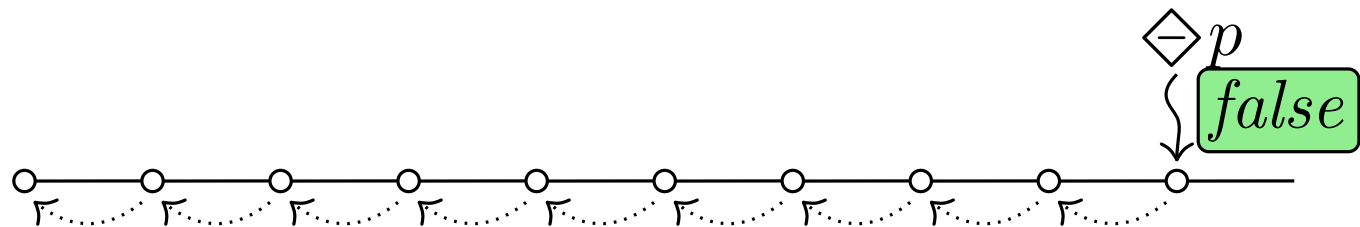
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

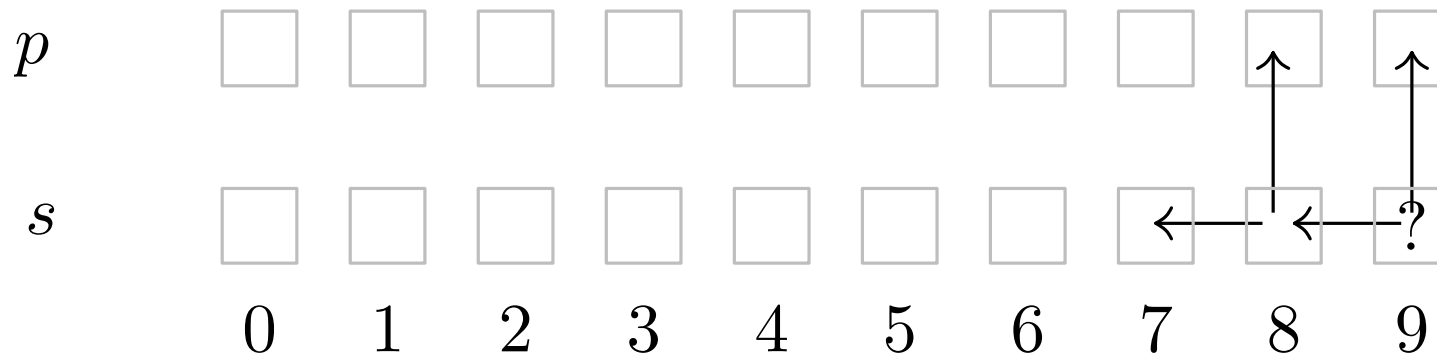
- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$$\Diamond p$$



$$s := p \vee s[-1, \text{false}]$$



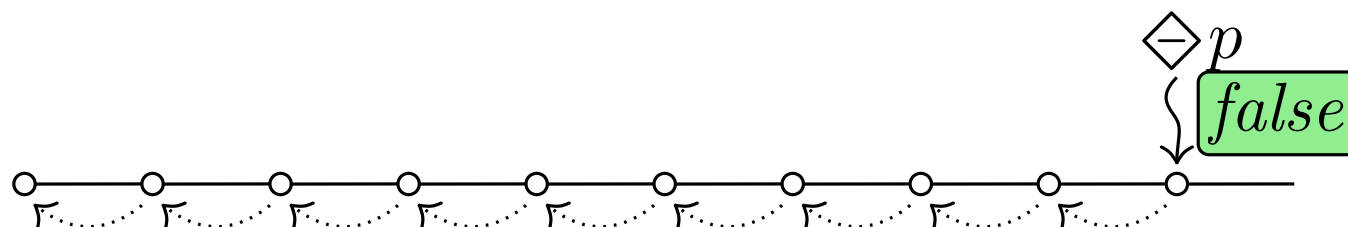
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

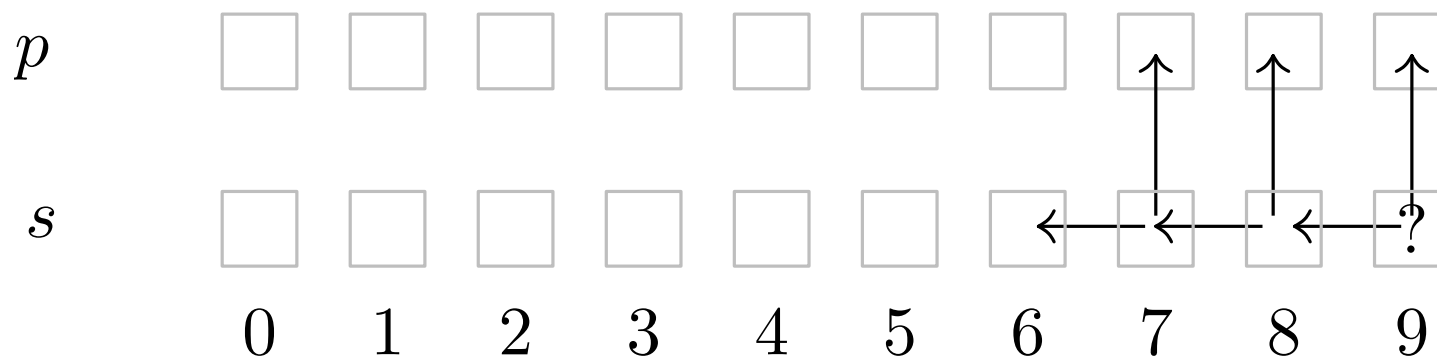
- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$$\Diamond p$$



$$s := p \vee s[-1, \text{false}]$$



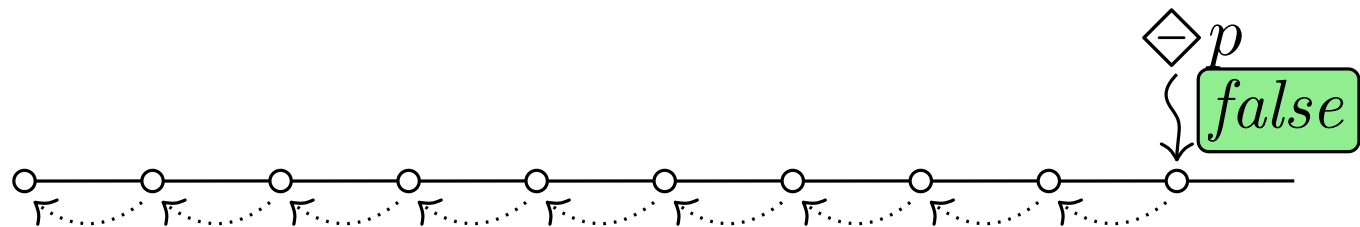
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

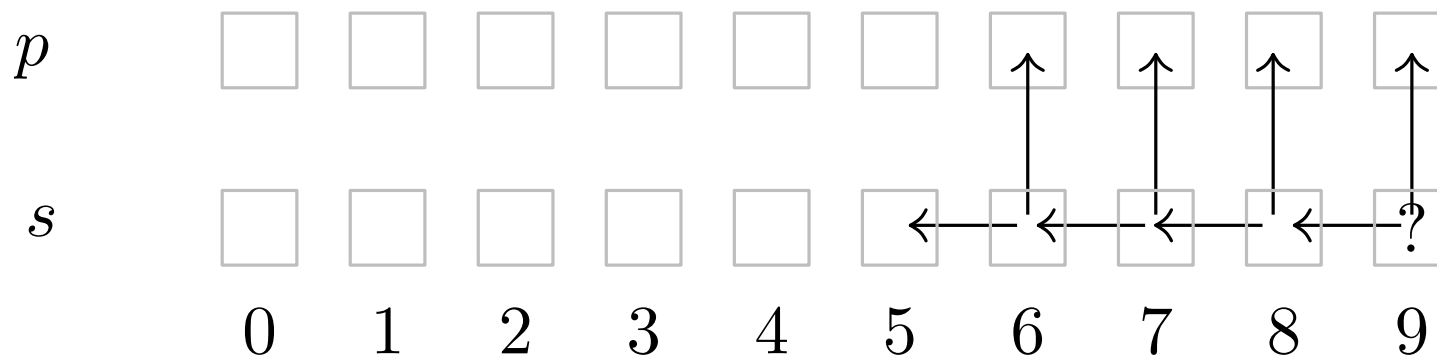
- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$\Diamond p$



$s := p \vee s[-1, \text{false}]$



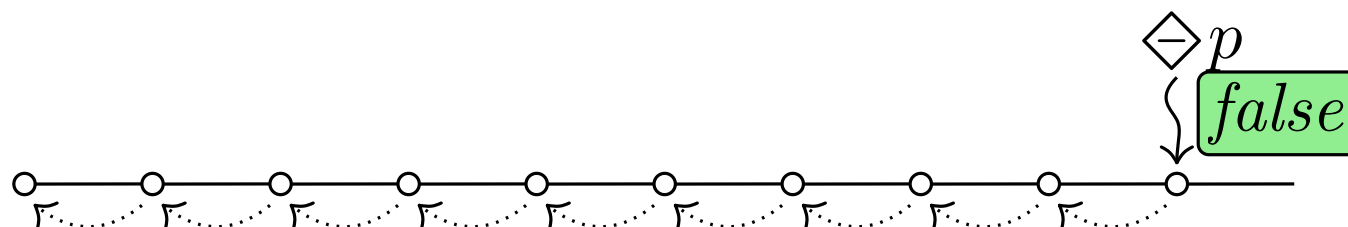
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

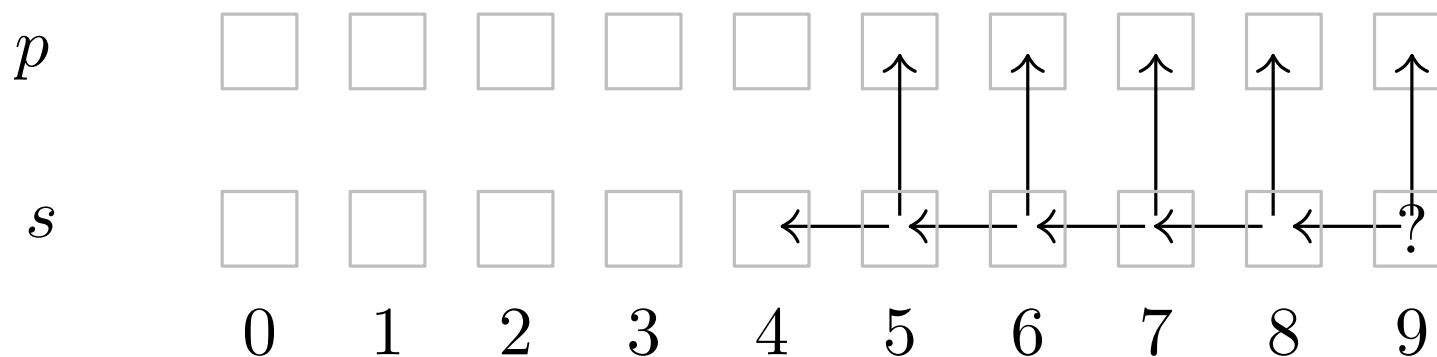
- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$$\Diamond p$$



$$s := p \vee s[-1, \text{false}]$$



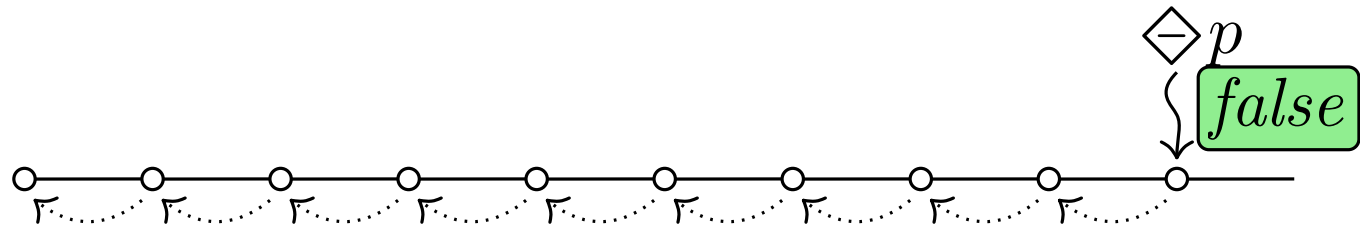
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

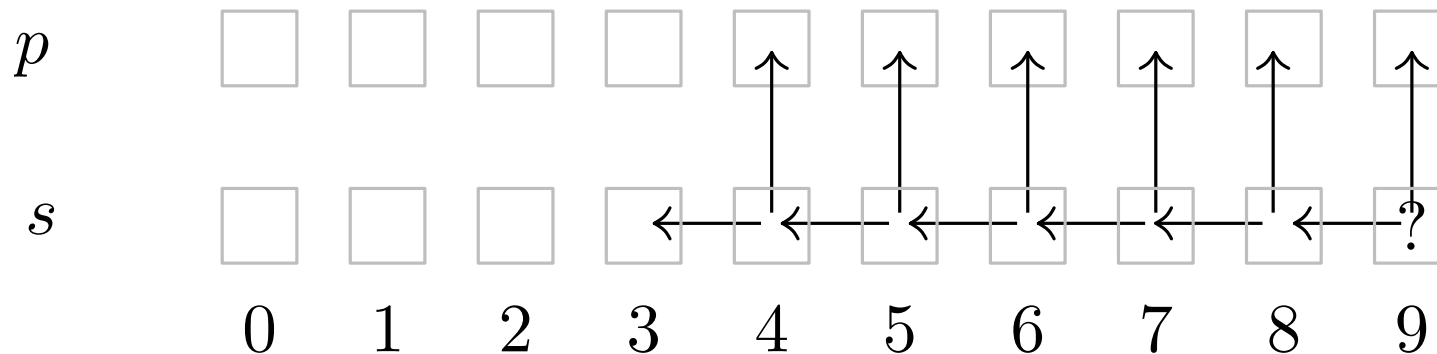
- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$$\Diamond p$$



$$s := p \vee s[-1, \text{false}]$$



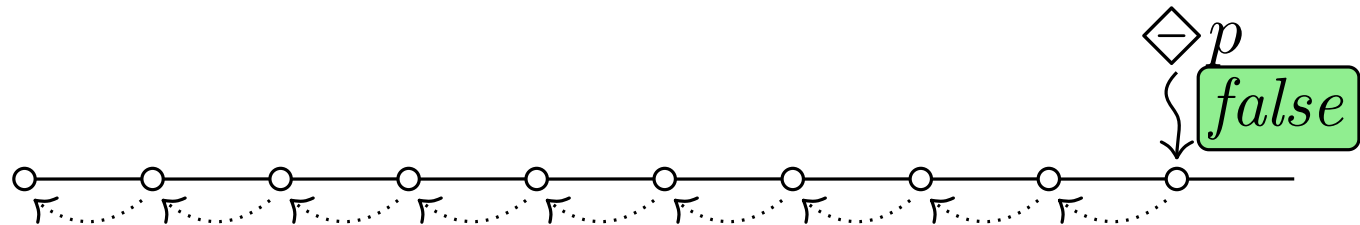
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

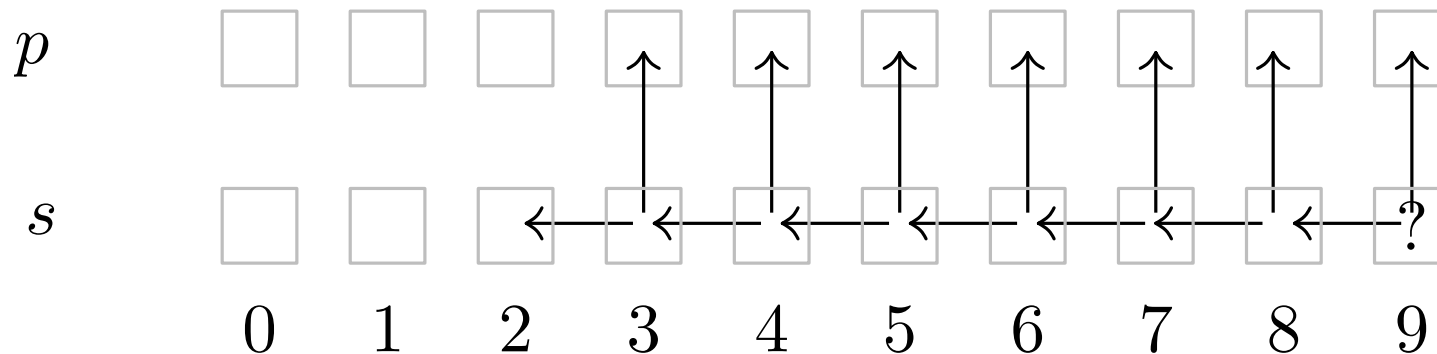
- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$$\Diamond p$$



$$s := p \vee s[-1, \text{false}]$$



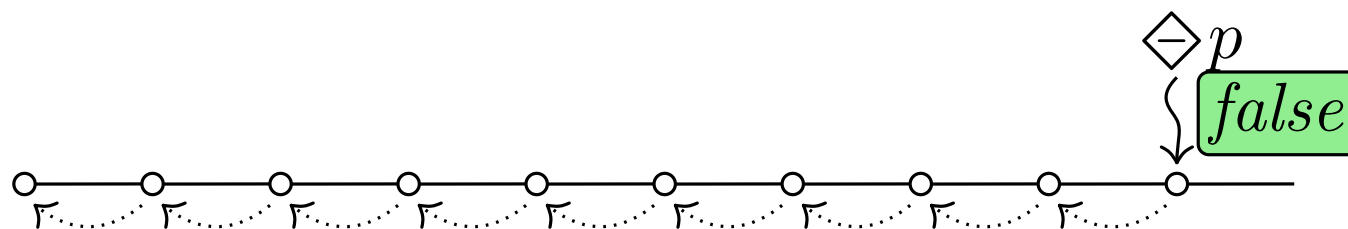
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

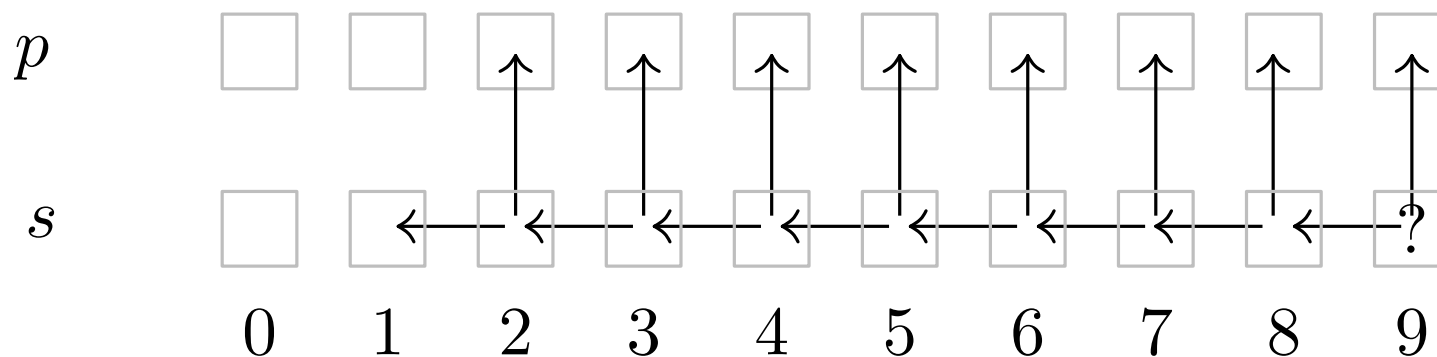
- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$$\Diamond p$$



$$s := p \vee s[-1, \text{false}]$$



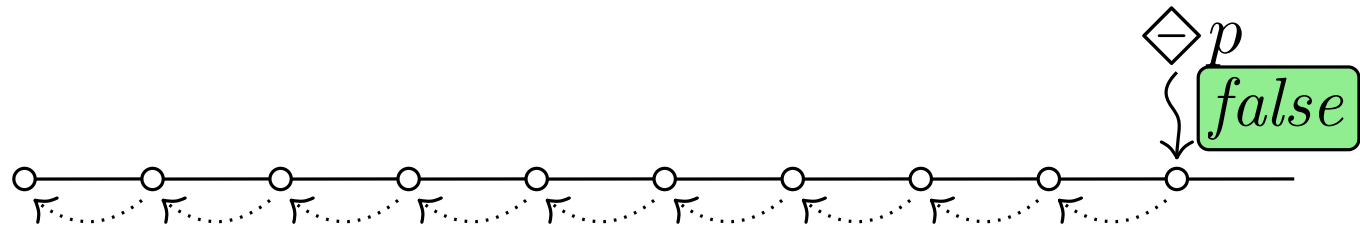
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

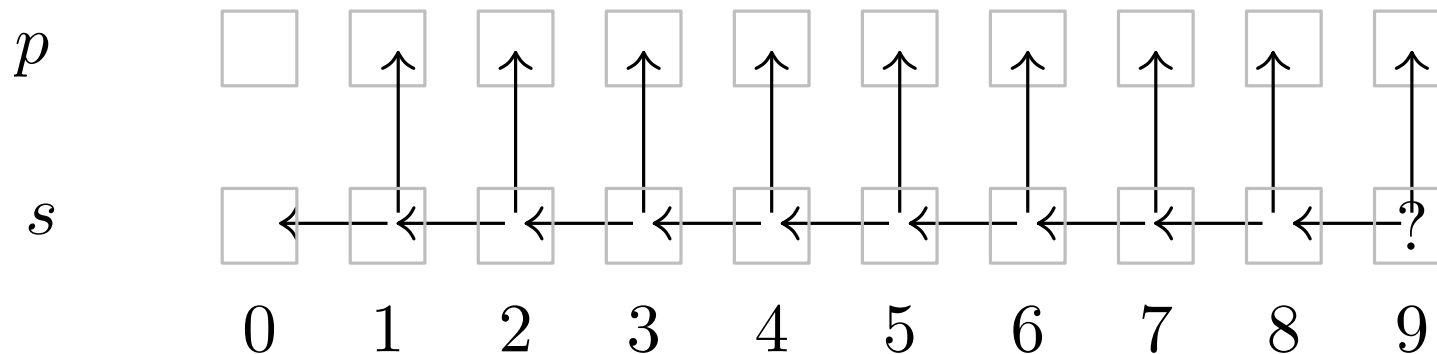
- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$\Diamond p$



$s := p \vee s[-1, \text{false}]$



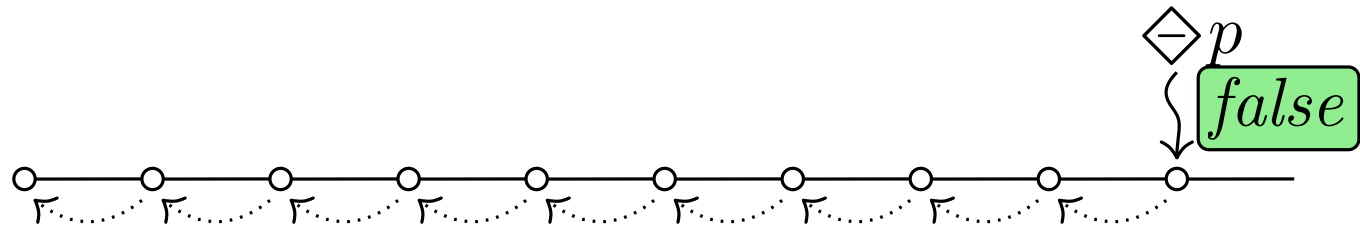
Separation of concerns

A **runtime verification** algorithm deals with two aspects:

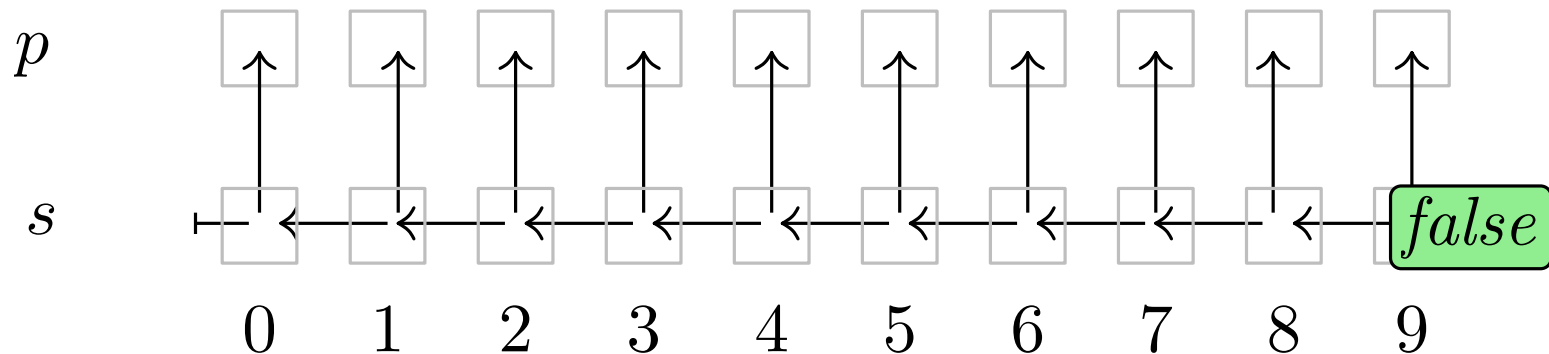
- ▶ an algorithm: a sequence of actions and computations that determine the verdict (*time*)
- ▶ the details how to compute each action (*the data*)

Example:

$$\Diamond p$$



$$s := p \vee s[-1, \text{false}]$$



Domains

- ▶ Domains model the *data* that monitors maintain
- ▶ Domains are *sorted* first-order theories such that:
 - all functions are interpreted
 - all theories have an `(if · then · else ·)`

Domains

- ▶ Domains model the *data* that monitors maintain
- ▶ Domains are *sorted* first-order theories such that:
 - all functions are interpreted
 - all theories have an (if \cdot then \cdot else \cdot)

Notes

- ▶ All terms are typed
- ▶ All functions \mathfrak{f} allow to construct terms
(given terms $e_1 \dots e_k$, \mathfrak{f} builds a new term $\mathfrak{f}(e_1, \dots, e_k)$)
- ▶ All functions have an interpretation
(given values $v_1 \dots v_k$, f computes a result $f(v_1, \dots, v_k)$)

Domains (examples)

Domain of Booleans

sorts: `bool`

► Syntax

Constants:

`false true` : `bool`

Functions:

`\wedge \vee \rightarrow \leftrightarrow` : `bool \times bool \rightarrow bool`

`if \cdot then \cdot else \cdot` : `bool \times bool \times bool \rightarrow bool`

Domains (examples)

Domain of Booleans

sorts: **bool**

► Syntax

Constants:

`false` `true` : **bool**

Functions:

`∧` `∨` `→` `↔` : **bool** × **bool** → **bool**

`if` · `then` · `else` · : **bool** × **bool** × **bool** → **bool**

► Example terms:

$x \wedge (\text{true} \vee y)$ `true` $x \rightarrow y$

► Example evaluation:

$T \wedge (T \vee F) \mapsto T \wedge T \mapsto T$

Domains (examples)

Domain of Booleans 3

sorts: **bool3**

► Syntax

Constants:

`false` `true` `?` : **bool3**

Functions:

`∧` `∨` `→` `↔` : **bool3** × **bool3** → **bool3**

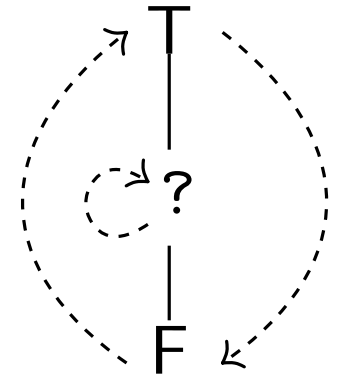
`if` · `then` · `else` · : **bool** × **bool3** × **bool3** → **bool3**

► Example terms:

$x \wedge (\text{true} \vee y)$ `true` $x \rightarrow y$ $? \vee x$

► Example evaluation:

$T \wedge (? \vee F) \mapsto T \wedge ? \mapsto ?$



Domains (examples)

Domain of Booleans 4

sorts: **bool4**

► Syntax

Constants:

false **true** \top_p \perp_p : **bool4**

Functions:

\wedge \vee \rightarrow \leftrightarrow : **bool4** \times **bool4** \rightarrow **bool4**

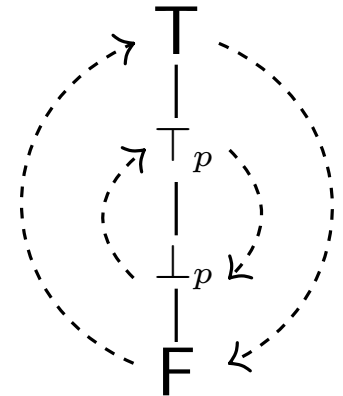
if \cdot *then* \cdot *else* \cdot : **bool** \times **bool4** \times **bool4** \rightarrow **bool4**

► Example terms:

$x \wedge (\top_p \vee y)$ **true** $x \rightarrow y$ $\perp_p \vee x$

► Example evaluation:

$\top \wedge (\top_p \vee \mathbf{F}) \mapsto \top \wedge \top_p \mapsto \top_p$



Domains (examples)

Domain of Integers

sorts: **bool**, **int**

► Syntax

Constants:

$\dots, -2, -1, 0, 1, 2 \dots$: **int**

Functions:

$+, -, *, /$: **int** \times **int** \rightarrow **int**

$=, \neq, <, \leq$: **int** \times **int** \rightarrow **bool**

if \cdot *then* \cdot *else* \cdot : **bool** \times **int** \times **int** \rightarrow **int**

Domains (examples)

Domain of Integers

sorts: **bool**, **int**

► Syntax

Constants:

$\dots, -2, -1, 0, 1, 2 \dots$: **int**

Functions:

$+, -, *, /$: **int** \times **int** \rightarrow **int**

$=, \neq, <, \leq$: **int** \times **int** \rightarrow **bool**

if \cdot *then* \cdot *else* \cdot : **bool** \times **int** \times **int** \rightarrow **int**

► Example terms:

$x + (3 - y)$

$x \leq y$

if $x \leq z$
then 3 *else* $(z + 1)$

Domains (examples)

Domain of Integers

sorts: **bool**, **int**

► Syntax

Constants:

$\dots, -2, -1, 0, 1, 2 \dots$: **int**

Functions:

$+, -, *, /$: **int** \times **int** \rightarrow **int**

$=, \neq, <, \leq$: **int** \times **int** \rightarrow **bool**

if \cdot *then* \cdot *else* \cdot : **bool** \times **int** \times **int** \rightarrow **int**

► Example terms:

$x + (3 - y)$

$x \leq y$

if $x \leq z$
then 3 *else* $(z + 1)$

► Example evaluation:

if $7 \leq 3$
then 3 *else* $(3 + 1)$ $\mapsto 3 + 1 \mapsto 4$

Domains (examples)

Domain of Sets

sorts: **bool**, **set**, **elem**

► Syntax

Constants:

\emptyset : **set**

Functions:

$[t]$: **elem** \rightarrow **set**

\in : **elem** \times **set** \rightarrow **bool**

\cup, \cap, \setminus : **set** \times **set** \rightarrow **set**

if \cdot *then* \cdot *else* \cdot : **bool** \times **set** \times **set** \rightarrow **set**

Domains (examples)

Domain of Sets

sorts: **bool**, **set**, **elem**

► Syntax

Constants:

\emptyset : **set**

Functions:

$[t]$: **elem** \rightarrow **set**

\in : **elem** \times **set** \rightarrow **bool**

\cup, \cap, \setminus : **set** \times **set** \rightarrow **set**

$\text{if } \cdot \text{ then } \cdot \text{ else } \cdot$: **bool** \times **set** \times **set** \rightarrow **set**

► Example terms:

$(x \cup [3])$

$x \setminus z$

$\text{if } 2 \in x$
 $\text{then } y \text{ else } (y \cup [2])$

Domains (examples)

Domain of Sets

sorts: **bool**, **set**, **elem**

► Syntax

Constants:

\emptyset : **set**

Functions:

$[t]$: **elem** \rightarrow **set**

\in : **elem** \times **set** \rightarrow **bool**

\cup, \cap, \setminus : **set** \times **set** \rightarrow **set**

$\text{if } \cdot \text{ then } \cdot \text{ else } \cdot$: **bool** \times **set** \times **set** \rightarrow **set**

► Example terms:

$(x \cup [3])$

$x \setminus z$

$\text{if } 2 \in x$
 $\text{then } y \text{ else } (y \cup [2])$

► Example evaluation:

$\text{if } 2 \in \{1, 2, 3\}$
 $\text{then } \{1, 2, 3\} \setminus [2]; \text{ else } \emptyset \quad \mapsto \quad \{1, 2, 3\} \setminus [2] \quad \mapsto \quad \{1, 3\}$

Domains (Simplifiers)

One important aspect *in practice* is simplification.

Sometimes terms $e(a, b)$ can be evaluated without knowing *all* parameters a, b .

Domains (Simplifiers)

One important aspect *in practice* is simplification.

Sometimes terms $e(a, b)$ can be evaluated without knowing *all* parameters a, b .

Examples:

Domains (Simplifiers)

One important aspect *in practice* is simplification.

Sometimes terms $e(a, b)$ can be evaluated without knowing *all* parameters a, b .

Examples:

`if true then 17 else f(x,y) \mapsto 17`

Domains (Simplifiers)

One important aspect *in practice* is simplification.

Sometimes terms $e(a, b)$ can be evaluated without knowing *all* parameters a, b .

Examples:

`if true then 17 else f(x,y)` \mapsto 17

`x ∧ true` \mapsto `x`

Domains (Simplifiers)

One important aspect *in practice* is simplification.

Sometimes terms $e(a, b)$ can be evaluated without knowing *all* parameters a, b .

Examples:

`if true then 17 else f(x,y)` \mapsto 17

$x \wedge \text{true}$ \mapsto x

$x * 0$ \mapsto 0

Domains (Simplifiers)

One important aspect *in practice* is simplification.

Sometimes terms $e(a, b)$ can be evaluated without knowing *all* parameters a, b .

Examples:

$$\text{if true then 17 else } f(x, y) \mapsto 17$$

$$x \wedge \text{true} \mapsto x$$

$$x * 0 \mapsto 0$$

We capture simplifications as *rewrite* rules

Domains (Simplifiers)

One important aspect *in practice* is simplification.

Sometimes terms $e(a, b)$ can be evaluated without knowing *all* parameters a, b .

Examples:

`if true then 17 else f(x,y)` \mapsto 17

$x \wedge \text{true}$ \mapsto x

$x * 0$ \mapsto 0

We capture simplifications as *rewrite* rules

Question: can we aim at *perfect* simplifiers?

Domains (Simplifiers)

One important aspect *in practice* is simplification.

Sometimes terms $e(a, b)$ can be evaluated without knowing *all* parameters a, b .

Examples:

`if true then 17 else f(x,y)` \mapsto 17

$x \wedge \text{true}$ \mapsto x

$x * 0$ \mapsto 0

We capture simplifications as *rewrite* rules

Question: can we aim at *perfect* simplifiers?

NO! (complexity and decidability)

Example

Every request has a response

Example

Every request has a response

$$\Box(\textit{req} \rightarrow \Diamond \textit{resp})$$

Example

Every request has a response

$$\Box(\textit{req} \rightarrow \Diamond \textit{resp})$$

With Booleans:

evresp := *resp* \vee *evresp*[1|false]

granted := *req* \rightarrow *evresp*

ok := *granted* \wedge *ok*[1|true]

Example

Every request has a response

$$\Box(\textit{req} \rightarrow \Diamond \textit{resp})$$

With Integers:

```
nreq      := nreq[-1|0] + if req then 1 else 0  
nresp     := nresp[-1|0] + if resp then 1 else 0  
ok       := last  $\rightarrow$  (nreq = nresp)
```

where

```
last      := false[1,true]
```

Example

Every request has a response

$$\Box(\textit{req} \rightarrow \Diamond \textit{resp})$$

With Integers:

```
nreq      := nreq[-1|0] + if req then 1 else 0  
nresp     := nresp[-1|0] + if resp then 1 else 0  
ok       := last  $\rightarrow$  (nreq = nresp)
```

where

```
last      := false[1,true]
```

An additional sanity check:

```
good      := nresp  $\leq$  nreq  
G_good    := G_good[-1,true]  $\wedge$  good
```

Example

Every request has a response

$$\Box(\textit{req} \rightarrow \Diamond \textit{resp})$$

With Sets:

$$\begin{aligned} \textit{pending} &:= \textit{pending}[-1|\emptyset] \cup \left(\textit{if } \textit{req} \textit{ then } [\textit{reqid}] \textit{ else } \emptyset \right) \\ &\quad \setminus \left(\textit{if } \textit{resp} \textit{ then } [\textit{respid}] \textit{ else } \emptyset \right) \\ \textit{ok} &:= \textit{last} \rightarrow (\textit{pending} = \emptyset) \end{aligned}$$

Example using Bool 4

$$\Box p \quad s := p \wedge s[1|\text{true}]$$

$$\Box p \quad s := p \wedge s[-1|\text{true}]$$

$$\Diamond p \quad s := p \vee s[1|\text{false}]$$

$$\Diamond p \quad s := p \vee s[-1|\text{false}]$$

$$p\mathcal{U}q \quad s := q \vee (p \wedge s[1|\text{false}])$$

$$p\mathcal{W}q \quad s := q \vee (p \wedge s[1|\text{true}])$$

$$\bigcirc p \quad s := p[1|\text{false}]$$

Example using Bool 4

$$\Box p \quad s := p \wedge s[1|\perp_p]$$

$$\Box p \quad s := p \wedge s[-1|\text{true}]$$

$$\Diamond p \quad s := p \vee s[1|\top_p]$$

$$\Diamond p \quad s := p \vee s[-1|\text{false}]$$

$$p\mathcal{U}q \quad s := q \vee (p \wedge s[1|\perp_p])$$

$$p\mathcal{W}q \quad s := q \vee (p \wedge s[1|\top_p])$$

$$\bigcirc p \quad s := p[1|\text{false}]$$

Stream Runtime Verification Syntax

Stream Runtime Verification syntax

A specification consists of:

Stream Runtime Verification syntax

A specification consists of:

- ▶ **inputs** (with their **types**)

```
input bool     $t_1$   
input int      $t_2$   
input string   $t_3$   
...
```

Stream Runtime Verification syntax

A specification consists of:

- ▶ **inputs** (with their **types**)
- ▶ **output** (with their **types**)

```
input bool     $t_1$ 
input int      $t_2$ 
input string   $t_3$ 
...
output bool    $s_1$ 
output int     $s_2$ 
...
```

Stream Runtime Verification syntax

A specification consists of:

- ▶ **inputs** (with their **types**)
- ▶ **output** (with their **types**)
- ▶ how **outputs** depend on **inputs** and **outputs**

```
input bool     $t_1$ 
input int      $t_2$ 
input string   $t_3$ 
...

output bool    $s_1$    $:= e_1(t_1, t_2, \dots s_1, s_2, \dots)$ 
output int     $s_2$    $:= e_2(t_2, t_2, \dots s_1, s_2, \dots)$ 
...           ...
```

Stream Runtime Verification syntax

A specification consists of:

- ▶ **inputs** (with their **types**)
- ▶ **output** (with their **types**)
- ▶ how **outputs** depend on **inputs** and **outputs**
- ▶ optionally **triggers** to notify the user

```
input bool     $t_1$ 
input int      $t_2$ 
input string   $t_3$ 
...

output bool    $s_1$    $:= e_1(t_1, t_2, \dots s_1, s_2, \dots)$ 
output int     $s_2$    $:= e_2(t_2, t_2, \dots s_1, s_2, \dots)$ 
...          ...

trigger  $T_1, T_2, \dots$ 
```

Stream Runtime Verification syntax

A specification consists of:

- ▶ inputs (with their types)
- ▶ output (with their types)
- ▶ how outputs depend on inputs and outputs
- ▶ optionally triggers to notify the user

```
input bool     $t_1$ 
input int      $t_2$ 
input string   $t_3$ 
...

output bool    $s_1$    $:= e_1(t_1, t_2, \dots s_1, s_2, \dots)$ 
output int     $s_2$    $:= e_2(t_2, t_2, \dots s_1, s_2, \dots)$ 
...          ...

trigger  $T_1, T_2, \dots$ 
```

Stream Runtime Verification syntax

A specification consists of:

- ▶ inputs (with their types)
- ▶ output (with their types)
- ▶ how outputs depend on inputs and outputs
- ▶ optionally triggers to notify the user

input bool
input int
input string

t_1
 t_2
 t_3

← *independent* stream variables

...
output bool s_1 $:= e_1(t_1, t_2, \dots s_1, s_2, \dots)$
output int s_2 $:= e_2(t_2, t_2, \dots s_1, s_2, \dots)$
...

trigger T_1, T_2, \dots

Stream Runtime Verification syntax

A specification consists of:

- ▶ inputs (with their types)
- ▶ output (with their types)
- ▶ how outputs depend on inputs and outputs
- ▶ optionally triggers to notify the user

input bool
input int
input string
...

t_1
 t_2
 t_3

independent stream variables

dependent stream variables

output bool
output int
...

s_1
 s_2

$:= e_1(t_1, t_2, \dots, s_1, s_2, \dots)$

$:= e_2(t_2, t_2, \dots, s_1, s_2, \dots)$

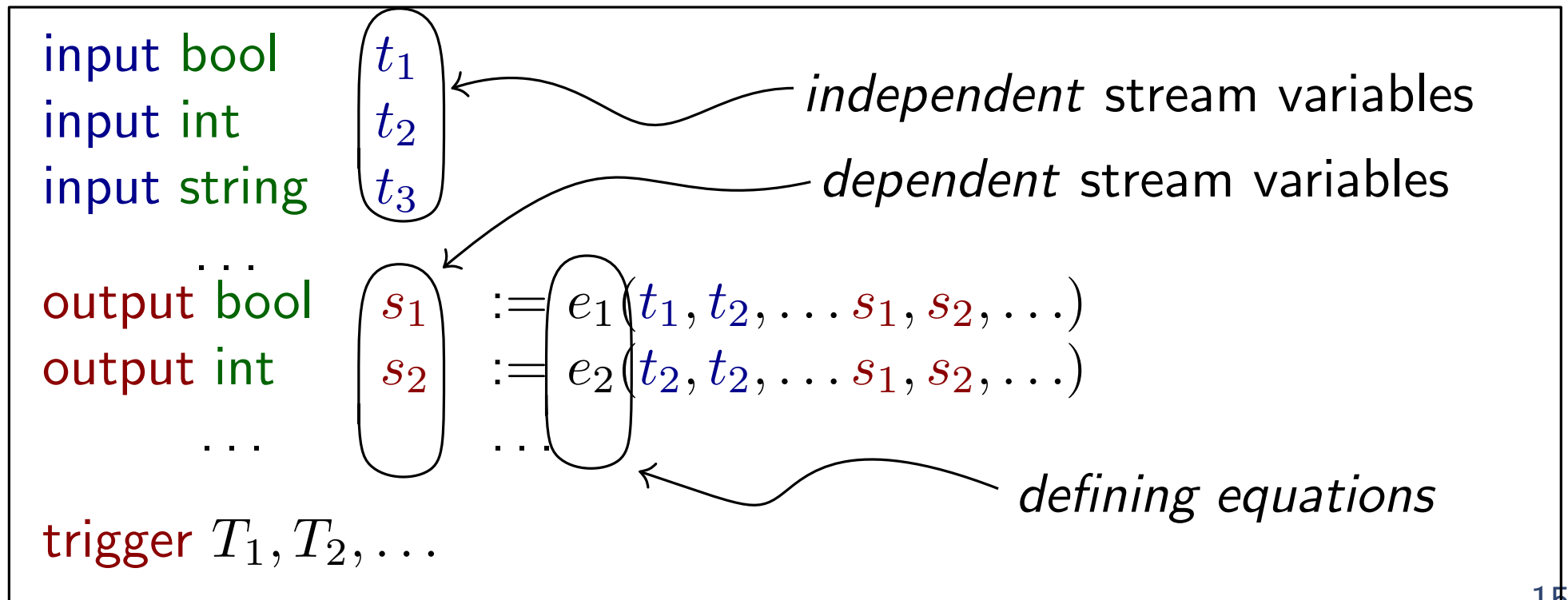
...

trigger T_1, T_2, \dots

Stream Runtime Verification syntax

A specification consists of:

- ▶ inputs (with their types)
- ▶ output (with their types)
- ▶ how outputs depend on inputs and outputs
- ▶ optionally triggers to notify the user



SRV syntax (defining equations)

output bool $s_1 := e_1(t_1, t_2, \dots s_1 s_2, \dots)$

A *defining equation* e is a stream expression:

SRV syntax (defining equations)

output bool $s_1 := e_1(t_1, t_2, \dots s_1 s_2, \dots)$

A *defining equation* e is a stream expression:

► **constant:** c of type T (assuming $c : T$)

SRV syntax (defining equations)

output bool $s_1 := e_1(t_1, t_2, \dots, s_1 s_2, \dots)$

A *defining equation* e is a stream expression:

- ▶ **constant:** c of type T (assuming $c : T$)
- ▶ **stream:** v_i of type T (assuming **output** T s_i or **input** T t_i)

SRV syntax (defining equations)

output bool $s_1 := e_1(t_1, t_2, \dots, s_1 s_2, \dots)$

A *defining equation* e is a stream expression:

- ▶ **constant:** c of type T (assuming $c : T$)
- ▶ **stream:** v_i of type T (assuming **output** T s_i or **input** T t_i)
- ▶ **function application:** $f(e_1, e_2, \dots, e_n)$ of type T
 (assuming $f^k :: T_1 \times T_2 \times \dots \times T_k \rightarrow T$ and
 $e_1 : T_1 \dots e_k : T_k$)

SRV syntax (defining equations)

output bool $s_1 := e_1(t_1, t_2, \dots s_1 s_2, \dots)$

A *defining equation* e is a stream expression:

- ▶ **constant:** c of type T (assuming $c : T$)
- ▶ **stream:** v_i of type T (assuming **output** T s_i or **input** T t_i)
- ▶ **function application:** $f(e_1, e_2, \dots, e_n)$ of type T
 (assuming $f^k :: T_1 \times T_2 \times \dots \times T_k \rightarrow T$ and
 $e_1 : T_1 \dots e_k : T_k$)
- ▶ **offset:** $e[i, d]$ of type T (assuming $e : T$)

SRV syntax (defining equations)

output bool $s_1 := e_1(t_1, t_2, \dots s_1 s_2, \dots)$

A *defining equation* e is a stream expression:

- ▶ **constant:** c of type T (assuming $c : T$)
- ▶ **stream:** v_i of type T (assuming **output** T s_i or **input** T t_i)
- ▶ **function application:** $\mathbf{f}(e_1, e_2, \dots, e_n)$ of type T

(assuming $f^k :: T_1 \times T_2 \times \dots \times T_k \rightarrow T$ and $e_1 : T_1 \dots e_k : T_k$)

- **offset:** $e[i, d]$ of type T (assuming $e : T$)

$\dots, -2, -1, 0, 1, \dots$

Examples

output bool *ok* := true

Examples

input int h

output int $height := h$

Examples

input int n
input bool ok

output bool $resp := ok \wedge (n \geq 0)$

Examples

input int n

output int $m := (n^2 + 7) \bmod 16$

Examples

input **int** n

input bool *cond*

```
output int      resp := if cond then n
                                     else n + 1
```

Examples

| | | |
|-------|------|-------------|
| input | int | s_4, t_3 |
| input | bool | $cond, s_3$ |

```
output bool   resp := if cond then  $t_3 < s_4$ 
                                else  $\neg s_3$ 
```

Examples

input bool *in*

output bool *succ* := *in*[+1, false]

Examples

input bool *in*

output bool *succ* := *in*[+1, false]

output bool *prev* := *in*[-1, false]

Examples

input **int** *inbit*

output **int** *par* := *par*[-1, 0] + (*inbit mod* 2)

Examples

input **bool** *req, resp*

output **bool** *ok* $:= resp \vee (\neg req \wedge ok[+1, \text{false}])$

Normalized Specifications

A specification is *normalized* if for every

$$\text{output dom } s := e$$

the equation e is of the form:

- ▶ **constant:** c
- ▶ **stream variable:** t or s_2
- ▶ **function** over stream variables: $f(t, s_2)$
- ▶ **shift** over stream variables: $s[k, d]$ or $t[j, c]$

Normalized Specifications

Example:

input int t_1, t_3, t_4, t_5

input bool t_2

output int $s := t_1[1, 0] + (\text{if } t_2[-1, \text{true}]$
then t_3
else $t_4 + t_5)$

Normalized Specifications

Example:

input int t_1, t_3, t_4, t_5

input bool t_2

[illegible]

can be normalized to:

output int $s := s_1 + s_2$

output int $s_1 := t_1[1, 0]$

output int $s_2 := \text{if } s_3 \text{ then } t_3 \text{ else } s_4$

output bool $s_3 := t_2[-1, \text{true}]$

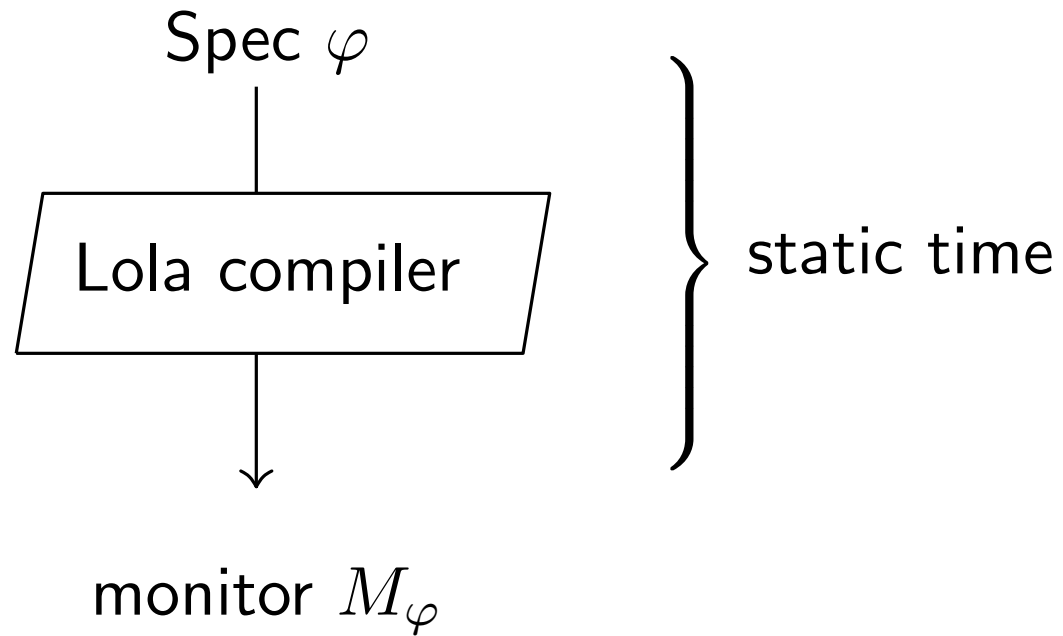
output int $s_4 := t_4 + t_5$

Stream Runtime Verification Semantics

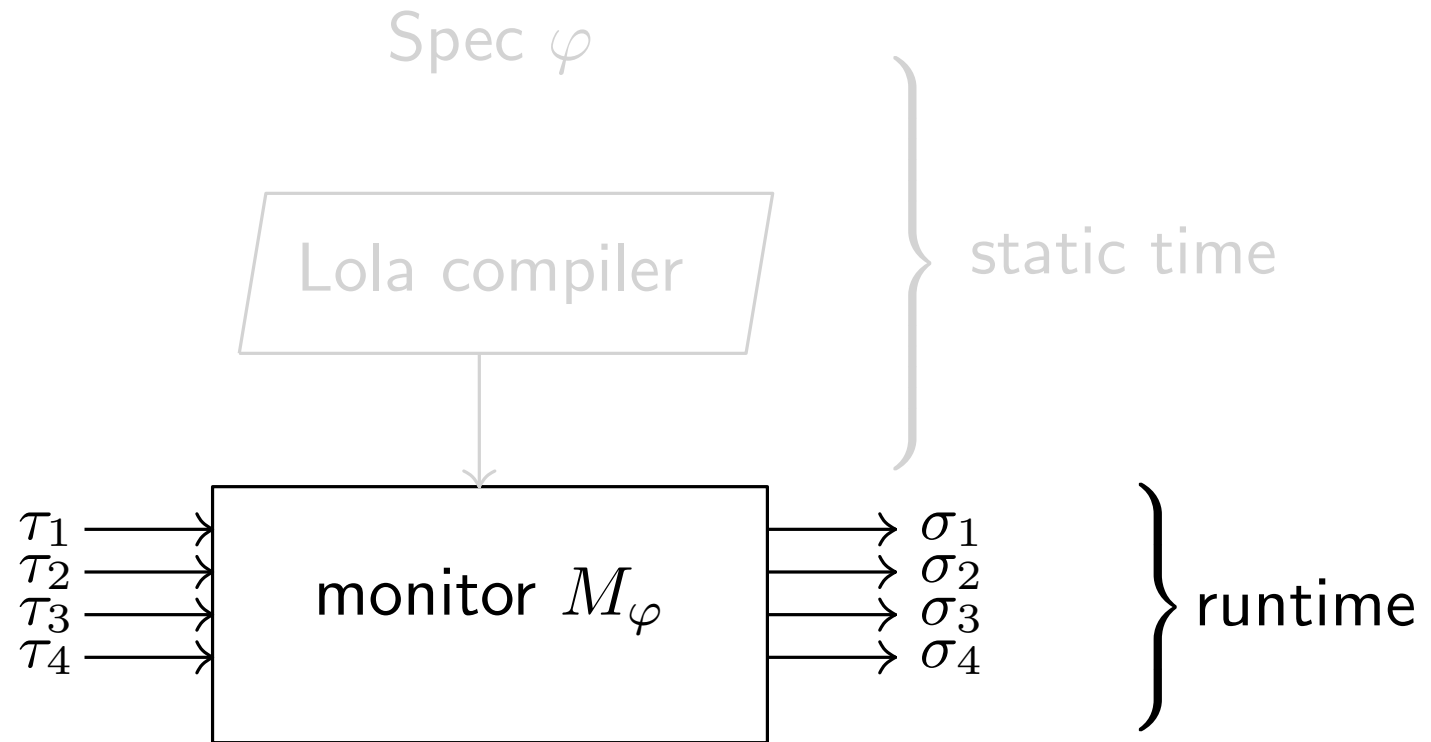
Semantics (intention)

Spec φ

Semantics (intention)



Semantics (intention)



Intention: M_φ is a “*function*” from inputs to outputs

Stream Runtime Verification semantics (valuation)

Consider **input** stream vars:

t_1

t_2

...

t_n

and **output** stream vars:

s_1

s_2

...

s_m

Stream Runtime Verification semantics (valuation)

Consider **input** stream vars:

t_1

t_2

...

t_n

and **output** stream vars:

s_1

s_2

...

s_m

a **valuation** of length N is an assignment
of a stream of values of length N for each stream variable

Stream Runtime Verification semantics (valuation)

Consider **input** stream vars:

$t_1 \rightarrow \tau_1$

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T | F | F | T | F | T | F | F | F | T | F | F | T | F | F | T | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

$t_2 \rightarrow \tau_2$

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 1 | 0 | 0 | 2 | 9 | 1 | 3 | 0 | 3 | 7 | 3 | 1 | 6 | 3 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

...

$t_n \rightarrow \tau_n$

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T | F | F | T | F | T | F | F | F | T | F | F | T | F | F | T | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

and **output** stream vars:

$s_1 \rightarrow \sigma_1$

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 1 | 0 | 0 | 2 | 9 | 1 | 3 | 0 | 3 | 7 | 3 | 1 | 6 | 3 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

$s_2 \rightarrow \sigma_2$

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T | F | F | T | F | T | F | F | F | T | F | F | T | F | F | T | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

...

$s_m \rightarrow \sigma_m$

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 1 | 0 | 0 | 2 | 9 | 1 | 3 | 0 | 3 | 7 | 3 | 1 | 6 | 3 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

a **valuation** of length N is an assignment
of a stream of values of length N for each stream variable

Stream Runtime Verification semantics (valuation)

Given valuation of length N :

$$t_1 \rightarrow \tau_1 \quad t_2 \rightarrow \tau_2 \qquad t_n \rightarrow \tau_n$$

$$s_1 \rightarrow \sigma_1 \quad s_2 \rightarrow \sigma_2 \qquad s_m \rightarrow \sigma_m$$

the *semantics* $\llbracket \cdot \rrbracket$ of an expression e is defined as a stream of length N :

Stream Runtime Verification semantics (valuation)

Given valuation of length N :

$$t_1 \rightarrow \tau_1 \quad t_2 \rightarrow \tau_2 \qquad t_n \rightarrow \tau_n$$

$$s_1 \rightarrow \sigma_1 \quad s_2 \rightarrow \sigma_2 \qquad s_m \rightarrow \sigma_m$$

the *semantics* $\llbracket \cdot \rrbracket$ of an expression e is defined as a stream of length N :

► **constant:** c

Stream Runtime Verification semantics (valuation)

Given valuation of length N :

$$t_1 \rightarrow \tau_1 \quad t_2 \rightarrow \tau_2$$

$$t_n \rightarrow \tau_n$$

$$s_1 \rightarrow \sigma_1 \quad s_2 \rightarrow \sigma_2$$

$$s_m \rightarrow \sigma_m$$

the *semantics* $\llbracket \cdot \rrbracket$ of an expression e is defined as a stream of length N :

► **constant:** c

$$\llbracket c \rrbracket(j) = c$$

Stream Runtime Verification semantics (valuation)

Given valuation of length N :

$$t_1 \rightarrow \tau_1 \quad t_2 \rightarrow \tau_2$$

$$t_n \rightarrow \tau_n$$

$$s_1 \rightarrow \sigma_1 \quad s_2 \rightarrow \sigma_2$$

$$s_m \rightarrow \sigma_m$$

the *semantics* $\llbracket \cdot \rrbracket$ of an expression e is defined as a stream of length N :

► **constant:** c

$$\llbracket c \rrbracket(j) = c$$

► **input var:** t

Stream Runtime Verification semantics (valuation)

Given valuation of length N :

$$t_1 \rightarrow \tau_1 \quad t_2 \rightarrow \tau_2$$

$$t_n \rightarrow \tau_n$$

$$s_1 \rightarrow \sigma_1 \quad s_2 \rightarrow \sigma_2$$

$$s_m \rightarrow \sigma_m$$

the *semantics* $\llbracket \cdot \rrbracket$ of an expression e is defined as a stream of length N :

► **constant:** c

$$\llbracket c \rrbracket(j) = c$$

► **input var:** t

$$\llbracket t \rrbracket(j) = \tau(j)$$

Stream Runtime Verification semantics (valuation)

Given valuation of length N :

$$t_1 \rightarrow \tau_1 \quad t_2 \rightarrow \tau_2$$

$$t_n \rightarrow \tau_n$$

$$s_1 \rightarrow \sigma_1 \quad s_2 \rightarrow \sigma_2$$

$$s_m \rightarrow \sigma_m$$

the *semantics* $\llbracket \cdot \rrbracket$ of an expression e is defined as a stream of length N :

► **constant:** c

$$\llbracket c \rrbracket(j) = c$$

► **input var:** t

$$\llbracket t \rrbracket(j) = \tau(j)$$

► **output var:** s

Stream Runtime Verification semantics (valuation)

Given valuation of length N :

$$t_1 \rightarrow \tau_1 \quad t_2 \rightarrow \tau_2$$

$$t_n \rightarrow \tau_n$$

$$s_1 \rightarrow \sigma_1 \quad s_2 \rightarrow \sigma_2$$

$$s_m \rightarrow \sigma_m$$

the *semantics* $\llbracket \cdot \rrbracket$ of an expression e is defined as a stream of length N :

► **constant:** c

$$\llbracket c \rrbracket(j) = c$$

► **input var:** t

$$\llbracket t \rrbracket(j) = \tau(j)$$

► **output var:** s

$$\llbracket s \rrbracket(j) = \sigma(j)$$

Stream Runtime Verification semantics (valuation)

Given valuation of length N :

$$t_1 \rightarrow \tau_1 \quad t_2 \rightarrow \tau_2$$

$$t_n \rightarrow \tau_n$$

$$s_1 \rightarrow \sigma_1 \quad s_2 \rightarrow \sigma_2$$

$$s_m \rightarrow \sigma_m$$

the *semantics* $\llbracket \cdot \rrbracket$ of an expression e is defined as a stream of length N :

► **constant:** c

$$\llbracket c \rrbracket(j) = c$$

► **input var:** t

$$\llbracket t \rrbracket(j) = \tau(j)$$

► **output var:** s

$$\llbracket s \rrbracket(j) = \sigma(j)$$

► **function:** f

Stream Runtime Verification semantics (valuation)

Given valuation of length N :

$$t_1 \rightarrow \tau_1 \quad t_2 \rightarrow \tau_2 \qquad t_n \rightarrow \tau_n$$

$$s_1 \rightarrow \sigma_1 \quad s_2 \rightarrow \sigma_2 \qquad s_m \rightarrow \sigma_m$$

the *semantics* $\llbracket \cdot \rrbracket$ of an expression e is defined as a stream of length N :

- ▶ **constant:** c $\llbracket c \rrbracket(j) = c$
- ▶ **input var:** t $\llbracket t \rrbracket(j) = \tau(j)$
- ▶ **output var:** s $\llbracket s \rrbracket(j) = \sigma(j)$
- ▶ **function:** f $\llbracket f(e_1, \dots, e_k) \rrbracket(j) = f(\llbracket e_1 \rrbracket(j), \dots, \llbracket e_k \rrbracket(j))$

Stream Runtime Verification semantics (valuation)

Given valuation of length N :

$$t_1 \rightarrow \tau_1 \quad t_2 \rightarrow \tau_2$$

$$t_n \rightarrow \tau_n$$

$$s_1 \rightarrow \sigma_1 \quad s_2 \rightarrow \sigma_2$$

$$s_m \rightarrow \sigma_m$$

the *semantics* $\llbracket \cdot \rrbracket$ of an expression e is defined as a stream of length N :

- ▶ **constant:** c $\llbracket c \rrbracket(j) = c$
- ▶ **input var:** t $\llbracket t \rrbracket(j) = \tau(j)$
- ▶ **output var:** s $\llbracket s \rrbracket(j) = \sigma(j)$
- ▶ **function:** f $\llbracket f(e_1, \dots, e_k) \rrbracket(j) = f(\llbracket e_1 \rrbracket(j), \dots, \llbracket e_k \rrbracket(j))$
- ▶ **shift** $s[k, d](j)$ $\llbracket s[k, d] \rrbracket(j) = \begin{cases} \sigma(j + k) & \text{if } 1 \leq j + k \leq N \\ d & \text{otherwise} \end{cases}$

SRV semantics (denotational)

Given spec φ with output variables:

$$s_1 \quad := \quad e_1$$

$$s_2 \quad := \quad e_2$$

\dots

$$s_m \quad := \quad e_m$$

SRV semantics (denotational)

Given spec φ with output variables:

$$\begin{array}{lll} s_1 & := & e_1 \\ s_2 & := & e_2 \\ & \dots & \\ s_m & := & e_m \end{array}$$

A valuation $\langle \tau_1, \dots, \tau_n, \sigma_1, \dots, \sigma_m \rangle$
is an *evaluation model* of φ whenever

$$\llbracket s_i \rrbracket = \llbracket e_i \rrbracket \quad \text{for every } s_i$$

SRV semantics (denotational)

Given spec φ with output variables:

$$\begin{array}{lll} s_1 & := & e_1 \\ s_2 & := & e_2 \\ & \dots & \\ s_m & := & e_m \end{array}$$

A valuation $\langle \tau_1, \dots, \tau_n, \sigma_1, \dots, \sigma_m \rangle$
is an *evaluation model* of φ whenever

$$\begin{array}{c} \llbracket s_i \rrbracket = \llbracket e_i \rrbracket \quad \text{for every } s_i \\ \uparrow \\ \underbrace{\qquad\qquad\qquad} \\ \llbracket s_i \rrbracket(j) = \llbracket e_i \rrbracket(j) \end{array}$$

SRV semantics (denotational)

Given spec φ with output variables:

$$\begin{array}{lll} s_1 & := & e_1 \\ s_2 & := & e_2 \\ & \dots & \\ s_m & := & e_m \end{array}$$

A valuation $\langle \tau_1, \dots, \tau_n, \sigma_1, \dots, \sigma_m \rangle$
is an *evaluation model* of φ whenever

$$\llbracket s_i \rrbracket = \llbracket e_i \rrbracket \quad \text{for every } s_i$$

If $\langle \tau_1, \dots, \tau_n, \sigma_1, \dots, \sigma_m \rangle$ is an evaluation model of φ we write

$$\langle \tau_1, \dots, \tau_n, \sigma_1, \dots, \sigma_m \rangle \models \varphi$$

SRV semantics (denotational)

Given spec φ with output variables:

$s_1 \quad := \quad e_1$

$s_2 \quad := \quad e_2$

\dots

$s \quad := \quad e$

This semantics *requires the output*

Given input *and output*

\models tells you (YES/NO)

$\llbracket s_i \rrbracket = \llbracket e_i \rrbracket$ for every s_i

If $\langle \tau_1, \dots, \tau_n, \sigma_1, \dots, \sigma_m \rangle$ is an evaluation model of φ we write

$\langle \tau_1, \dots, \tau_n, \sigma_1, \dots, \sigma_m \rangle \models \varphi$

SRV semantics (examples)

input int t
output bool $s := t \leq 10$

SRV semantics (examples)

input int t
output bool $s := t \leq 10$

For τ : 1 2 3 4 5 6 7 8 9 10 11 12

σ : T T T T T T T T T T F F

$$\langle \tau, \sigma \rangle \models \varphi$$

SRV semantics (examples)

input int t
output bool $s := t \leq 10$

For τ : 1 2 3 4 5 6 7 8 9 10 11 12

σ : T T T T T T T T T T F F

$$\langle \tau, \sigma \rangle \models \varphi$$

In fact, σ is the only output for τ

SRV semantics (examples)

input int t
output bool $s := s \wedge t \leq 10$

SRV semantics (examples)

input int t
output bool $s := s \wedge t \leq 10$

For τ : 1 2 3 4 5 6 7 8 9 10 11 12

σ : T T T T T T T T T T F F

$$\langle \tau, \sigma \rangle \models \varphi$$

SRV semantics (examples)

input int t
output bool $s := s \wedge t \leq 10$

For τ : 1 2 3 4 5 6 7 8 9 10 11 12

σ : T T T T T T T T T T F F

$$\langle \tau, \sigma \rangle \models \varphi$$

BUT σ' : F F F F F F F F F F F F

$$\langle \tau, \sigma' \rangle \models \varphi$$

SRV semantics (examples)

input int t
output bool $s := \neg s$

SRV semantics (examples)

input int t
output bool $s := \neg s$

For τ : 1 2 3 4 5 6 7 8 9 10 11 12

There is no σ with

$$\langle \tau, \sigma \rangle \models \varphi$$

Well-defined specifications

Well-defined specifications

def

A spec φ is *well-defined* if
for all **input** streams $\langle \tau_1, \dots, \tau_n \rangle$
there is a unique **output** streams $\langle \sigma_1, \dots, \sigma_m \rangle$ such that
 $\langle \tau_1, \dots, \tau_n, \sigma_1, \dots, \sigma_m \rangle \models \varphi$

- Well-definedness captures that φ is functional

Well-defined specifications

def

A spec φ is *well-defined* if
for all **input** streams $\langle \tau_1, \dots, \tau_n \rangle$
there is a unique **output** streams $\langle \sigma_1, \dots, \sigma_m \rangle$ such that
 $\langle \tau_1, \dots, \tau_n, \sigma_1, \dots, \sigma_m \rangle \models \varphi$

- ▶ Well-definedness captures that φ is functional
- ▶ ... but it is a *semantic* condition
(hard or even impossible to check)

Dependency graph

Goal: to capture the information a stream *may* depend on

Dependency graph

Goal: to capture the information a stream *may* depend on

A *dependency graph* $G : (V, E)$ for a given spec φ
is a weighted multi-graph:

Dependency graph

Goal: to capture the information a stream *may* depend on

A *dependency graph* $G : (V, E)$ for a given spec φ is a weighted multi-graph:

- **Nodes:** V are the stream variables t_i and s_j

Dependency graph

Goal: to capture the information a stream *may* depend on

A *dependency graph* $G : (V, E)$ for a given spec φ is a weighted multi-graph:

- ▶ **Nodes:** V are the stream variables t_i and s_j
- ▶ **Edges:** For every $s_i := e_i$, consider subterms of e_i :
subterm edge

Dependency graph

Goal: to capture the information a stream *may* depend on

A *dependency graph* $G : (V, E)$ for a given spec φ is a weighted multi-graph:

- ▶ **Nodes:** V are the stream variables t_i and s_j
- ▶ **Edges:** For every $s_i := e_i$, consider subterms of e_i :

subterm

t

edge

$s_i \xrightarrow{0} t$

Dependency graph

Goal: to capture the information a stream *may* depend on

A *dependency graph* $G : (V, E)$ for a given spec φ is a weighted multi-graph:

- ▶ **Nodes:** V are the stream variables t_i and s_j
- ▶ **Edges:** For every $s_i := e_i$, consider subterms of e_i :

subterm

edge

t

$s_i \xrightarrow{0} t$

s_j

$s_i \xrightarrow{0} s_j$

Dependency graph

Goal: to capture the information a stream *may* depend on

A *dependency graph* $G : (V, E)$ for a given spec φ is a weighted multi-graph:

- **Nodes:** V are the stream variables t_i and s_j
- **Edges:** For every $s_i := e_i$, consider subterms of e_i :

subterm

edge

t

$s_i \xrightarrow{0} t$

s_j

$s_i \xrightarrow{0} s_j$

$t_j[k, d]$

$s_i \xrightarrow{k} t_j$

Dependency graph

Goal: to capture the information a stream *may* depend on

A *dependency graph* $G : (V, E)$ for a given spec φ is a weighted multi-graph:

- **Nodes:** V are the stream variables t_i and s_j
- **Edges:** For every $s_i := e_i$, consider subterms of e_i :

subterm

edge

t

$s_i \xrightarrow{0} t$

s_j

$s_i \xrightarrow{0} s_j$

$t_j[k, d]$

$s_i \xrightarrow{k} t_j$

$s_j[k, d]$

$s_i \xrightarrow{k} s_j$

Dependency graph (examples)

Consider the following specification:

```
input  int     $t_1, t_2$ 
output int     $s_1 := s_2[1, 0] +$ 
                                     if  $s_2[-1, 7] \leq t_1[1, 0]$ 
                                     then  $s_2[-1, 0]$ 
                                     else  $s_2$ 
output int     $s_2 := s_1 + t_2[-2, 1]$ 
```

Dependency graph (examples)

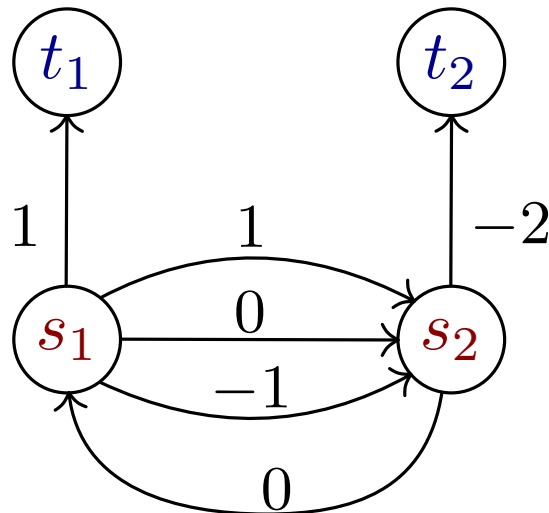
Consider the following specification:

input int t_1, t_2

```
output int       $s_1 := s_2[1, 0] + \text{if } s_2[-1, 7] \leq t_1[1, 0]$   
                                     then  $s_2[-1, 0]$   
                                     else  $s_2$ 
```

output int $s_2 := s_1 + t_2[-2, 1]$

The dependency graph is:



Well-formed specifications

def

A spec φ is *well-formed* if
its dependency graph has *no closed walks* of *zero weight*.

Well-formed specifications

def

A spec φ is *well-formed* if
its dependency graph has *no closed walks* of *zero weight*.

- How to check well-formedness?

Well-formed specifications

def

A spec φ is *well-formed* if
its dependency graph has *no closed walks* of *zero weight*.

► How to check well-formedness?

FACT: A graph has a *closed walk of zero-weight*
if and only if
some node n has both a *simple non-negative cycle*
and a *simple non-positive cycle*

Well-formed specifications

def

A spec φ is *well-formed* if its dependency graph has *no closed walks* of *zero weight*.

- How to check well-formedness?

FACT: A graph has a *closed walk of zero-weight* if and only if some node n has both a *simple non-negative cycle* and a *simple non-positive cycle*

- To decide well-formedness check, for every node, the existence of both non-negative cycles and non-positive cycles.

Well-formed specifications

def

A spec φ is *well-formed* if
its dependency graph has *no closed walks* of *zero weight*.

FACT: A graph has a *closed walk* of zero-weight
if and only if
some node n has both a *simple non-negative cycle*
and *a simple non-positive cycle*

Well-formed specifications

def

A spec φ is *well-formed* if its dependency graph has *no closed walks of zero weight*.

FACT: A graph has a *closed walk of zero-weight* if and only if

some node n has both a *simple non-negative cycle* and *a simple non-positive cycle*

FACT: Let G be dependency graph of a well-formed spec and let S be a strongly connected component of G . Then, either

- ▶ all the simple cycles in S are *strictly positive* or
- ▶ all the simple cycles in S are *strictly negative*

Evaluation graph

Given a specification φ and a length N ,
an *evaluation graph* $G_N : (V, E)$ is:

Evaluation graph

Given a specification φ and a length N ,
an *evaluation graph* $G_N : (V, E)$ is:

► **Nodes:**

► **Edges:**

Evaluation graph

Given a specification φ and a length N ,
an *evaluation graph* $G_N : (V, E)$ is:

► **Nodes:**

For each stream variable s and position $k = 1 \dots N$
there is a node s^k .

For each stream variable t and position $k = 1 \dots N$
there is a node t^k .

► **Edges:**

Evaluation graph

Given a specification φ and a length N ,
an *evaluation graph* $G_N : (V, E)$ is:

► **Nodes:**

For each stream variable s and position $k = 1 \dots N$
there is a node s^k .

For each stream variable t and position $k = 1 \dots N$
there is a node t^k .

► **Edges:** For every defining equation $s := e$

There is an edge $s^k \rightarrow u^k$ if u is a subterm of e .

There is an edge $s^k \rightarrow u^{k+j}$ if $u[j, d]$ occurs in e .

Evaluation graph (example)

Consider the specification:

input bool t

output bool $s = t \vee s[-1, \text{false}]$

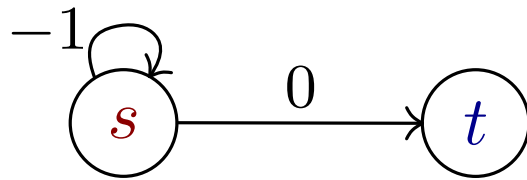
Evaluation graph (example)

Consider the specification:

input bool t

output bool $s = t \vee s[-1, \text{false}]$

The dependency graph is:



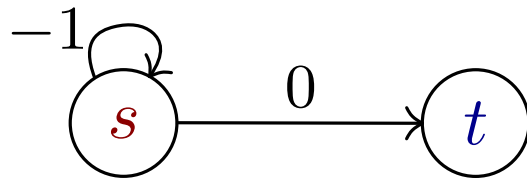
Evaluation graph (example)

Consider the specification:

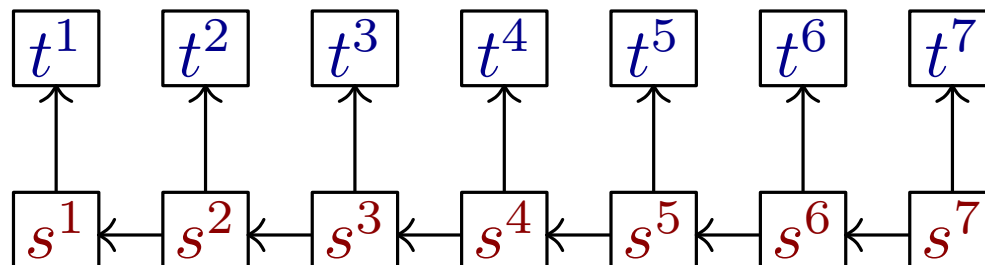
input bool t

output bool $s = t \vee s[-1, \text{false}]$

The dependency graph is:



For length 7 the evaluation graph is:



Evaluation graph and dependency graph

Lemma

Let φ be a specification.

Let G be its dependency graph and

let G_N be its evaluation graph for length N .

If G_N has a cycle then G has a zero-weight closed walk.

Evaluation graph and dependency graph

Lemma

Let φ be a specification.

Let G be its dependency graph and

let G_N be its evaluation graph for length N .

If G_N has a cycle then G has a zero-weight closed walk.

Proof: Follows from the following observation

A traverse from s^k to s^j in G_N corresponds to a walk of weight $k - j$ from s to itself in G .

Evaluation graph and dependency graph

Lemma

Let φ be a specification.

Let G be its dependency graph and

let G_N be its evaluation graph for length N .

If G_N has a cycle then G has a zero-weight closed walk.

Corollary

If G is well-formed, then for every N , G_N has no cycles.

Evaluation Graph and Evaluation Models

Lemma

Let φ be a spec and $\langle \tau_1, \dots, \tau_m \rangle$ a valuation of inputs of length N .

If G_N has no cycles, then φ has a *unique* evaluation model

$$\langle \tau_1, \dots, \tau_m, \sigma_1, \dots, \sigma_m \rangle \models \varphi$$

that extends $\langle \tau_1, \dots, \tau_m \rangle$.

Evaluation Graph and Evaluation Models

Lemma

Let φ be a spec and $\langle \tau_1, \dots, \tau_m \rangle$ a valuation of inputs of length N .

If G_N has no cycles, then φ has a *unique* evaluation model

$$\langle \tau_1, \dots, \tau_m, \sigma_1, \dots, \sigma_m \rangle \models \varphi$$

that extends $\langle \tau_1, \dots, \tau_m \rangle$.

Proof: Evaluate s^k in reverse topological order to obtain the only possible value.

Theorem

Every *well-formed* specification is *well-defined*

Wellformed and welldefined

Theorem

Every *well-formed* specification is *well-defined*

Wellformed and welldefined

Theorem

Every *well-formed* specification is *well-defined*

The converse is not true

Wellformed and welldefined

Theorem

Every *well-formed* specification is *well-defined*

The converse is not true

```
input  bool   $t$   
output bool   $s_1 := (s_2 \vee \neg s_2) \wedge t$   
output bool   $s_2 := s_1$ 
```

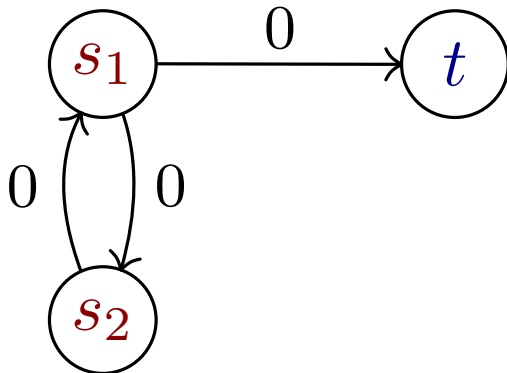
Wellformed and welldefined

Theorem

Every *well-formed* specification is *well-defined*

The converse is not true

input bool t
output bool $s_1 := (s_2 \vee \neg s_2) \wedge t$
output bool $s_2 := s_1$



Wellformed and welldefined

Theorem

Every *well-formed* specification is *well-defined*

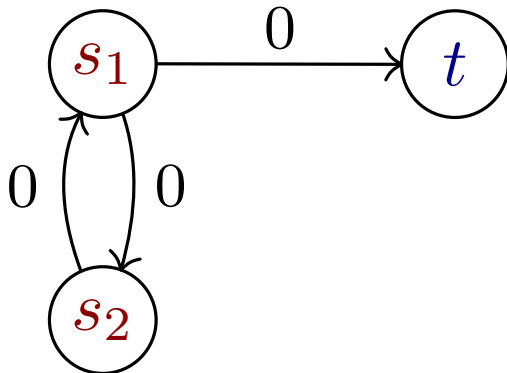
The converse is not true

input bool t

output bool $s_1 := (s_2 \vee \neg s_2) \wedge t$ ← Well-defined

output bool $s_2 := s_1$

$s_1 := t$



← Not well-formed

Operational Semantics

Online Runtime Verification

Operational semantics

The denotational semantics (well-definedness) only guarantee a single output per input.

... but *how* to compute this output?

Online algorithm

The algorithm will work on *position variables* s_i^k

At position k , the algorithm will instantiate

$s_i := e_i$ into $s_i^k = e_i^k$, where

- ▶ $c^k \rightarrow c$
- ▶ $f(a_1, \dots, a_j)^k = f(a_1^k, \dots, a_j^k)$
- ▶ $s_i[j, d]^k = \begin{cases} d & \text{if } j + k < 1 \\ s_i^{j+k} \Big|_d & \text{otherwise} \end{cases}$

Online algorithm

The algorithm maintains two storages:

- ▶ R : resolved equations $\{\dots t_j^k = c \dots s_i^k = c' \dots\}$

All position variables s_i^k with known values are in R .

- ▶ U : unresolved equations $\{\dots s_i^k = g \dots\}$

Position variables s_i^k whose values are not completely determined yet are in U .

Initially: R is empty, U is empty

Online algorithm

At every step k :

1. add $t_i^k = \tau_i^k$ to R for every input

2. add $s_i^k = e^k$ to U for every dependent stream s_i

3. Repeat

substitute $s_j^l \leftarrow c$ in every eq in U if $s_j^l = c \in R$,

apply functions $f^k(c_1, \dots, c_k)$ if all arguments are constants

apply simplifiers

if some eq in U becomes $s_i^l = c$ move to R

Until U does not change

Online algorithm

Can R be pruned?

Online algorithm

Can R be pruned?

def

The *back reference* distance of a node v in the dependency graph is:

$$\nabla v = \max(0, \{k | u \xrightarrow{-k} v\})$$

Online algorithm

Can R be pruned?

def

The *back reference* distance of a node v in the dependency graph is:

$$\nabla v = \max(0, \{k \mid u \xrightarrow{-k} v\})$$

FACT: A term $v^k = c$ can be removed from R at $k + \nabla v$

Why?

Online algorithm

Can R be pruned?

def

The *back reference* distance of a node v in the dependency graph is:

$$\nabla v = \max(0, \{k \mid u \xrightarrow{-k} v\})$$

FACT: A term $v^k = c$ can be removed from R at $k + \nabla v$

Why?

All equations that will ever need v^k are already in U at $k + \nabla v$

Online algorithm (example)

Consider the specification:

```
input  bool   $t_1, t_2$   
output bool   $s_1 := t_2 \vee (t_1 \wedge s_2)$   
output bool   $s_2 := s_1[1, \text{false}]$ 
```

and the input

| | 1 | 2 | 3 | 4 |
|--|---|---|---|---|
|--|---|---|---|---|

| | | | | |
|------------|---|---|---|---|
| $\tau_1 :$ | F | F | F | F |
|------------|---|---|---|---|

| | | | | |
|------------|---|---|---|---|
| $\tau_2 :$ | T | F | T | F |
|------------|---|---|---|---|

Online algorithm (example)

Consider the specification:

```
input  bool   $t_1, t_2$ 
output bool   $s_1 := t_2 \vee (t_1 \wedge s_2)$ 
output bool   $s_2 := s_1[1, \text{false}]$ 
```

and the input

| | 1 | 2 | 3 | 4 |
|--|---|---|---|---|
|--|---|---|---|---|

| | | | | |
|------------|---|---|---|---|
| $\tau_1 :$ | F | F | F | F |
|------------|---|---|---|---|

| | | | | |
|------------|---|---|---|---|
| $\tau_2 :$ | T | F | T | F |
|------------|---|---|---|---|

After each step, R and U are (before pruning):

R

U

Online algorithm (example)

Consider the specification:

input bool t_1, t_2

output bool $s_1 := t_2 \vee (t_1 \wedge s_2)$

output bool $s_2 := s_1[1, \text{false}]$

and the input

| | 1 | 2 | 3 | 4 |
|--|---|---|---|---|
|--|---|---|---|---|

| | | | | |
|------------|---|---|---|---|
| $\tau_1 :$ | F | F | F | F |
|------------|---|---|---|---|

| | | | | |
|------------|---|---|---|---|
| $\tau_2 :$ | T | F | T | F |
|------------|---|---|---|---|

After each step, R and U are (before pruning):

| | |
|-----|--------------------|
| 1 | |
| R | $t_1^1 = \text{F}$ |
| | $t_2^1 = \text{T}$ |
| | $s_1^1 = \text{T}$ |
| U | $s_2^1 := s_2^2$ |

Online algorithm (example)

Consider the specification:

input bool t_1, t_2

output bool $s_1 := t_2 \vee (t_1 \wedge s_2)$

output bool $s_2 := s_1[1, \text{false}]$

and the input

τ_1 : F F F F

τ_2 : T F T F

After each step, R and U are (before pruning):

| | 1 | 2 |
|-----|--|---|
| R | $t_1^1 = \text{F}$ $t_2^1 = \text{T}$ $s_1^1 = \text{T}$ | $t_1^2 = \text{F}$ $t_2^2 = \text{F}$ $s_1^2 = \text{F}$ $s_2^1 = \text{F}$ |
| U | $s_2^1 := s_2^2$ | $s_2^2 := s_1^3$ |

Online algorithm (example)

Consider the specification:

```
input  bool   $t_1, t_2$ 
output bool   $s_1 := t_2 \vee (t_1 \wedge s_2)$ 
output bool   $s_2 := s_1[1, \text{false}]$ 
```

and the input

τ_1 : F F F F

τ_2 : T F T F

After each step, R and U are (before pruning):

| | 1 | 2 | 3 |
|-----|---|---|---|
| R | $t_1^1 = F$ $t_2^1 = T$ $s_1^1 = T$ | $t_1^2 = F$ $t_2^2 = F$ $s_1^2 = F$ $s_2^1 = F$ | $t_1^3 = F$ $t_2^3 = T$ |
| U | $s_2^1 := s_2^2$ | $s_2^2 := s_1^3$ | $s_1^3 := s_2^3$ $s_2^3 := s_1^4$ $s_2^2 := s_1^3$ |

Online algorithm (example)

Consider the specification:

input bool t_1, t_2

output bool $s_1 := t_2 \vee (t_1 \wedge s_2)$

output bool $s_2 := s_1[1, \text{false}]$

and the input

τ_1 : F F F F

τ_2 : T F T F

After each step, R and U are (before pruning):

| | 1 | 2 | 3 | 4 |
|-----|--|---|---|---|
| R | $t_1^1 = \text{F}$ $t_2^1 = \text{T}$ $s_1^1 = \text{T}$ | $t_1^2 = \text{F}$ $t_2^2 = \text{F}$ $s_1^2 = \text{F}$ $s_2^1 = \text{F}$ | $t_1^3 = \text{F}$ $t_2^3 = \text{T}$ | $t_1^4 = \text{F}$ $s_2^3 = \text{F}$ $t_2^4 = \text{F}$ $s_1^3 = \text{F}$ $s_2^4 = \text{F}$ $s_2^2 = \text{F}$ $s_1^4 = \text{F}$ |
| U | $s_2^1 := s_2^2$ | $s_2^2 := s_1^3$ | $s_1^3 := s_2^3$ $s_2^3 := s_1^4$ $s_2^2 := s_1^3$ | |

Online algorithm (memory)

What is the *worst case memory* requirement?

Online algorithm (memory)

What is the *worst case memory* requirement?

```
input  int     $t$ 
output bool    $s_1 := \text{false}$ 
output bool    $s_2 := s_1[1, \text{true}]$ 
output int     $s_3 := s_4[1, 0]$ 
output int     $s_4 := \text{if } s_2 \text{ then } t \text{ else } s_3$ 
```

Consider the input $\tau : \langle 37, 39, 79, 17, 14 \rangle$

Online algorithm (memory)

What is the *worst case memory* requirement?

```
input  int     $t$ 
output bool     $s_1 := \text{false}$ 
output bool     $s_2 := s_1[1, \text{true}]$ 
output int      $s_3 := s_4[1, 0]$ 
output int      $s_4 := \text{if } s_2 \text{ then } t \text{ else } s_3$ 
```

Consider the input $\tau : \langle 37, 39, 79, 17, 14 \rangle$

1 2 3 4 5

τ

σ_1

σ_2

σ_3

σ_4

Online algorithm (memory)

What is the *worst case memory* requirement?

```
input  int     $t$ 
output bool     $s_1 := \text{false}$ 
output bool     $s_2 := s_1[1, \text{true}]$ 
output int      $s_3 := s_4[1, 0]$ 
output int      $s_4 := \text{if } s_2 \text{ then } t \text{ else } s_3$ 
```

Consider the input $\tau : \langle 37, 39, 79, 17, 14 \rangle$

| | 1 | 2 | 3 | 4 | 5 |
|------------|----|---|---|---|---|
| τ | 37 | | | | |
| σ_1 | F | | | | |
| σ_2 | | | | | |
| σ_3 | | | | | |
| σ_4 | | | | | |

Online algorithm (memory)

What is the *worst case memory* requirement?

```
input  int     $t$ 
output bool     $s_1 := \text{false}$ 
output bool     $s_2 := s_1[1, \text{true}]$ 
output int      $s_3 := s_4[1, 0]$ 
output int      $s_4 := \text{if } s_2 \text{ then } t \text{ else } s_3$ 
```

Consider the input $\tau : \langle 37, 39, 79, 17, 14 \rangle$

| | 1 | 2 | 3 | 4 | 5 |
|------------|----|----|---|---|---|
| τ | 37 | 39 | | | |
| σ_1 | F | F | | | |
| σ_2 | F | | | | |
| σ_3 | | | | | |
| σ_4 | | | | | |

Online algorithm (memory)

What is the *worst case memory* requirement?

```
input  int     $t$ 
output bool     $s_1 := \text{false}$ 
output bool     $s_2 := s_1[1, \text{true}]$ 
output int      $s_3 := s_4[1, 0]$ 
output int      $s_4 := \text{if } s_2 \text{ then } t \text{ else } s_3$ 
```

Consider the input $\tau : \langle 37, 39, 79, 17, 14 \rangle$

| | 1 | 2 | 3 | 4 | 5 |
|------------|----|----|----|---|---|
| τ | 37 | 39 | 79 | | |
| σ_1 | F | F | F | | |
| σ_2 | F | F | | | |
| σ_3 | | | | | |
| σ_4 | | | | | |

Online algorithm (memory)

What is the *worst case memory* requirement?

```
input  int     $t$ 
output bool     $s_1 := \text{false}$ 
output bool     $s_2 := s_1[1, \text{true}]$ 
output int      $s_3 := s_4[1, 0]$ 
output int      $s_4 := \text{if } s_2 \text{ then } t \text{ else } s_3$ 
```

Consider the input $\tau : \langle 37, 39, 79, 17, 14 \rangle$

| | 1 | 2 | 3 | 4 | 5 |
|------------|----|----|----|----|---|
| τ | 37 | 39 | 79 | 17 | |
| σ_1 | F | F | F | F | |
| σ_2 | F | F | F | | |
| σ_3 | | | | | |
| σ_4 | | | | | |

Online algorithm (memory)

What is the *worst case memory* requirement?

```
input  int     $t$ 
output bool     $s_1 := \text{false}$ 
output bool     $s_2 := s_1[1, \text{true}]$ 
output int      $s_3 := s_4[1, 0]$ 
output int      $s_4 := \text{if } s_2 \text{ then } t \text{ else } s_3$ 
```

Consider the input $\tau : \langle 37, 39, 79, 17, 14 \rangle$

| | 1 | 2 | 3 | 4 | 5 |
|------------|----|----|----|----|----|
| τ | 37 | 39 | 79 | 17 | 14 |
| σ_1 | F | F | F | F | F |
| σ_2 | F | F | F | F | T |
| σ_3 | 14 | 14 | 14 | 14 | 0 |
| σ_4 | 14 | 14 | 14 | 14 | 14 |

Online algorithm (memory)

What is the *worst case memory* requirement?

```
input  int     $t$ 
output bool    $s_1 := \text{false}$ 
output bool    $s_2 := s_1[1, \text{true}]$ 
output int     $s_3 := s_4[1, 0]$ 
output int     $s_4 := \text{if } s_2 \text{ then } t \text{ else } s_3$ 
```

Consider

Memory required is *linear* in the size of the *trace*

| τ | 37 | 39 | 79 | 17 | 14 |
|------------|----|----|----|----|----|
| σ_1 | F | F | F | F | F |
| σ_2 | F | F | F | F | T |
| σ_3 | 14 | 14 | 14 | 14 | 0 |
| σ_4 | 14 | 14 | 14 | 14 | 14 |

Online algorithm (memory)

def

Let G_N be an evaluation graph

the *fan* of a variable is the set of nodes it (may) depend on

$$Fan(s^k) = \{v^j \mid s^k \rightarrow^* v^j\}$$

the *latency* is the farthest distance to a node in the fan:

$$Lat(s^k) = \max(0, \{j \mid v^j \in Fan(s^k)\})$$

Online algorithm (memory)

def

Let G_N be an evaluation graph

the *fan* of a variable is the set of nodes it (may) depend on

$$Fan(s^k) = \{v^j \mid s^k \rightarrow^* v^j\}$$

the *latency* is the farthest distance to a node in the fan:

$$Lat(s^k) = \max(0, \{j \mid v^j \in Fan(s^k)\})$$

Theorem

Let G_N be an evaluation graph, if s^k has $Lat(s^k) = j$, then the online algorithm resolves s^k at $k + j$ or earlier.

Efficient monitorability

Goal: capture specifications that only require *bounded memory*

Efficient monitorability

Goal: capture specifications that only require *bounded memory*

def

A specification is *efficiently monitorable* if the worst case memory requirement is independent on N

Efficient monitorability (example)

Example: *every request is followed by a grant
before the trace ends*

Efficient monitorability (example)

Example: *every request is followed by a grant
before the trace ends*

input bool *request, grant*

output bool *reqgrant* := if *request* then *evgrant* else true

output bool *evgrant* := *grant* \vee *evgrant*[1, false]

trigger (\neg *reqgrant*)

Efficient monitorability (example)

Example: *every request is followed by a grant before the trace ends*

```
input  bool  request, grant
output bool  reqgrant := if request then evgrant else true
output bool  evgrant := grant  $\vee$  evgrant[1, false]
trigger      ( $\neg$ reqgrant)
```

This is *not* efficiently monitorable

Efficient monitorability (example)

Example: *every request is followed by a grant before the trace ends*

```
input  bool  request, grant
output bool  reqgrant := if request then evgrant else true
output bool  evgrant := grant  $\vee$  evgrant[1, false]
trigger      ( $\neg$ reqgrant)
```

This is *not* efficiently monitorable

```
input  bool  request, grant
output bool  wait :=  $\neg$ grant  $\wedge$  (request  $\vee$  wait[-1, false])
output bool  ended := false[1, true]
trigger      ended  $\wedge$  wait
```

Efficient monitorability (example)

Example: *every request is followed by a grant before the trace ends*

```
input  bool  request, grant
output bool  reqgrant := if request then evgrant else true
output bool  evgrant := grant  $\vee$  evgrant[1, false]
trigger      ( $\neg$ reqgrant)
```

This is *not* efficiently monitorable

```
input  bool  request, grant
output bool  wait :=  $\neg$ grant  $\wedge$  (request  $\vee$  wait[-1, false])
output bool  ended := false[1, true]
trigger      ended  $\wedge$  wait
```

This *is* efficiently monitorable

Efficiently Monitorable

def

A specification is *future bounded* if G has no positive-weight cycles

The *lookahead* Δs of a node s is the maximum positive weight of a walk from s

FACT: Let G be a dependency graph of a FB spec, and G_N the evaluation graph for some N . Then $Lat(s^k) \leq \Delta s$.

Theorem

Every future bounded specification is efficiently monitorable

FACT: The number of equations stored in U and R is *linear* in the spec and in Δs .

Very Efficient Monitorable

def

A well-formed specification is *very efficiently monitorable* if it only uses zero or *negative* shift

Very Efficient Monitorable

def

A well-formed specification is *very efficiently monitorable* if it only uses zero or *negative* shift

For a very efficiently monitorable specification:

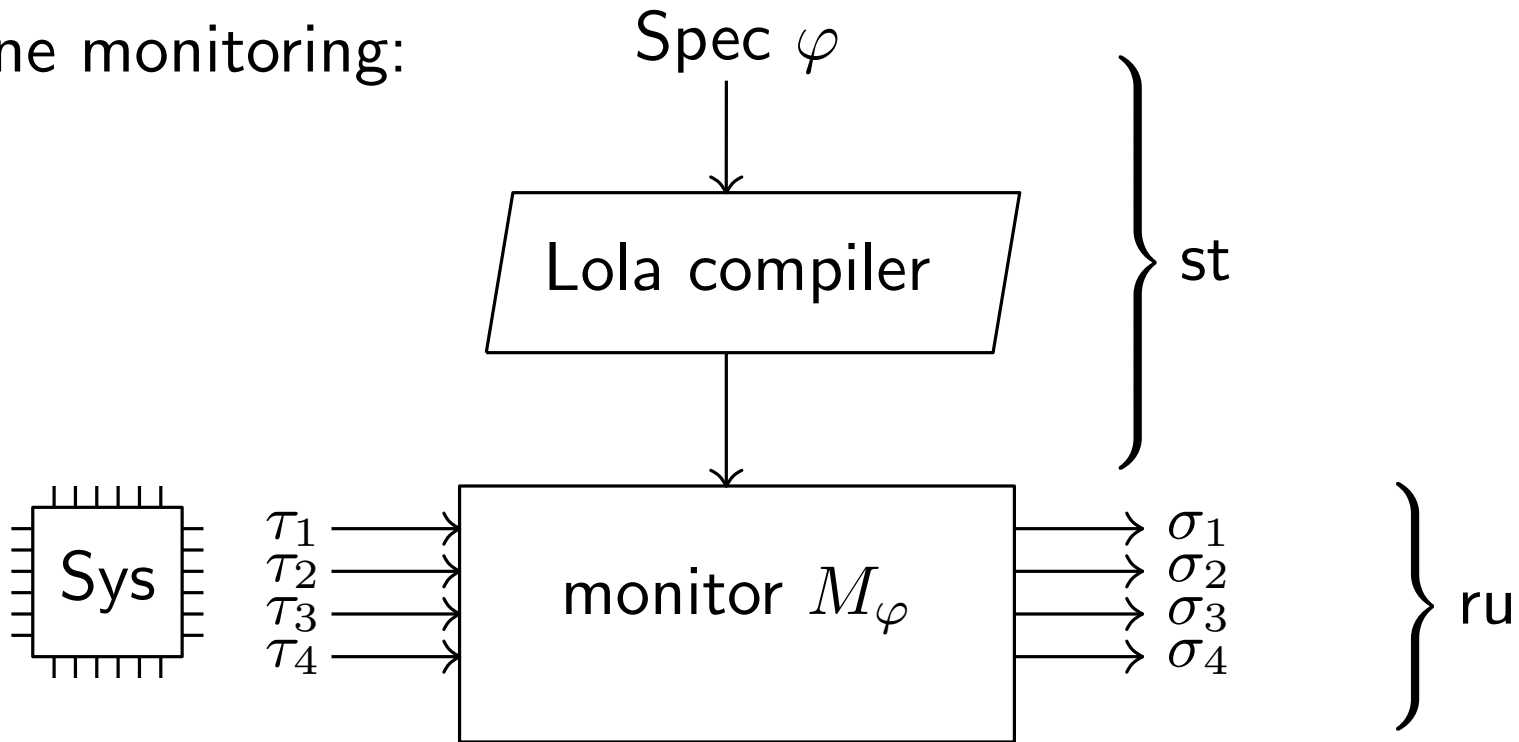
- ▶ The lookahead of every s is 0.
- ▶ Every s^k is resolved *immediately*
- ▶ The memory required is *linear* in the size of the spec

Operational Semantics

Offline Runtime Verification

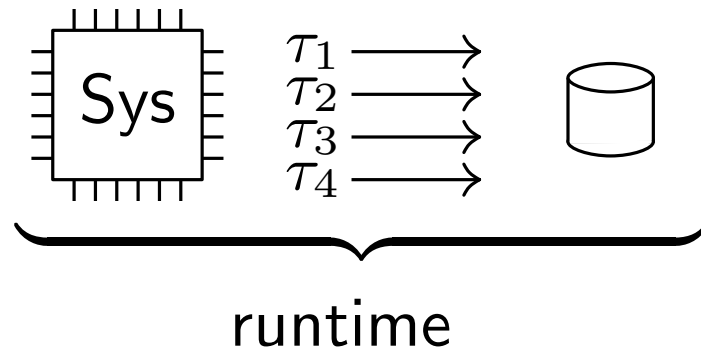
Offline monitoring

Online monitoring:



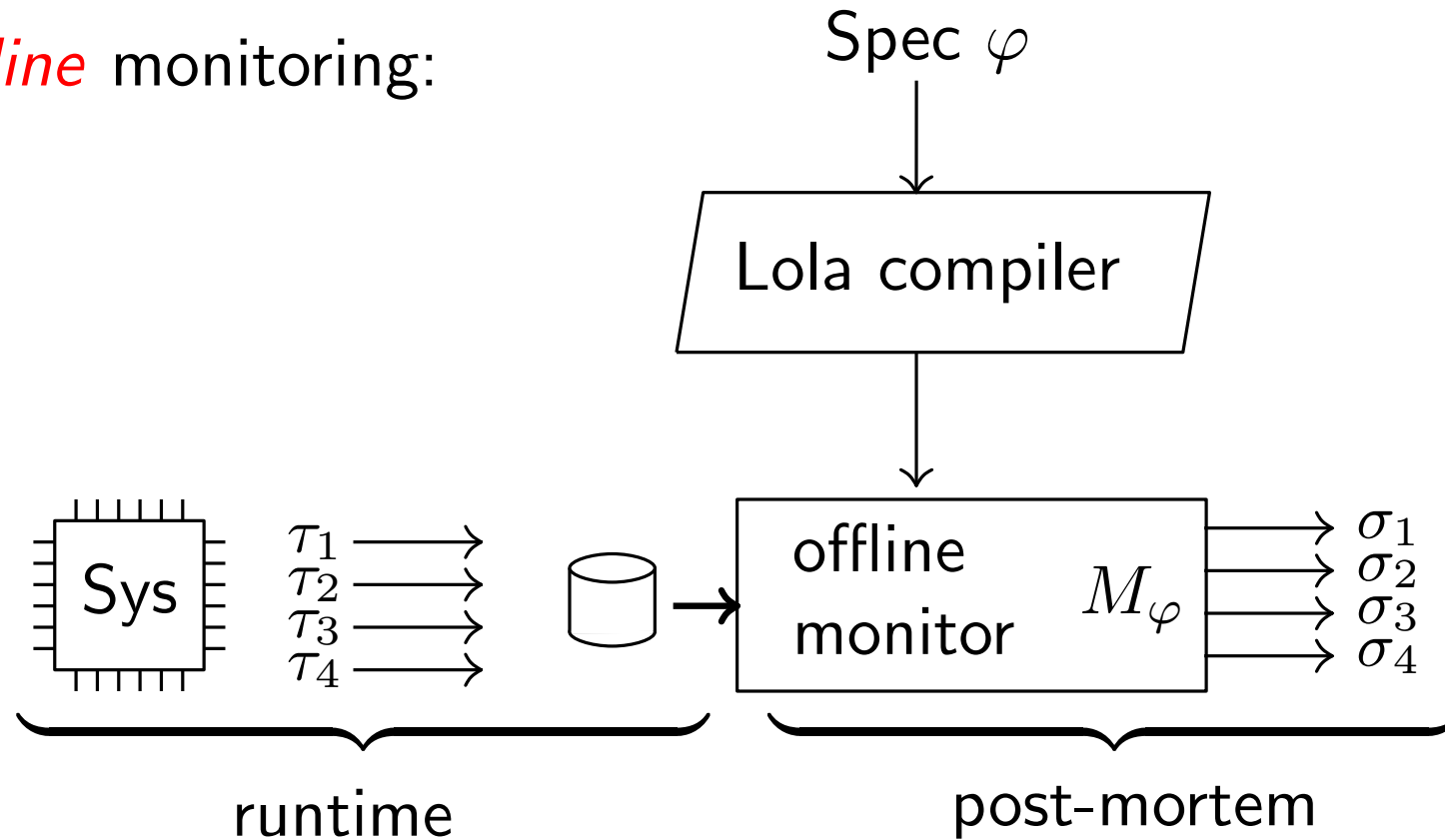
Offline monitoring

Offline monitoring:



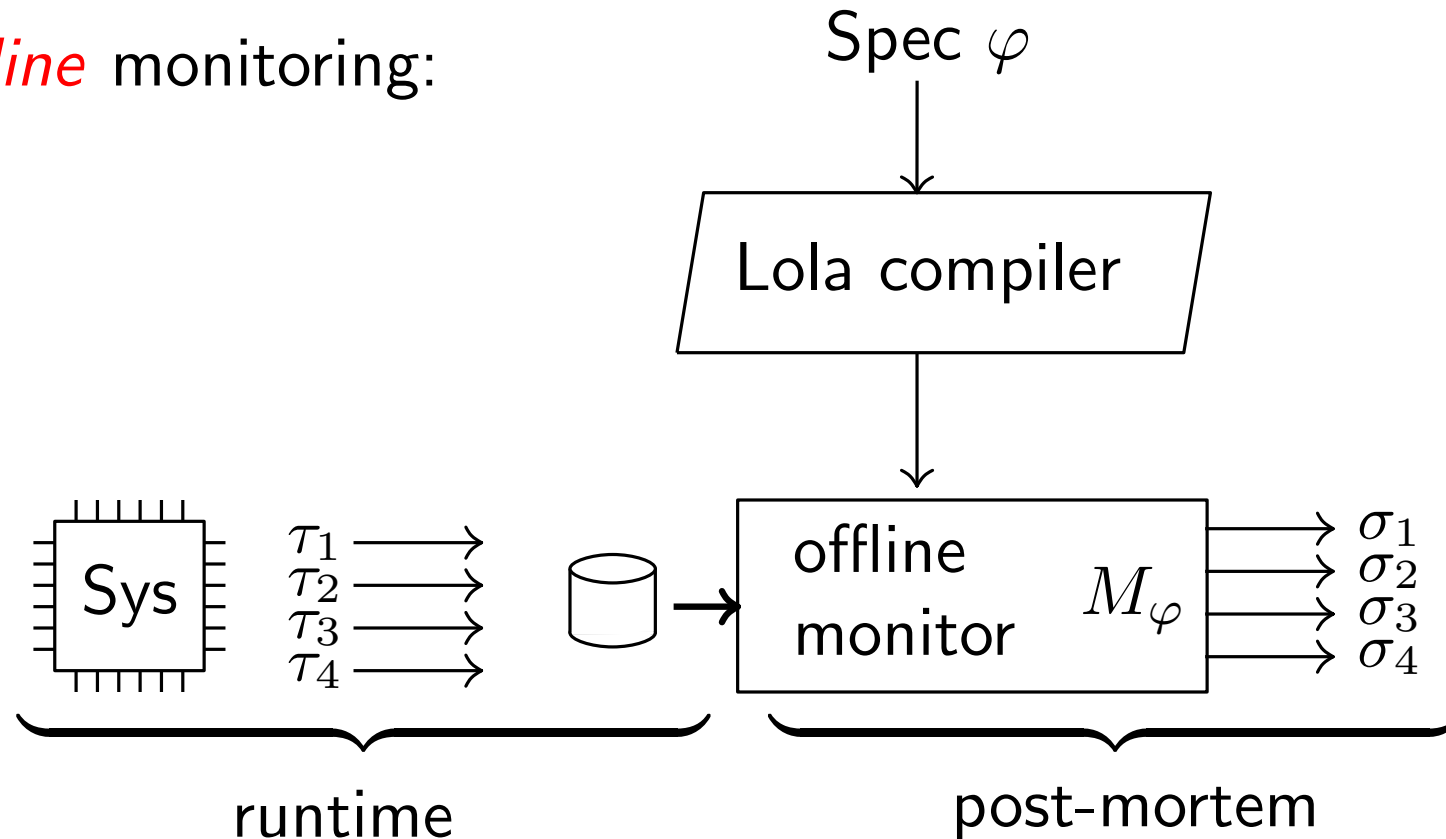
Offline monitoring

Offline monitoring:



Offline monitoring

Offline monitoring:

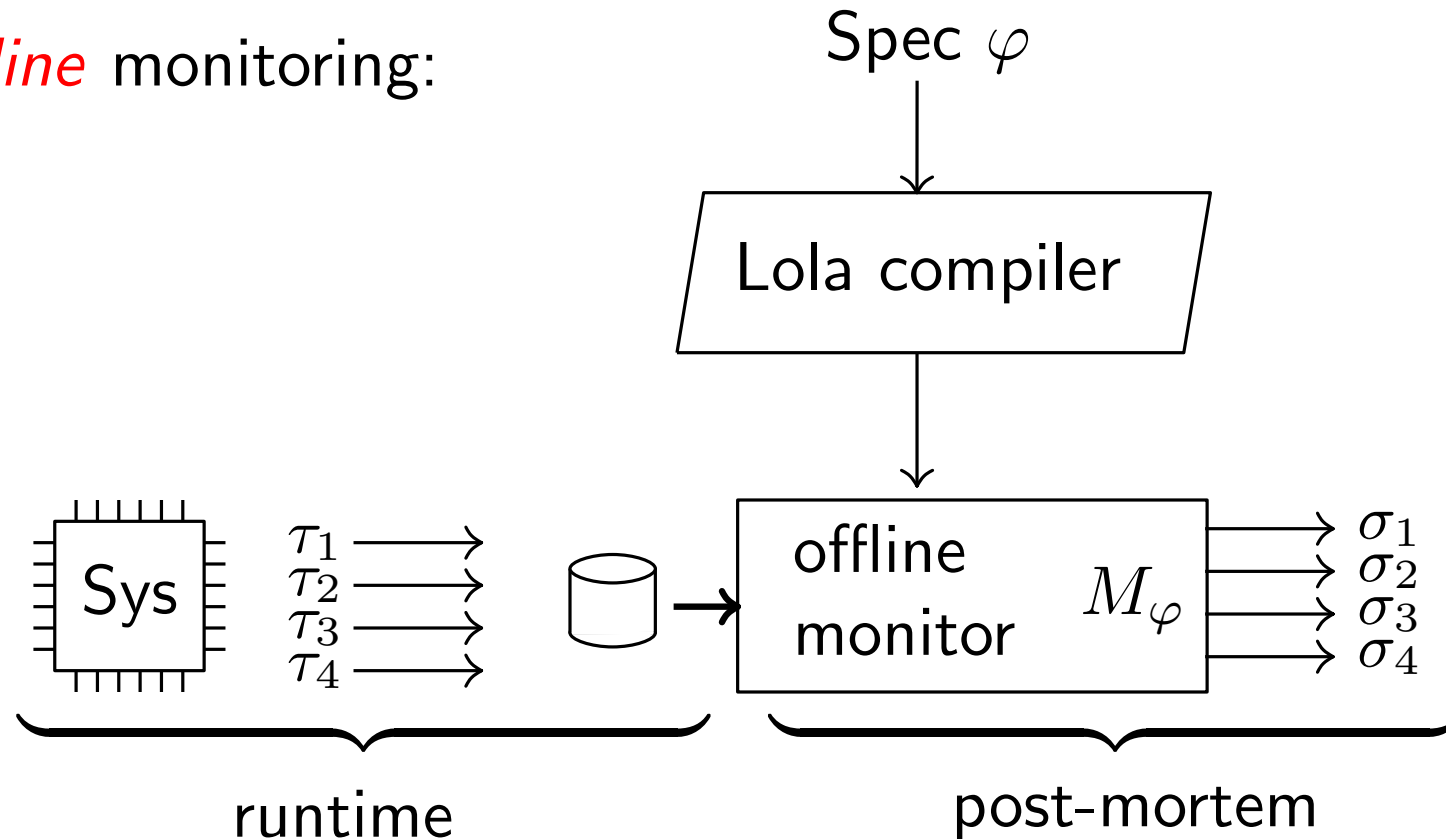


Advantages: monitors can

- ▶ use clairvoyance
- ▶ schedule passes

Offline monitoring

Offline monitoring:



Advantages: monitors can

- ▶ use clairvoyance
- ▶ schedule passes

Challenges: how to

- ▶ evaluate efficiently richer specs
- ▶ handle huge traces

Offline monitoring

Goal: algorithms that can schedule passes that only require *bounded memory*

Offline monitoring

Goal: algorithms that can schedule passes that only require *bounded memory*

def

A specification is *reverse efficiently monitorable* if the worst case memory requirement when applying the online algorithm to the *reverse trace* is independent of N

Reverse efficient monitorability (example)

Example: *every request is followed by a grant
before the trace ends*

Reverse efficient monitorability (example)

Example: *every request is followed by a grant
before the trace ends*

```
input  bool  request, grant
output bool  reqgrant := if request then evgrant else true
output bool  evgrant := grant  $\vee$  evgrant[1, false]
trigger      ( $\neg$ reqgrant)
```

Reverse efficient monitorability (example)

Example: *every request is followed by a grant
before the trace ends*

```
input  bool  request, grant
output bool  reqgrant := if request then evgrant else true
output bool  evgrant := grant  $\vee$  evgrant[1, false]
trigger      ( $\neg$ reqgrant)
```

This *is* reverse efficiently monitorable

Reverse efficient monitorability (example)

Example: *every request is followed by a grant
before the trace ends*

```
input  bool  request, grant
output bool  reqgrant := if request then evgrant else true
output bool  evgrant := grant  $\vee$  evgrant[1, false]
trigger      ( $\neg$ reqgrant)
```

This *is* reverse efficiently monitorable

```
input  bool  request, grant
output bool  wait :=  $\neg$ grant  $\wedge$  (request  $\vee$  wait[-1, false])
output bool  ended := false[1, true]
trigger      ended  $\wedge$  wait
```

Reverse efficient monitorability (example)

Example: *every request is followed by a grant
before the trace ends*

```
input  bool  request, grant
output bool  reqgrant := if request then evgrant else true
output bool  evgrant := grant  $\vee$  evgrant[1, false]
trigger      ( $\neg$ reqgrant)
```

This *is* reverse efficiently monitorable

```
input  bool  request, grant
output bool  wait :=  $\neg$ grant  $\wedge$  (request  $\vee$  wait[-1, false])
output bool  ended := false[1, true]
trigger      ended  $\wedge$  wait
```

This is *not* reverse efficiently monitorable

Reverse Efficiently Monitorable

def

A specification is *past bounded*
if G has no positive-weight cycles

Theorem

Every past bounded specification is reverse efficiently monitorable

Partition Graph

Consider a **well-formed** specification φ

Partition the **dependency graph** G into its maximally strongly connected component (MSCCs).

Partition Graph

Consider a **well-formed** specification φ

Partition the **dependency graph** G into its maximally strongly connected component (MSCCs).

Note: a MSCC is $U \subset V$ such that:

- ▶ for all $a, b \in U$, $a \rightarrow^* b$ and $b \rightarrow^* a$
- ▶ for every $v \notin U$, either $v \not\rightarrow^* U$ or $U \not\rightarrow^* v$.

Partition Graph

Consider a **well-formed** specification φ

Partition the **dependency graph** G into its maximally strongly connected component (MSCCs).

Note: a MSCC is $U \subset V$ such that:

- ▶ for all $a, b \in U$, $a \rightarrow^* b$ and $b \rightarrow^* a$
- ▶ for every $v \notin U$, either $v \not\rightarrow^* U$ or $U \not\rightarrow^* v$.

The **partition-graph** G_M is:

- ▶ **Nodes:** MSCCs from G
- ▶ **Edges:** $N \rightarrow M$ if for some $n \in N$ and $m \in M$, $n \rightarrow m$
- ▶ An MSCC N is **positive** if all its closed walks are positive
- ▶ An MSCC N is **negative** if all its closed walks are negative

Partition Graph

Consider a **well-formed** specification φ

Partition the **dependency graph** G into its maximally strongly connected component (MSCCs).

Note: a MSCC is $U \subset V$ such that:

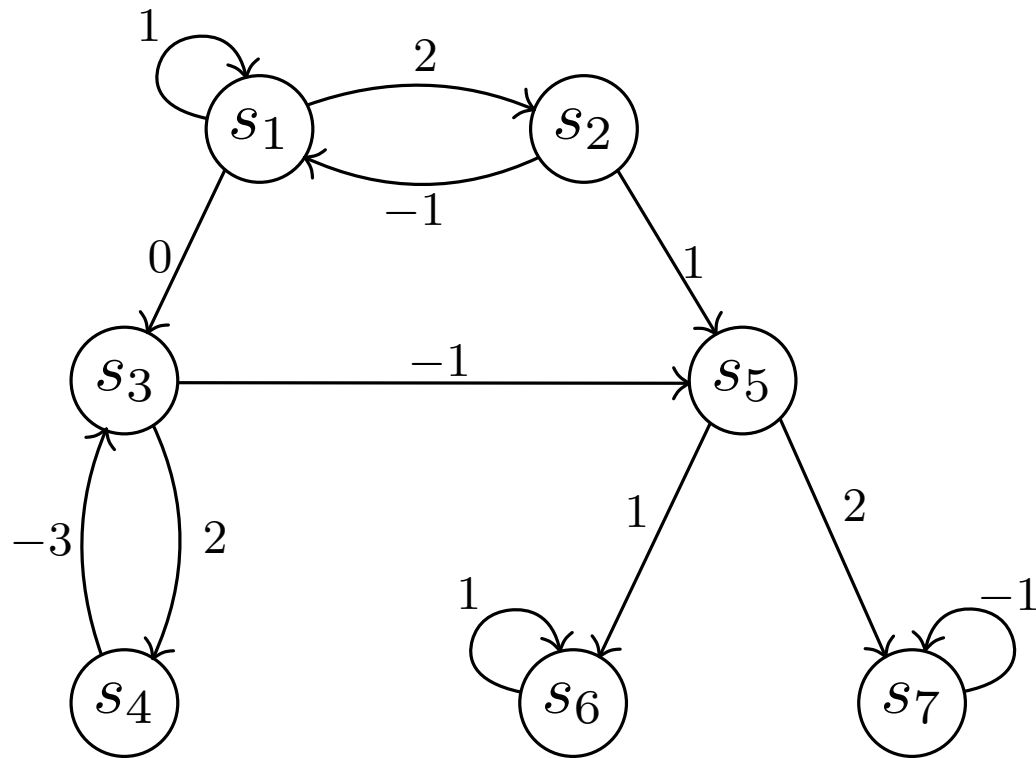
- ▶ for all $a, b \in U$, $a \rightarrow^* b$ and $b \rightarrow^* a$
- ▶ for every $v \notin U$, either $v \not\rightarrow^* U$ or $U \not\rightarrow^* v$.

The **partition-graph** G_M is:

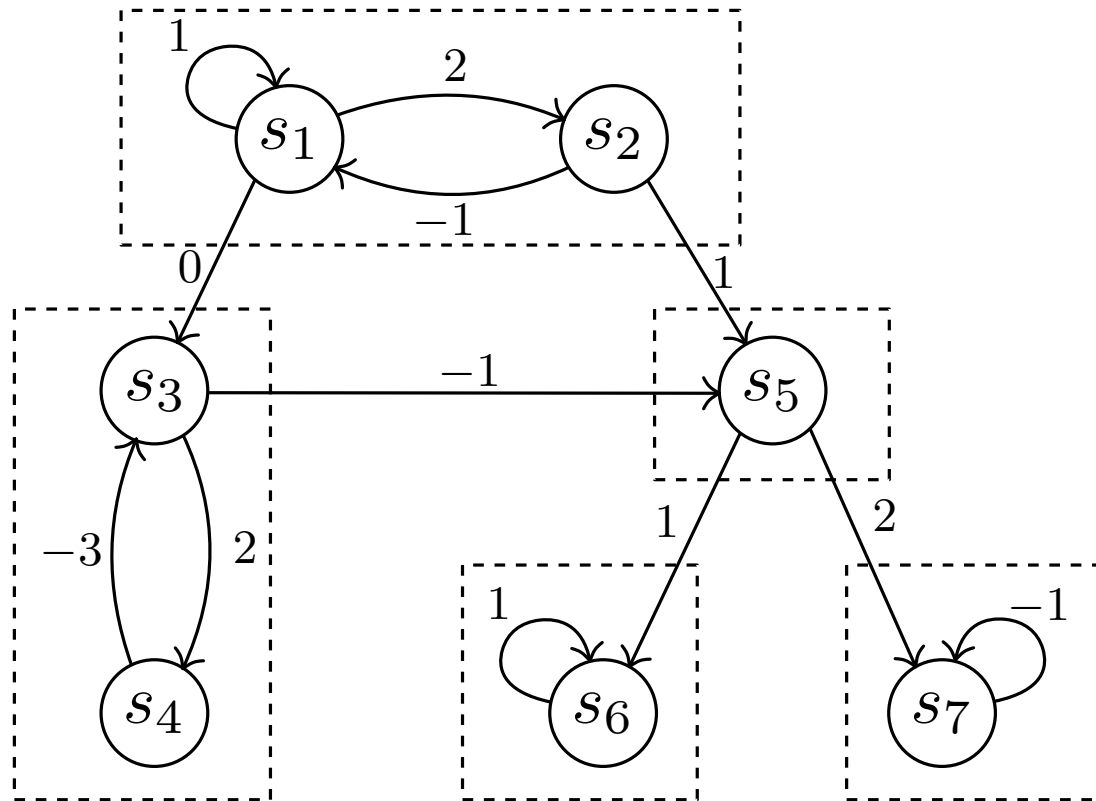
- ▶ **Nodes:** MSCCs from G
- ▶ **Edges:** $N \rightarrow M$ if for some $n \in N$ and $m \in M$, $n \rightarrow m$
- ▶ An MSCC N is **positive** if all its closed walks are positive
- ▶ An MSCC N is **negative** if all its closed walks are negative

FACT: G_M is acyclic

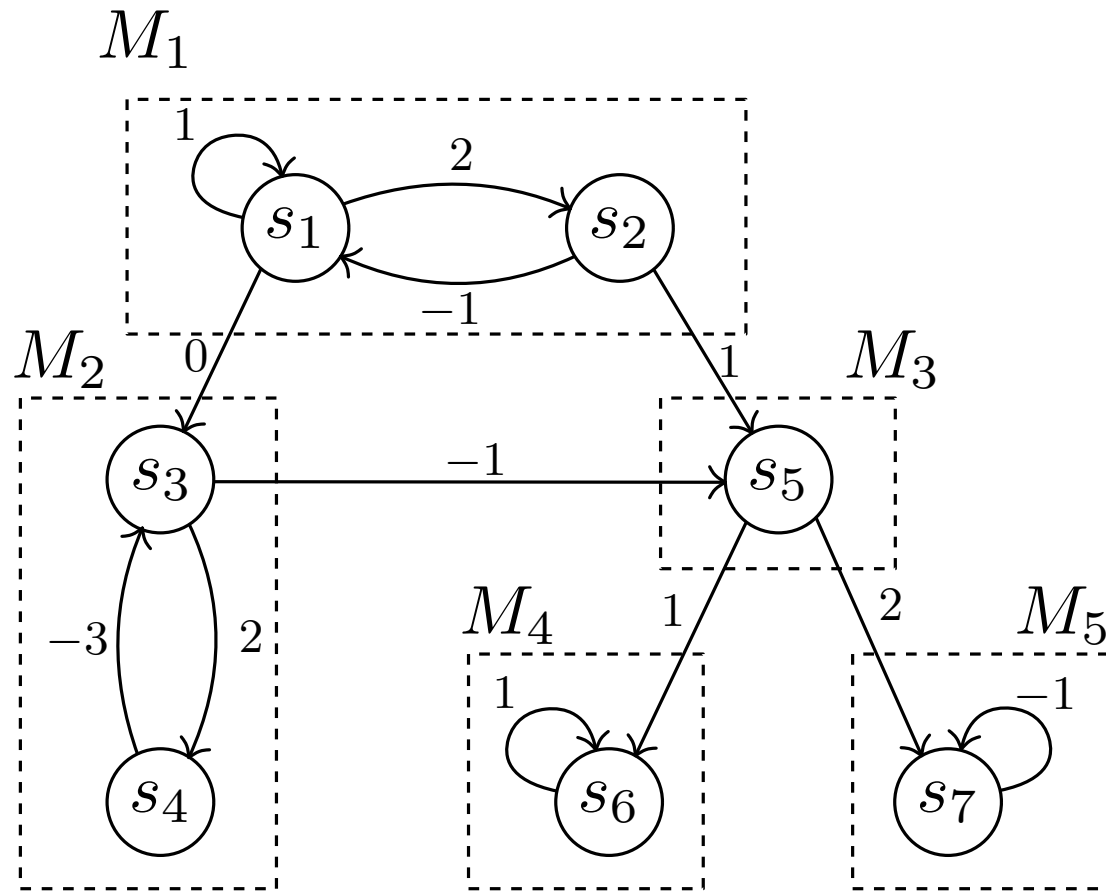
Partition Graph (example)



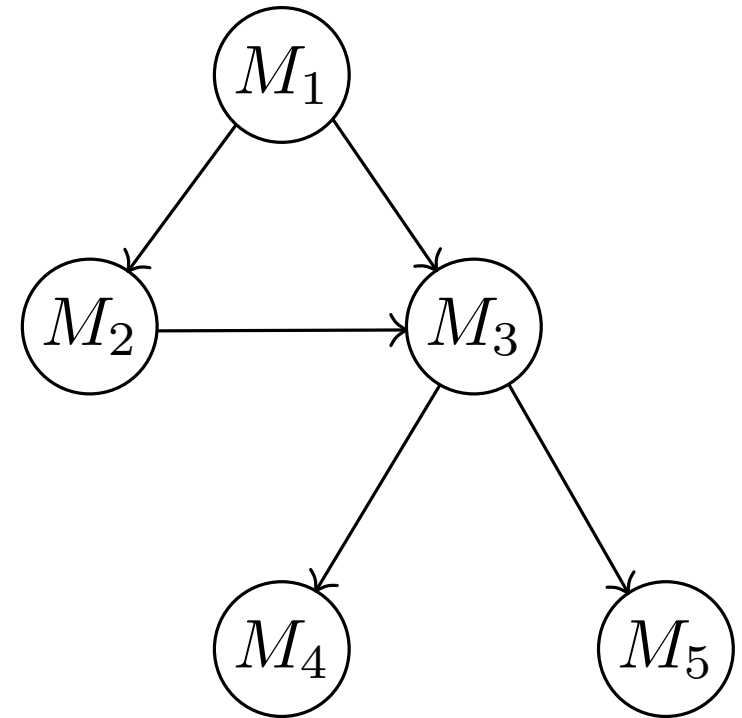
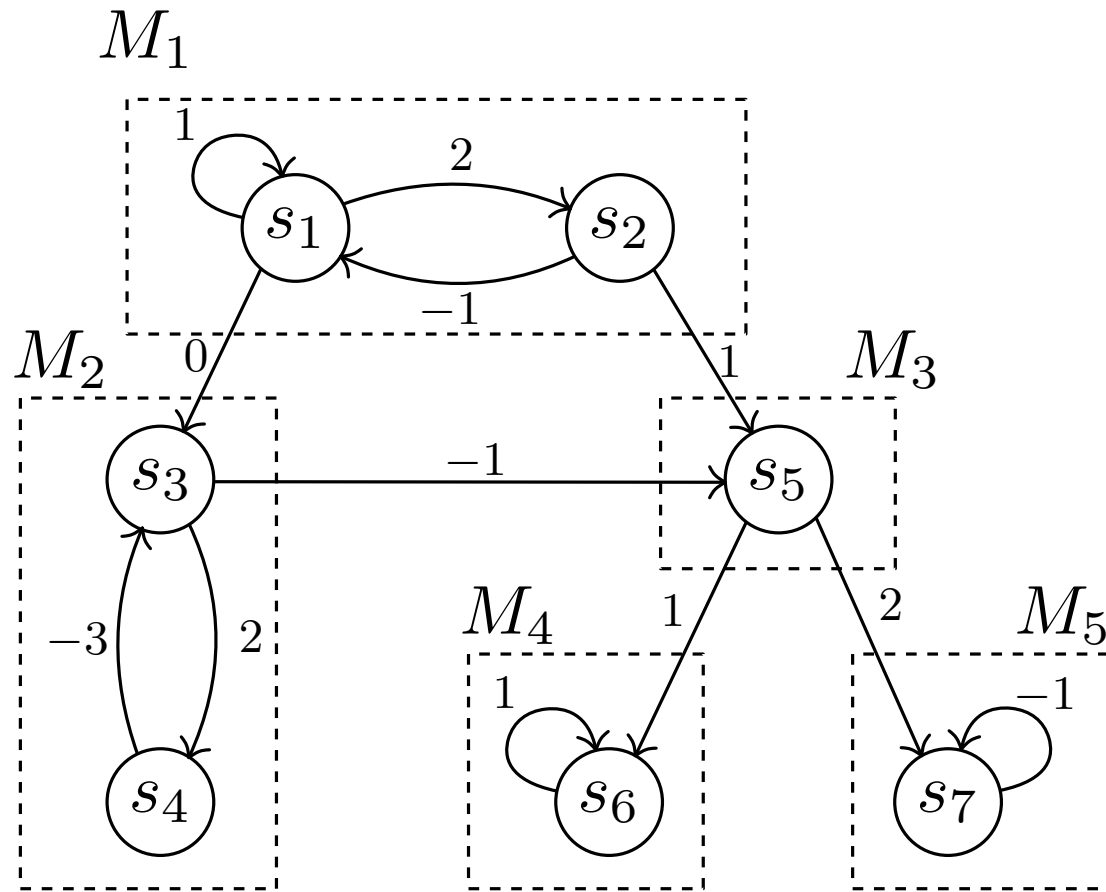
Partition Graph (example)



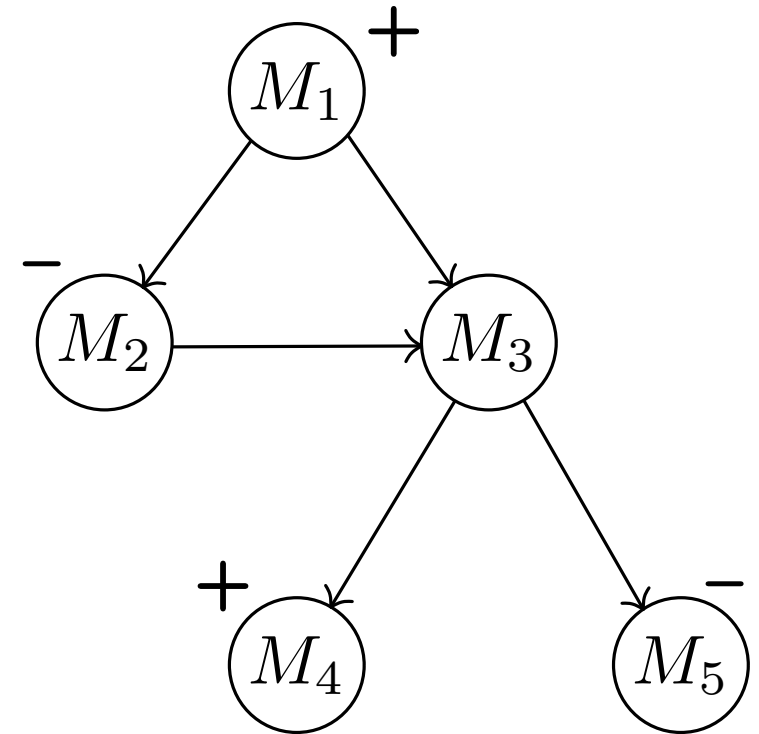
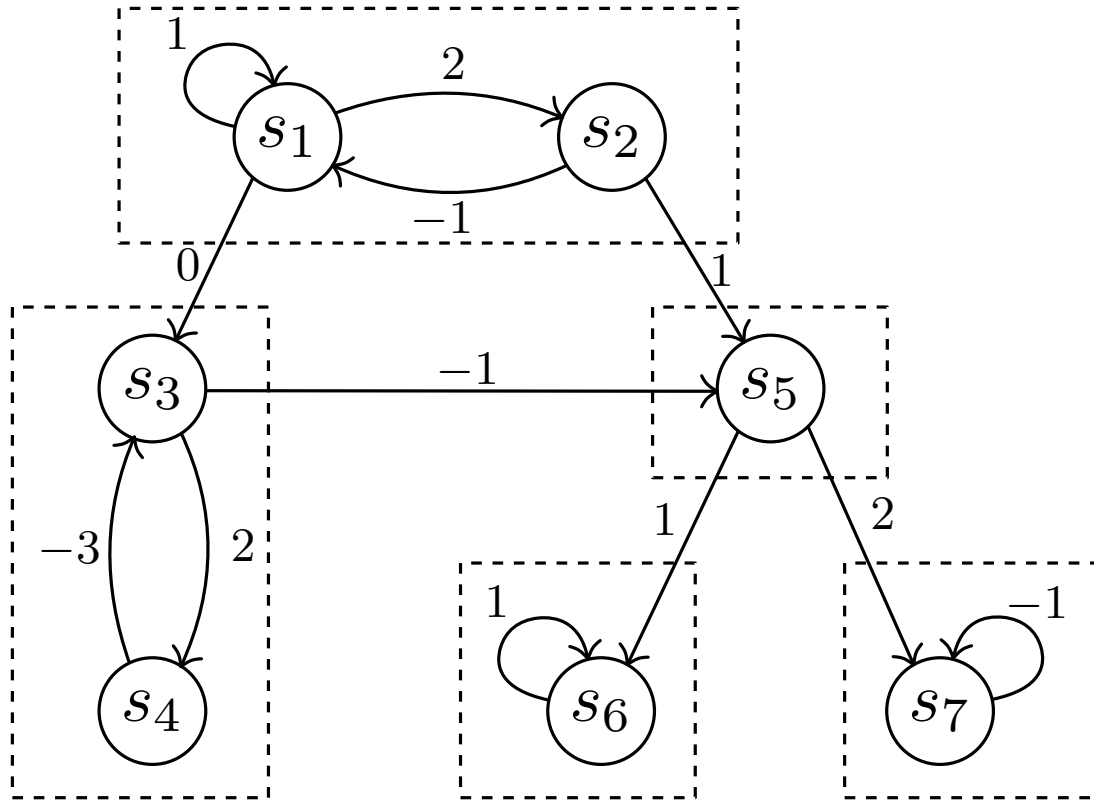
Partition Graph (example)



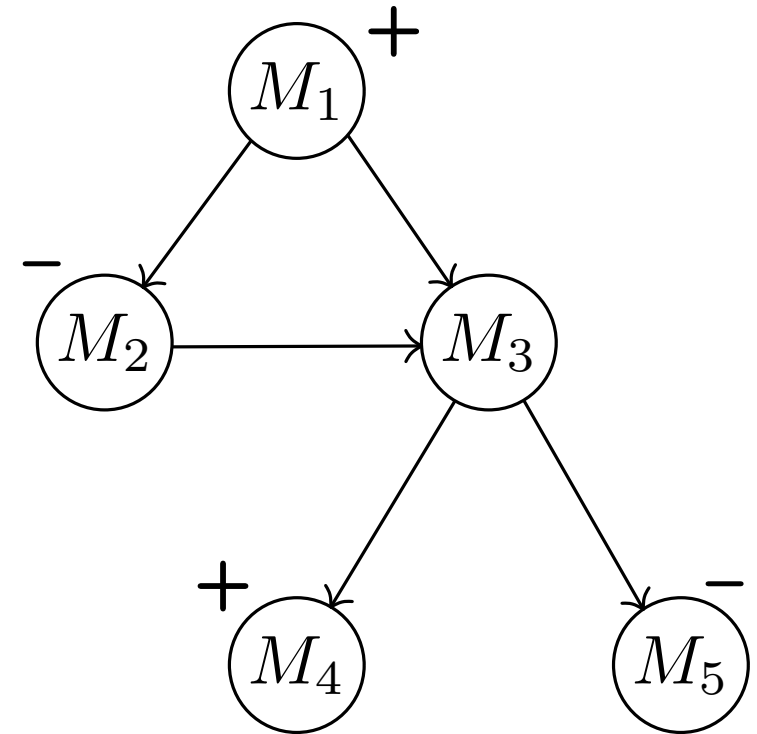
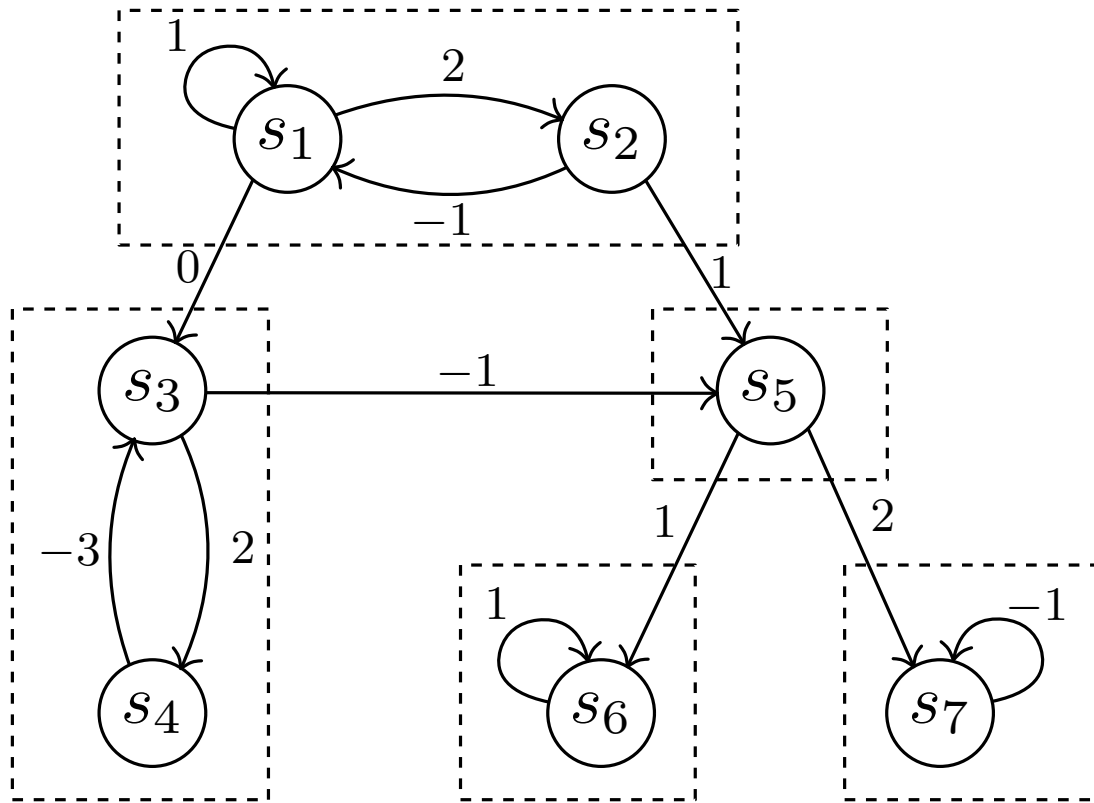
Partition Graph (example)



Partition Graph (example)



Partition Graph (example)



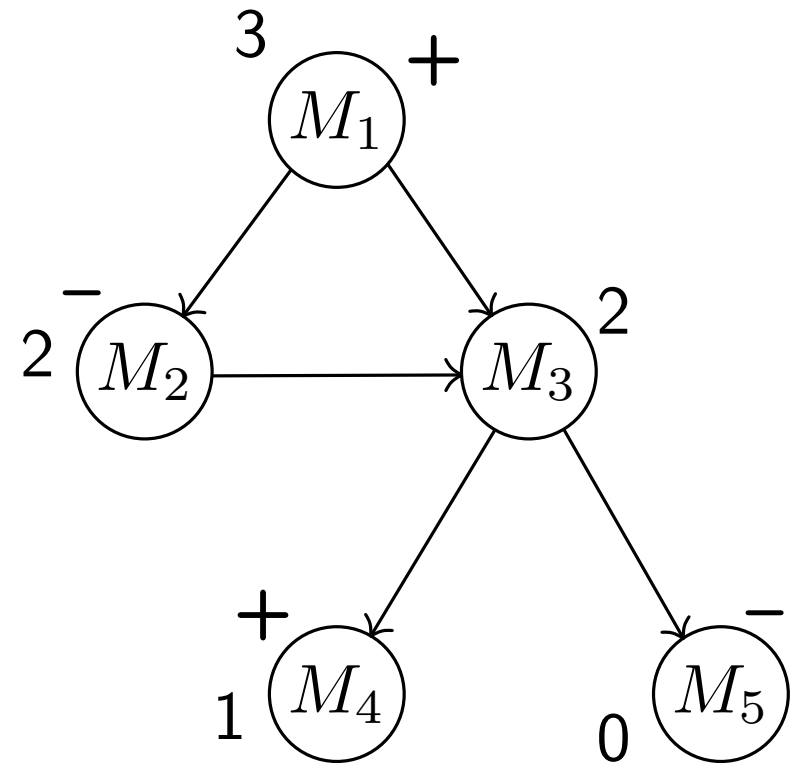
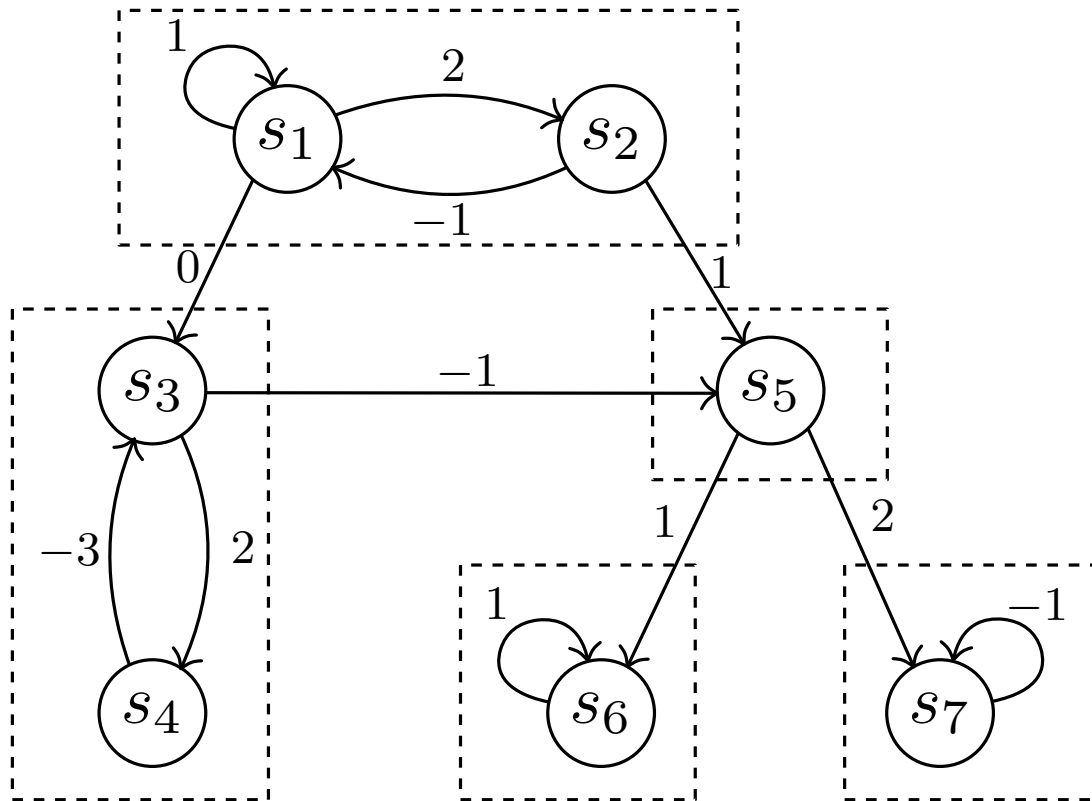
The *processing order* of a node in G_M is defined as:

- ▶ $p(M) = 0$ if M is a $-$ leaf.
- ▶ $p(M) = 1$ if M is a $+$ leaf.

▶ Other nodes:

$$p(M) = \max\{1 + p(N) \mid M \rightarrow^* N, \text{ and } M \text{ and } N \text{ switch}\}$$

Partition Graph (example)



The *processing order* of a node in G_M is defined as:

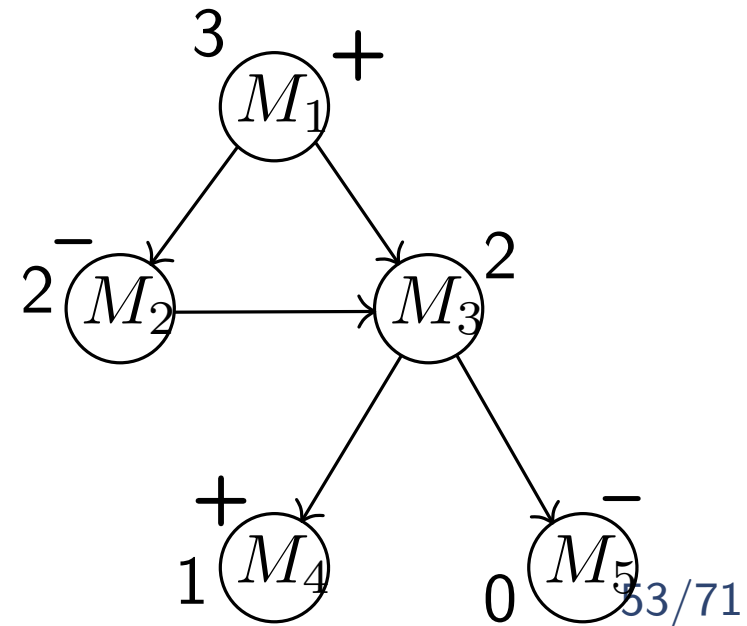
- ▶ $p(M) = 0$ if M is a $-$ leaf.
- ▶ $p(M) = 1$ if M is a $+$ leaf.

▶ Other nodes:

$$p(M) = \max\{1 + p(N) \mid M \rightarrow^* N, \text{ and } M \text{ and } N \text{ switch}\}$$

Offline Monitoring Algorithm

A node M in the partition graph is a **legal** specification
...whose inputs are the streams in the nodes N with $M \rightarrow N$.



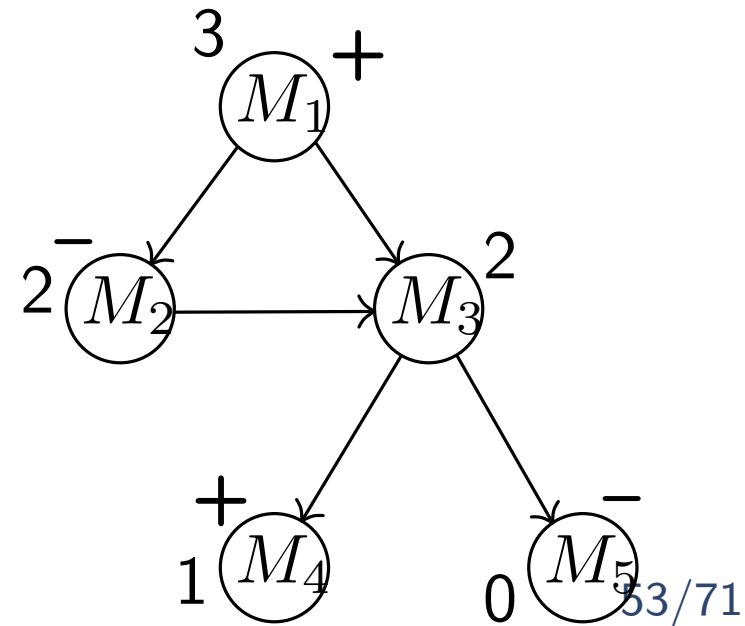
Offline Monitoring Algorithm

A node M in the partition graph is a **legal** specification
...whose inputs are the streams in the nodes N with $M \rightarrow N$.

Offline algorithm

For $i = 0$ to $\max(p(M))$, with increment 2:

1. Apply online algorithm **forward** to specs M with $p(M) = i$
2. Apply online algorithm **backwards** to specs M with $p(M) = i + 1$



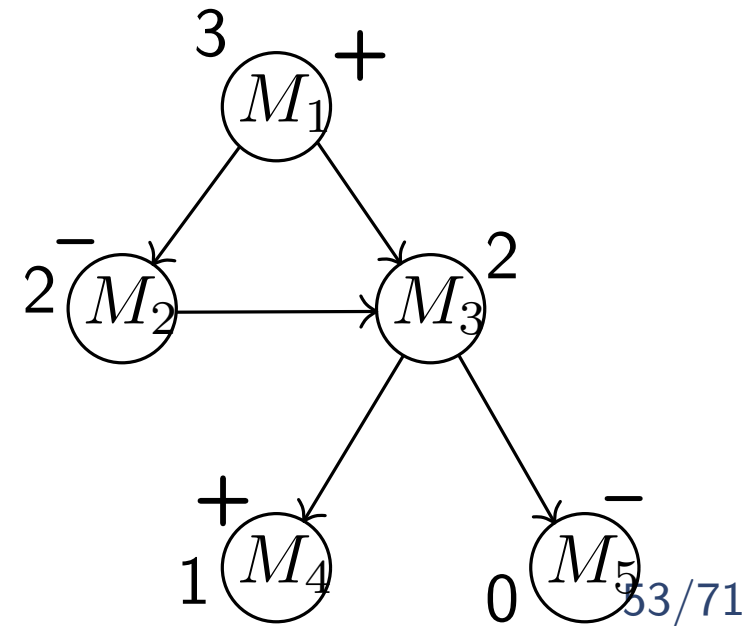
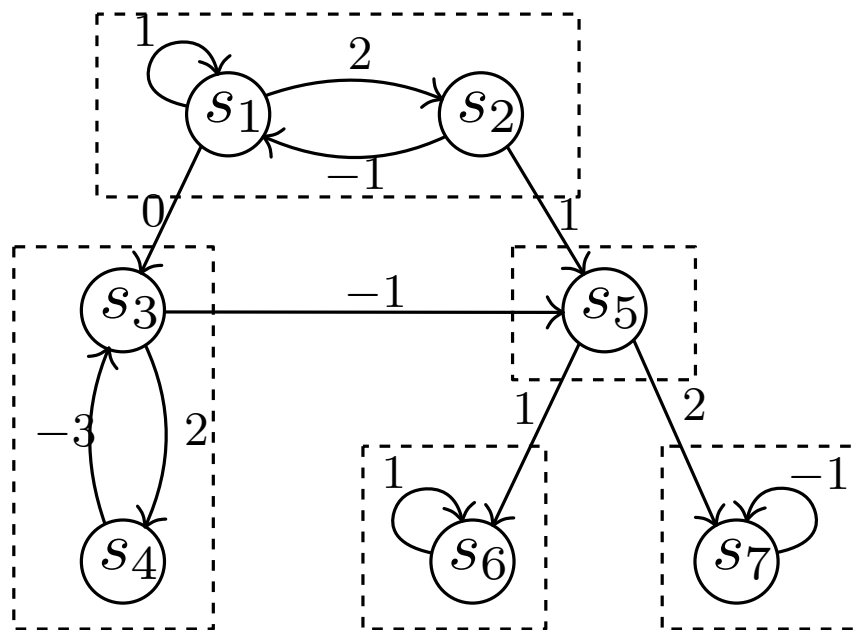
Offline Monitoring Algorithm

A node M in the partition graph is a **legal** specification
...whose inputs are the streams in the nodes N with $M \rightarrow N$.

Offline algorithm

For $i = 0$ to $\max(p(M))$, with increment 2:

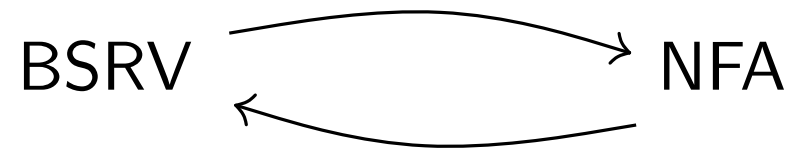
1. Apply online algorithm **forward** to specs M with $p(M) = i$
2. Apply online algorithm **backwards** to specs M with $p(M) = i + 1$



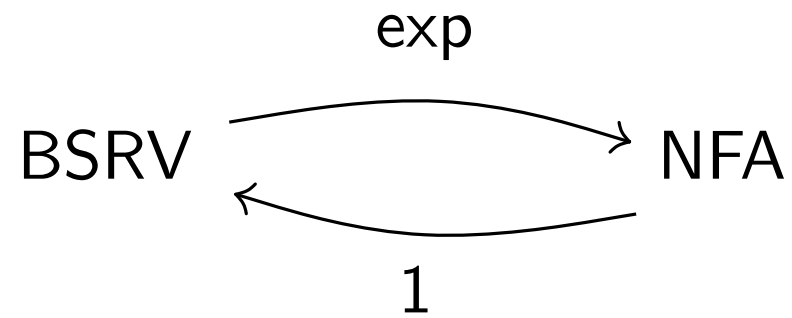
Boolean SRV

Theoretical Results

Main Idea



Main Idea



BSRV as Language Recognizers

- Given SPEC φ :

$$\mathcal{L}(\varphi) := \{\bar{\tau} \mid (\bar{\tau}, \bar{\sigma}) \models \varphi \text{ for some } \bar{\sigma}\}$$

BSRV as Language Recognizers

- Given SPEC φ :

$$\mathcal{L}(\varphi) := \{\bar{\tau} \mid (\bar{\tau}, \bar{\sigma}) \models \varphi \text{ for some } \bar{\sigma}\}$$

- Example:

output bool $y := \text{if } E \text{ then } y \text{ else } \neg y$

$$\begin{aligned} E := & (\text{first} \rightarrow (x \wedge y)) \wedge \\ & (y \rightarrow \neg y[+1|\text{false}]) \wedge \\ & (\neg y \rightarrow (x[+1|\text{true}] \wedge y[+1|\text{true}])) \end{aligned}$$

BSRV as Language Recognizers

- ▶ Given SPEC φ :

$$\mathcal{L}(\varphi) := \{\bar{\tau} \mid (\bar{\tau}, \bar{\sigma}) \models \varphi \text{ for some } \bar{\sigma}\}$$

- ▶ Example:

output **bool** $y := \text{if } E \text{ then } y \text{ else } \neg y$

$$\begin{aligned} E := & (\text{first} \rightarrow (x \wedge y)) \wedge \\ & (y \rightarrow \neg y[+1|\text{false}]) \wedge \\ & (\neg y \rightarrow (x[+1|\text{true}] \wedge y[+1|\text{true}])) \end{aligned}$$

- ▶ The language $\mathcal{L}(\varphi)$:

{input x holds at odd positions}

From BSRV to NFA

- Given φ we build an NFA over 2^X :

Forward distance f : maximum k in $t[k|d]$

Backwards distance b : maximum k in $t[-k|d]$

From BSRV to NFA

input stream variables



- Given φ we build an NFA over 2^X :

Forward distance f : maximum k in $t[k|d]$

Backwards distance b : maximum k in $t[-k|d]$

From BSRV to NFA

- ▶ Given φ we build an NFA over 2^X :

Forward distance f : maximum k in $t[k|d]$

Backwards distance b : maximum k in $t[-k|d]$

- ▶ The states are built from $A = 2^{X \cup Y}$ and $A_{\perp} = A \cup \perp$

input stream variables

output stream variables

From BSRV to NFA

- ▶ Given φ we build an NFA over 2^X :

Forward distance f : maximum k in $t[k|d]$

Backwards distance b : maximum k in $t[-k|d]$

- ▶ The states are built from $A = 2^{X \cup Y}$ and $A_{\perp} = A \cup \perp$

From BSRV to NFA

- ▶ Given φ we build an NFA over 2^X :

Forward distance f : maximum k in $t[k|d]$

Backwards distance b : maximum k in $t[-k|d]$

- ▶ The states are built from $A = 2^{X \cup Y}$ and $A_{\perp} = A \cup \perp$

$$P_{\varphi} : (a_{-b}, \dots, a_{-1}, a, a_1, \dots, a_f)$$

From BSRV to NFA

- ▶ Given φ we build an NFA over 2^X :

Forward distance f : maximum k in $t[k|d]$

Backwards distance b : maximum k in $t[-k|d]$

- ▶ The states are built from $A = 2^{X \cup Y}$ and $A_{\perp} = A \cup \perp$

$$P_{\varphi} : (\underbrace{a_{-b}, \dots, a_{-1}}_{\text{history}}, \underbrace{a}_{\text{current}}, \underbrace{a_1, \dots, a_f}_{\text{look-ahead}})$$

From BSRV to NFA

- ▶ Given φ we build an NFA over 2^X :

Forward distance f : maximum k in $t[k|d]$

Backwards distance b : maximum k in $t[-k|d]$

- ▶ The states are built from $A = 2^{X \cup Y}$ and $A_{\perp} = A \cup \perp$

$$P_{\varphi} : (a_{-b}, \dots, a_{-1}, a, a_1, \dots, a_f)$$

From BSRV to NFA

- ▶ Given φ we build an NFA over 2^X :

Forward distance f : maximum k in $t[k|d]$

Backwards distance b : maximum k in $t[-k|d]$

- ▶ The states are built from $A = 2^{X \cup Y}$ and $A_{\perp} = A \cup \perp$

$$P_{\varphi} : (a_{-b}, \dots, a_{-1}, a, a_1, \dots, a_f)$$

- ▶ States: $Q = \{p \in P_{\varphi} \mid \text{for every output } s, \llbracket s, p \rrbracket = \llbracket e, p \rrbracket \}$

From BSRV to NFA

- ▶ Given φ we build an NFA over 2^X :

Forward distance f : maximum k in $t[k|d]$

Backwards distance b : maximum k in $t[-k|d]$

- ▶ The states are built from $A = 2^{X \cup Y}$ and $A_{\perp} = A \cup \perp$

$$P_{\varphi} : (a_{-b}, \dots, a_{-1}, a, a_1, \dots, a_f)$$

- ▶ Initial: fresh q_0

From BSRV to NFA

- ▶ Given φ we build an NFA over 2^X :

Forward distance f : maximum k in $t[k|d]$

Backwards distance b : maximum k in $t[-k|d]$

- ▶ The states are built from $A = 2^{X \cup Y}$ and $A_{\perp} = A \cup \perp$

$$P_{\varphi} : (a_{-b}, \dots, a_{-1}, a, a_1, \dots, a_f)$$

- ▶ Initial: fresh q_0
- ▶ Transition: $\delta(q_0, i) = (\perp, \dots, \perp, a, a_1, \dots, a_f)$ with $a \cap X = i$

From BSRV to NFA

- ▶ Given φ we build an NFA over 2^X :

Forward distance f : maximum k in $t[k|d]$

Backwards distance b : maximum k in $t[-k|d]$

- ▶ The states are built from $A = 2^{X \cup Y}$ and $A_{\perp} = A \cup \perp$

$$P_{\varphi} : (a_{-b}, \dots, a_{-1}, a, a_1, \dots, a_f)$$

- ▶ Initial: fresh q_0

- ▶ Transition: $\delta((\perp, \dots, \perp, a, a_1 \dots, a_f) , i)$

$$(\perp, \dots, \underset{\swarrow}{a}, \underset{\swarrow}{a_1} \dots, \underset{\swarrow}{a_f}, d) \quad \text{with } a \cap X = i$$

From BSRV to NFA

- ▶ Given φ we build an NFA over 2^X :

Forward distance f : maximum k in $t[k|d]$

Backwards distance b : maximum k in $t[-k|d]$

- ▶ The states are built from $A = 2^{X \cup Y}$ and $A_{\perp} = A \cup \perp$

$$P_{\varphi} : (a_{-b}, \dots, a_{-1}, a, a_1, \dots, a_f)$$

- ▶ Final: $(a_{-b}, \dots, a_{-1}, a, \perp, \dots, \perp)$

From BSRV to NFA

- ▶ Given φ we build an NFA over 2^X :

Forward distance f : maximum k in $t[k|d]$

Backwards distance b : maximum k in $t[-k|d]$

- ▶ The states are built from $A = 2^{X \cup Y}$ and $A_{\perp} = A \cup \perp$

$$P_{\varphi} : (a_{-b}, \dots, a_{-1}, a, a_1, \dots, a_f)$$

The NFA is **exponentially** bigger than the BSRV.

From LTL to BSRV

- ▶ Is the (BSRV to NFA) translation tight?

From LTL to BSRV

- ▶ Is the (BSRV to NFA) translation tight?
- ▶ Consider LTL+past:

$$p \mid \neg a \mid a \wedge b \mid \bigcirc a \mid \ominus a \mid a \mathcal{U} b \mid a \mathcal{S} b$$

From LTL to BSRV

- Is the (BSRV to NFA) translation tight?
- Consider LTL+past:

$$p \mid \neg a \mid a \wedge b \mid \bigcirc a \mid \ominus a \mid a \mathcal{U} b \mid a \mathcal{S} b$$

- Given ψ , the output streams are $Y = SF(\psi) \cup \{\text{init}\}$

$$\text{init} : \text{first} \rightarrow (y_\psi \vee \neg \text{init})$$

$$y_p : p$$

$$y_{\neg a} : \neg y_a$$

$$y_{a \vee b} : y_a \vee y_b$$

$$y_{\bigcirc a} : y_a[+1|\text{false}]$$

$$y_{\ominus a} : y_a[-1|\text{false}]$$

$$y_{a \mathcal{U} b} : y_b \vee (\neg \text{last} \wedge y_a \wedge y_{a \mathcal{U} b}[+1|\text{true}])$$

$$y_{a \mathcal{S} b} : y_b \vee (\neg \text{first} \wedge y_a \wedge y_{a \mathcal{S} b}[-1|\text{true}])$$

From LTL to BSRV

- ▶ Is the (BSRV to NFA) translation tight?
- ▶ Consider LTL+past:

$$p \mid \neg a \mid a \wedge b \mid \bigcirc a \mid \ominus a \mid a \mathcal{U} b \mid a \mathcal{S} b$$

- ▶ LTL+past is exponentially more succinct than NFA [LMS'02]

BSRV is **exponentially** more succinct than NFA.

From NFA to BSRV

- ▶ Start from $A : \langle \Sigma, Q, q_0, \delta, F \rangle$ Create:

From NFA to BSRV

- ▶ Start from $A : \langle \Sigma, Q, q_0, \delta, F \rangle$ Create:
- ▶ Start φ with $Y = Q \cup \{\text{control}\}$.

$\text{control} = \text{if } E_{\text{ev}} \text{ then control else } \neg \text{control}$

$E_{\text{ev}} : \text{unique} \wedge \text{initial} \wedge \text{transition} \wedge \text{accepting}$

From NFA to BSRV

- ▶ Start from $A : \langle \Sigma, Q, q_0, \delta, F \rangle$ Create:
- ▶ Start φ with $Y = Q \cup \{\text{control}\}$.

$\text{control} = \text{if } E_{\text{ev}} \text{ then control else } \neg \text{control}$

$E_{\text{ev}} : \text{unique} \wedge \text{initial} \wedge \text{transition} \wedge \text{accepting}$

$$\text{unique} \stackrel{\text{def}}{=} \bigvee_{q \in Q} (q \wedge \bigwedge_{p \in Q \setminus \{q\}} \neg p)$$

From NFA to BSRV

- ▶ Start from $A : \langle \Sigma, Q, q_0, \delta, F \rangle$ Create:
- ▶ Start φ with $Y = Q \cup \{\text{control}\}$.

$\text{control} = \text{if } E_{\text{ev}} \text{ then control else } \neg \text{control}$

$E_{\text{ev}} : \text{unique} \wedge \text{initial} \wedge \text{transition} \wedge \text{accepting}$

$\text{init} \stackrel{\text{def}}{=} (\text{first} \longrightarrow q_0)$

From NFA to BSRV

- ▶ Start from $A : \langle \Sigma, Q, q_0, \delta, F \rangle$ Create:
- ▶ Start φ with $Y = Q \cup \{\text{control}\}$.

$\text{control} = \text{if } E_{\text{ev}} \text{ then control else } \neg \text{control}$

$E_{\text{ev}} : \text{unique} \wedge \text{initial} \wedge \text{transition} \wedge \text{accepting}$

$$\text{transition} \stackrel{\text{def}}{=} \bigwedge_{q \in Q} \bigwedge_{a \in \Sigma} ((q \wedge a) \longrightarrow \bigvee_{p \in \delta(q, a)} p[+1|\text{true}])$$

From NFA to BSRV

- ▶ Start from $A : \langle \Sigma, Q, q_0, \delta, F \rangle$ Create:
- ▶ Start φ with $Y = Q \cup \{\text{control}\}$.

$\text{control} = \text{if } E_{\text{ev}} \text{ then control else } \neg \text{control}$

$E_{\text{ev}} : \text{unique} \wedge \text{initial} \wedge \text{transition} \wedge \text{accepting}$

$$\text{accepting} \stackrel{\text{def}}{=} (\text{last} \longrightarrow \bigvee_{q \rightarrow_a F} (q \wedge a))$$

From NFA to BSRV

- ▶ Start from $A : \langle \Sigma, Q, q_0, \delta, F \rangle$ Create:
- ▶ Start φ with $Y = Q \cup \{\text{control}\}$.

$\text{control} = \text{if } E_{\text{ev}} \text{ then control else } \neg \text{control}$

$E_{\text{ev}} : \text{unique} \wedge \text{initial} \wedge \text{transition} \wedge \text{accepting}$

The translation from NFA to BSRV is **linear**.

Main results (Expressivity)

Theorem: BSRV as recognizers capture the set of all regular languages

Main results (Expressivity)

Theorem: BSRV as recognizers capture the set of all regular languages

Theorem: BSRV are closed under union, concatenation and Kleene star

Main results (Expressivity)

Theorem: BSRV as recognizers capture the set of all regular languages

Theorem: BSRV are closed under union, concatenation and Kleene star

Theorem: A BSRV φ is **well-defined** if \mathcal{A}_φ is **unambiguous** and **universal**

Main Results (Offline Algorithm)

Offline Algorithm:

1. Take φ , compute \mathcal{A}_φ .
2. Process σ_X forward
 computing a stream of sets of states of \mathcal{A}_φ .
3. At the end, only one state is guaranteed to be final
4. Process the powerset stream backwards,
 generating the **unique** state

Corollary: Given φ of alternation depth k .

We can construct ψ (exponential) of alternation depth 1.

Remark φ requires $k + 1$ passes. ψ requires 2 passes.

Decision Problems

| | | |
|--|-------------------|-------------|
| under-defined | PSPACE-complete | |
| well-defined | EXPTIME | PSPACE-hard |
| over-defined | EXPSPACE-complete | |
| semantic equivalence | EXPSPACE-complete | |
| language emptiness | PSPACE-complete | |
| language universality, equivalence, inclusion | EXPSPACE-complete | |

Decision Problems

| | | |
|--|-------------------|-------------|
| under-defined | PSPACE-complete | |
| well-defined | EXPTIME | PSPACE-hard |
| over-defined | EXPSPACE-complete | |
| semantic equivalence | EXPSPACE-complete | |
| language emptiness | PSPACE-complete | |
| language universality, equivalence, inclusion | EXPSPACE-complete | |

under-defined: φ is under-defined iff \mathcal{A}_φ is not unambiguous.

Decision Problems

| | | |
|--|-------------------|-------------|
| under-defined | PSPACE-complete | |
| well-defined | EXPTIME | PSPACE-hard |
| over-defined | EXPSPACE-complete | |
| semantic equivalence | EXPSPACE-complete | |
| language emptiness | PSPACE-complete | |
| language universality, equivalence, inclusion | EXPSPACE-complete | |

under-defined: φ is under-defined iff \mathcal{A}_φ is not unambiguous.

over-defined: φ is over-defined iff \mathcal{A}_φ is not universal.

Decision Problems

| | | |
|--|-------------------|-------------|
| under-defined | PSPACE-complete | |
| well-defined | EXPTIME | PSPACE-hard |
| over-defined | EXPSPACE-complete | |
| semantic equivalence | EXPSPACE-complete | |
| language emptiness | PSPACE-complete | |
| language universality, equivalence, inclusion | EXPSPACE-complete | |

under-defined: φ is under-defined iff \mathcal{A}_φ is not unambiguous.

over-defined: φ is over-defined iff \mathcal{A}_φ is not universal.

well-defined: checking universality of unambiguous is PTIME
checking unambiguous is PSPACE

Extensions

Nested words

Syntax of CaReT:

$\varphi ::=$ **true** $\varphi \wedge \varphi$ $\neg \varphi$ $\bigcirc \varphi$ $\varphi \mathcal{U} \varphi$

Nested words

Syntax of CaReT:

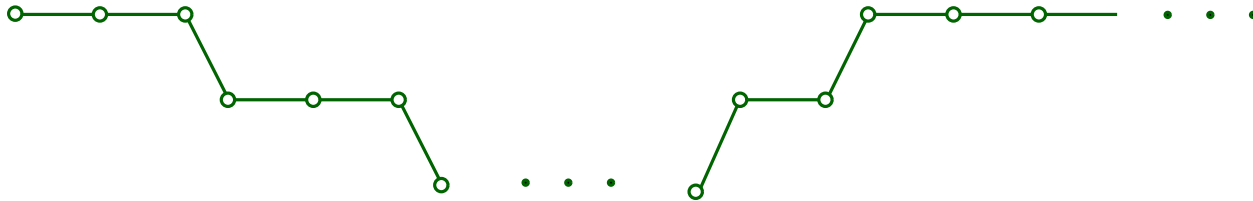
$$\begin{array}{l} \varphi ::= \text{true} \quad \varphi \wedge \varphi \quad \neg \varphi \quad \bigcirc \varphi \quad \varphi \mathcal{U} \varphi \\ \quad \bigcirc^a \varphi \quad \varphi \mathcal{U}^a \varphi \quad \bigcirc^c \varphi \quad \varphi \mathcal{U}^c \varphi \end{array}$$

Nested words

Syntax of CaReT:

$\varphi ::=$ **true** $\varphi \wedge \varphi$ $\neg \varphi$ $\bigcirc \varphi$ $\varphi \mathcal{U} \varphi$

$\bigcirc^a \varphi$ $\varphi \mathcal{U}^a \varphi$ $\bigcirc^c \varphi$ $\varphi \mathcal{U}^c \varphi$

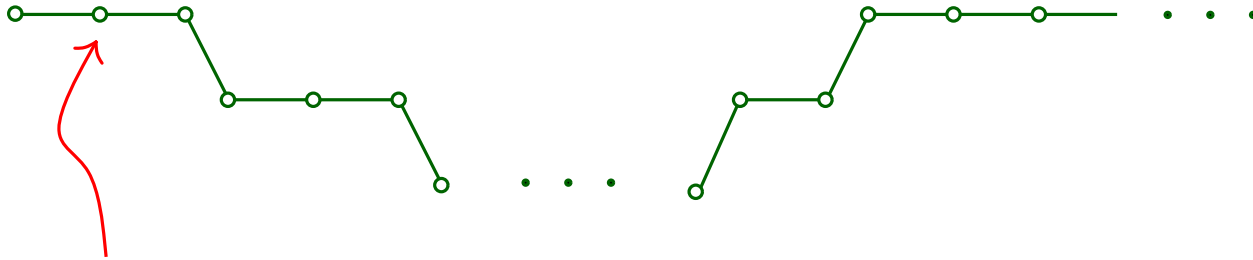


Nested words

Syntax of CaReT:

$\varphi ::=$ **true** $\varphi \wedge \varphi$ $\neg \varphi$ $\bigcirc \varphi$ $\varphi \mathcal{U} \varphi$

$\bigcirc^a \varphi$ $\varphi \mathcal{U}^a \varphi$ $\bigcirc^c \varphi$ $\varphi \mathcal{U}^c \varphi$

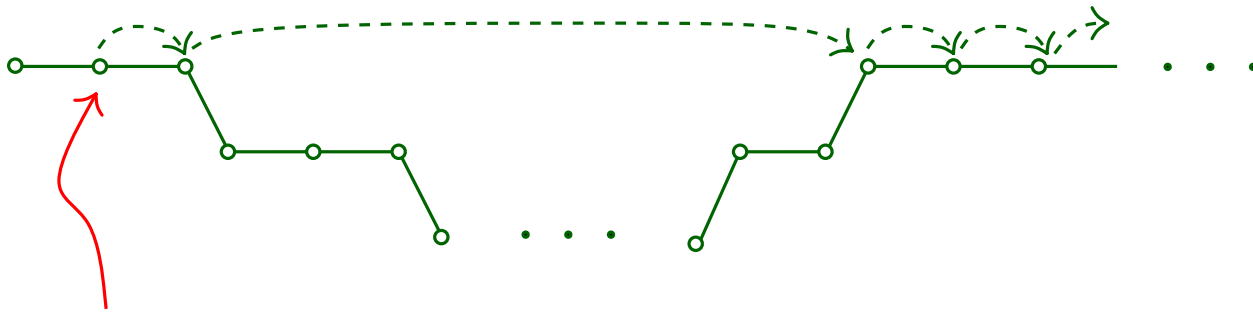


Nested words

Syntax of CaReT:

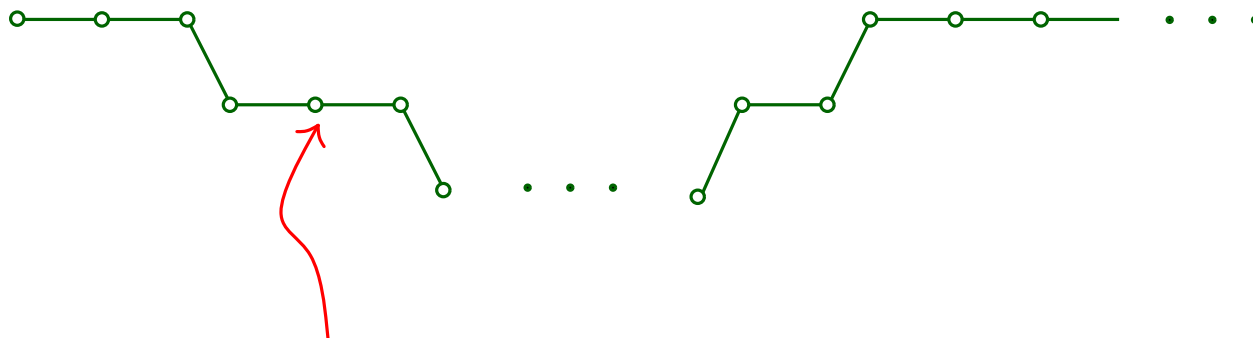
$\varphi ::=$ **true** $\varphi \wedge \varphi$ $\neg \varphi$ $\bigcirc \varphi$ $\varphi \mathcal{U} \varphi$

$\bigcirc^a \varphi$ $\varphi \mathcal{U}^a \varphi$ $\bigcirc^c \varphi$ $\varphi \mathcal{U}^c \varphi$



Nested words

Syntax of CaReT:

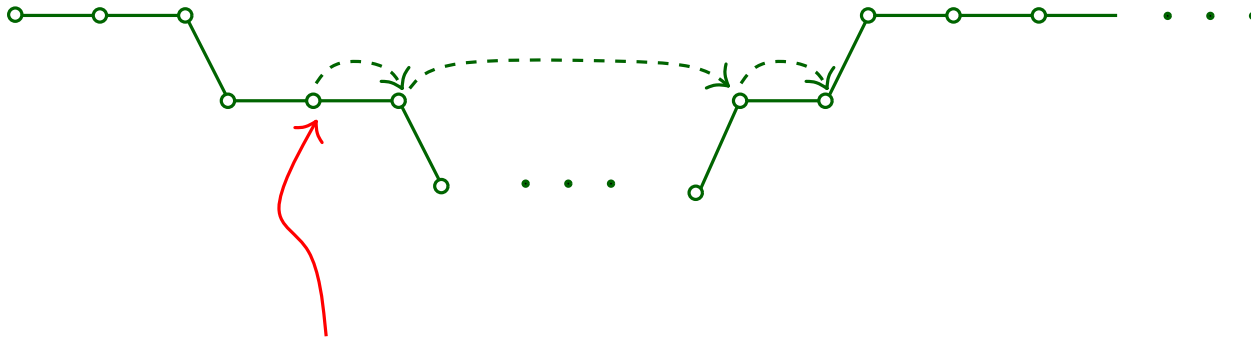
$$\varphi ::= \text{true} \quad \varphi \wedge \varphi \quad \neg \varphi \quad \text{O}\varphi \quad \varphi \mathcal{U} \varphi$$
$$\begin{array}{cc|cc} \bigcirc^a \varphi & \varphi \mathcal{U}^a \varphi & \bigcirc^c \varphi & \varphi \mathcal{U}^c \varphi \end{array}$$


Nested words

Syntax of CaReT:

$\varphi ::=$ **true** $\varphi \wedge \varphi$ $\neg \varphi$ $\bigcirc \varphi$ $\varphi \mathcal{U} \varphi$

$\bigcirc^a \varphi$ $\varphi \mathcal{U}^a \varphi$ $\bigcirc^c \varphi$ $\varphi \mathcal{U}^c \varphi$

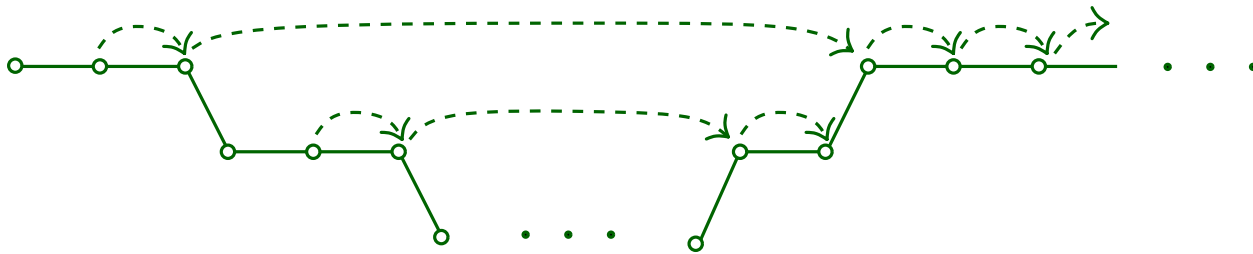


Nested words

Syntax of CaReT:

$\varphi ::=$ **true** $\varphi \wedge \varphi$ $\neg \varphi$ $\bigcirc \varphi$ $\varphi \mathcal{U} \varphi$

$\bigcirc^a \varphi$ $\varphi \mathcal{U}^a \varphi$ $\bigcirc^c \varphi$ $\varphi \mathcal{U}^c \varphi$

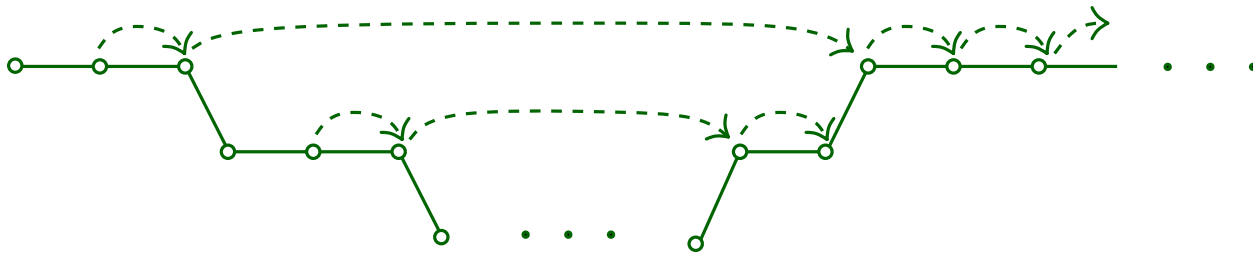


Nested words

Syntax of CaReT:

$\varphi ::=$ **true** $\varphi \wedge \varphi$ $\neg \varphi$ $\bigcirc \varphi$ $\varphi \mathcal{U} \varphi$

$\bigcirc^a \varphi$ $\varphi \mathcal{U}^a \varphi$ $\bigcirc^c \varphi$ $\varphi \mathcal{U}^c \varphi$



In NestedSRV we extend offsets:

$s[n + k, d]$

$s[n \oplus k, d]$

where \oplus means to follow the abstract next and prev

The operational semantics use stack pushing and popping R and U

Parametrized SRV

- ▶ Lola2.0 extends SRV with parametrization

Parametrized SRV

- ▶ Lola2.0 extends SRV with parametrization

- ▶ Main idea:

spawn an *instance* copy of a stream when a *condition* holds
with observed *parameters*

then, pass to the instance the necessary events

Parametrized SRV

- ▶ Lola2.0 extends SRV with parametrization

- ▶ Main idea:

spawn an *instance* copy of a stream when a *condition* holds
with observed *parameters*

then, pass to the instance the necessary events

- ▶ New syntax
-
- The diagram illustrates the new syntax for parametrized SRV. It shows a sequence of declarations and a function call. Annotations with arrows point to specific parts of the syntax:
- parameters**: Points to the list of parameters $\langle p_1 \dots p_l \rangle$ in the stream declaration.
 - invocation condition**: Points to the s_{inv} in the `invoke` declaration.
 - extension**: Points to the s_{ext} in the `extend` declaration.
 - termination**: Points to the s_{ter} in the `terminate` declaration.
- The syntax is as follows:
- ```
output T s⟨p1 ... pl⟩
 invoke : sinv;
 extend : sext;
 terminate : ster
:= e(t1, ..., s1, ..., p1, ...)
```

# Parametrized SRV

- ▶ Lola2.0 extends SRV with parametrization

- ▶ Main idea:

spawn an *instance* copy of a stream when a *condition* holds  
with observed *parameters*

then, pass to the instance the necessary events

- ▶ Example

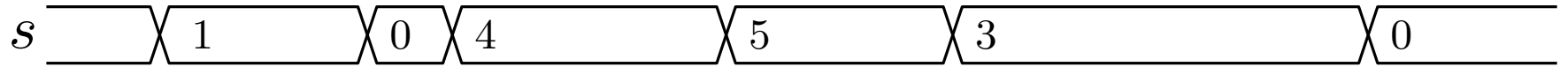
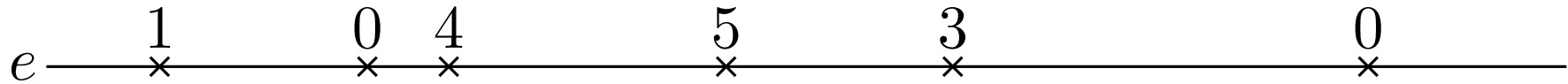
```
input int prodId
output int salesFreq <int id>
 invoke: prodId
 extend: prodId = id
 := prodId[1h, 0, count(id)]
trigger any(salesFreq > 100)
```

# SRV for Real-Time

► Extension:

input streams can be time-stamped *event streams*

equivalently *piece-wise constant signals*

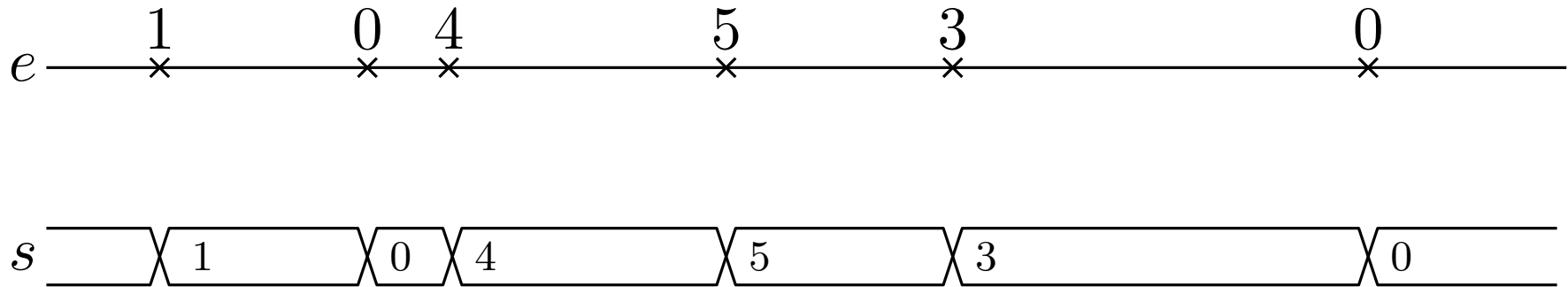


# SRV for Real-Time

## ► Extension:

input streams can be time-stamped *event streams*

equivalently *piece-wise constant signals*



## ► Target application: observation of *timed-asynchronous systems*

- multi-core hardware (non-intrusive, FPGA)
- RV for cloud monitoring

## ► The synchronous notion of time is relaxed:

*Asynchronous Stream Runtime Verification*

# SRV for Real-Time (sol 1: TeSSLa 1.0)

- ▶ Input streams can arrive *asynchronously* at different speeds
- system time (time-stamps)  
 $\neq$   
monitor time (arrival and processing)

# SRV for Real-Time (sol 1: TeSSLa 1.0)

- ▶ Input streams can arrive *asynchronously* at different speeds

system time (time-stamps)

≠

monitor time (arrival and processing)

- ▶ Solution in TeSSLa 1.0:

1. forbid explicit *time* and *offsets*
2. provide a collection of native operators (all *past*)

*shift*                  *within*                  *delay*

3. user defined functions (non-temporal)
4. *no recursion*

# SRV for Real-Time (sol 1: TeSSLa 1.0)

- ▶ Input streams can arrive *asynchronously* at different speeds

system time (time-stamps)

≠

monitor time (arrival and processing)

- ▶ *Deprecated*

Superseded by TeSSLa 2.0

*shift*

*within*

*delay*

3. user defined functions (non-temporal)

4. *no recursion*

# SRV for Real-Time (sol 2: TeSSLa 2.0)

- ▶ Provided temporal building blocks

*default*      *last*      *delayLast*      *time*      *lift*

- ▶ (all past temporal operators)
- ▶ Provides *controlled recursion*
- ▶ Evaluation is guaranteed to be finite state (FPGA)

<https://www.isp.uni-luebeck.de/tessla>



# SRV for Real-Time (sol 3: Striver)

- ▶ Main idea: keep time explicit

# SRV for Real-Time (sol 3: Striver)

- ▶ Main idea: keep time explicit
- ▶ Generalize  $s[-1]$  by the *previous* event in a stream time of  $s$
- ▶ Defining equations

# SRV for Real-Time (sol 3: Striver)

- ▶ Main idea: keep time explicit
- ▶ Generalize  $s[-1]$  by the *previous* event in a stream time of  $s$
- ▶ Defining equations

output  $T\ s \quad := e$

# SRV for Real-Time (sol 3: Striver)

- ▶ Main idea: keep time explicit
- ▶ Generalize  $s[-1]$  by the *previous* event in a stream time of  $s$
- ▶ Defining equations

output  $T$   $s$   $(\overset{\curvearrowright}{t}) := e$  — how to capture legal  $t$  ??

# SRV for Real-Time (sol 3: Striver)

- ▶ Main idea: keep time explicit
- ▶ Generalize  $s[-1]$  by the *previous* event in a stream time of  $s$
- ▶ Defining equations

$$\begin{array}{lll} \text{output } T & s.ticks & := \alpha \\ & s.val(t) & := e \end{array}$$

# SRV for Real-Time (sol 3: Striver)

- ▶ Main idea: keep time explicit
- ▶ Generalize  $s[-1]$  by the *previous* event in a stream time of  $s$
- ▶ Defining equations

$$\begin{aligned} \text{output } T \quad s.ticks &:= \alpha \\ s.val(t) &:= e \end{aligned}$$

- Ticking expressions

$$\alpha ::= \{c\} \mid v.ticks \mid \alpha \cup \alpha \mid delay(w)$$

# SRV for Real-Time (sol 3: Striver)

- ▶ Main idea: keep time explicit
- ▶ Generalize  $s[-1]$  by the *previous* event in a stream time of  $s$
- ▶ Defining equations

$$\begin{aligned}\text{output } T \quad s.ticks &:= \alpha \\ s.val(t) &:= e\end{aligned}$$

- Ticking expressions

$$\alpha ::= \{c\} \mid v.ticks \mid \alpha \cup \alpha \mid delay(w)$$

- Value expressions

$$e ::= d \mid x(\tau_x) \mid f(e, \dots e) \mid \tau'$$

# SRV for Real-Time (sol 3: Striver)

- ▶ Main idea: keep time explicit
- ▶ Generalize  $s[-1]$  by the *previous* event in a stream time of  $s$
- ▶ Defining equations

$$\begin{aligned} \text{output } T \quad s.ticks &:= \alpha \\ s.val(t) &:= e \end{aligned}$$

- Ticking expressions

$$\alpha ::= \{c\} \mid v.ticks \mid \alpha \cup \alpha \mid delay(w)$$

- Value expressions

$$e ::= d \mid x(\tau_x) \mid f(e, \dots e) \mid \tau'$$

- Offset expressions

$$\begin{aligned} \tau_x &::= x.prevEq \tau' \mid x.prev \tau' \\ \tau' &::= t \mid \tau_z \end{aligned}$$



# Striver (examples)

►  $x : \text{default}(s, v)$

input  $T$

$s$

output  $T$

$x.ticks := s.ticks \cup \{0\}$

$x.val(t) := \text{if } t = 0 \text{ then } v \text{ else } s(t)$

# Striver (examples)

►  $x : \text{default}(s, v)$

input  $T$   $s$

output  $T$   $x.ticks := s.ticks \cup \{0\}$   
 $x.val(t) := \text{if } t = 0 \text{ then } v \text{ else } s(t)$

►  $x : \text{time}(s)$

input  $T$   $s$

output  $\text{time}$   $x.ticks := s.ticks$   
 $x.val(t) := t$

# Striver (examples)

►  $x : \text{default}(s, v)$

input  $T$   $s$   
output  $T$   $x.\text{ticks} := s.\text{ticks} \cup \{0\}$   
 $x.\text{val}(t) := \text{if } t = 0 \text{ then } v \text{ else } s(t)$

►  $x : \text{time}(s)$

input  $T$   $s$   
output  $\text{time}$   $x.\text{ticks} := s.\text{ticks}$   
 $x.\text{val}(t) := t$

►  $x : \text{lift}(f, r, s)$

input  $T_1$   $r$   
input  $T_2$   $s$   
output  $T$   $x.\text{ticks} := s.\text{ticks} \cup r.\text{ticks}$   
 $x.\text{val}(t) := \text{let } t_r = (r.\text{prev}_{\leq} t), t_s = (s.\text{prev}_{\leq} t) \text{ in } f(r(t_r), s(t_s))$

# Conclusions

- ▶ Stream RV specify behaviors as dependencies between streams
- ▶ SRV generalizes monitoring algs to the collection of statistics
- ▶ Well-formed specifications guarantee semantics
- ▶ Efficiently monitorability guarantee *finite state* monitors
- ▶ Offline monitors can be scheduled to (finite state) passes
- ▶ Current directions: generalizing SRV to
  - asynchronous time
  - real time
  - decentralized monitoring
  - distributed time
  - monitoring under uncertainty