



Runtime Verification for Timed Event Streams with Partial Information

Martin Leucker¹ César Sánchez² Torben Scheffel¹ **Malte Schmitz¹**
Daniel Thoma¹

¹Institute for Software Engineering and Programming Languages,
University of Lübeck, Germany

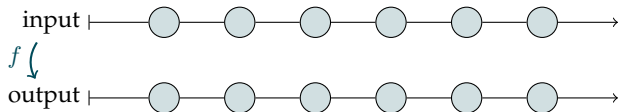
²IMDEA Software Institute, Spain

The 19th International Conference on Runtime Verification

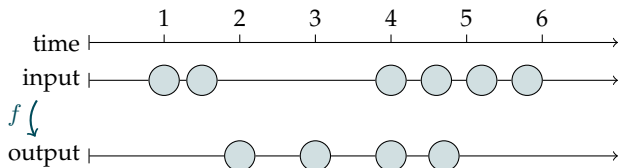
Event Sequence



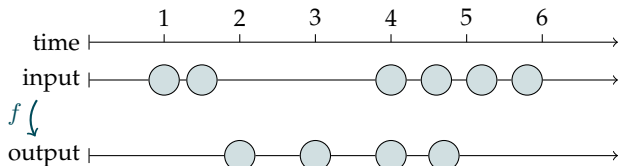
Stream Transformation



Non-Synchronized Stream Transformation



Non-Synchronized Stream Transformation



Every monotonous, continuous and future-independent stream transformation function f

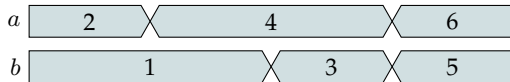
can be represented as

recursive equation system of stream transforming functions:

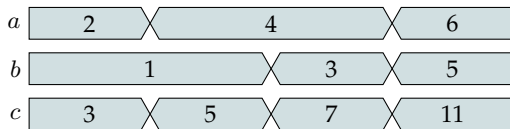
- ▶ **lift**
lifts functions on values to functions on streams
- ▶ **last**
gives access to values of previous events
- ▶ **delay**
generates events with additional timestamps



Stream Processing by Example

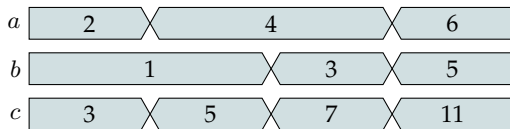


Stream Processing by Example



$$c = a + b$$

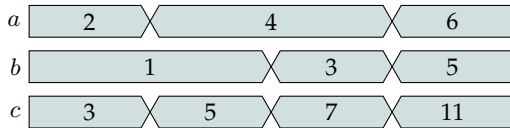
Stream Processing by Example



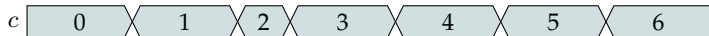
$$c = a + b$$



Stream Processing by Example

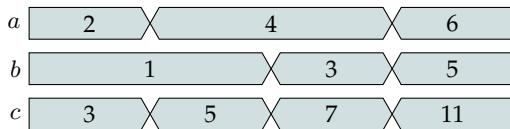


$$c = a + b$$

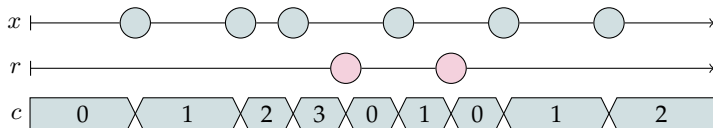


$$c = \text{count}(x)$$

Stream Processing by Example



$$c = a + b$$

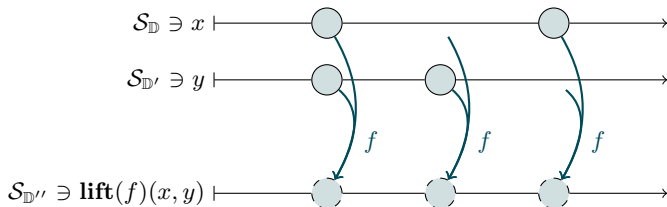


$$c = \text{count}(x, r)$$

Lift

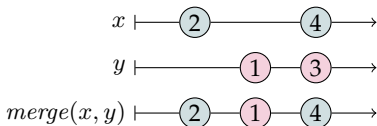
$$\mathbf{lift} : (\mathbb{D}_\perp \times \mathbb{D}'_\perp \rightarrow \mathbb{D}''_\perp) \rightarrow (\mathcal{S}_\mathbb{D} \times \mathcal{S}_{\mathbb{D}'_\perp} \rightarrow \mathcal{S}_{\mathbb{D}''_\perp})$$

$$\mathbb{D}_\perp := \mathbb{D} \cup \{\perp\}$$



Lift: Merge

- ▶ Process streams in an **event-oriented fashion**
- ▶ **Merge** combines two streams into one, giving preference to the first stream when both streams contain identical timestamps.



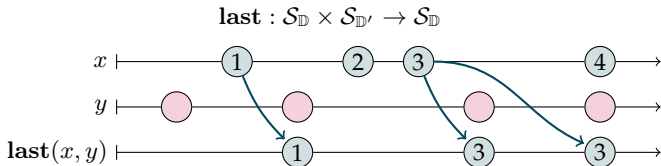
$$merge(x, y) = \mathbf{lift}(f)(x, y)$$

$$f : \mathbb{D}_{\perp} \times \mathbb{D}_{\perp} \rightarrow \mathbb{D}_{\perp}$$

$$f(a, b) = \begin{cases} b & \text{if } a = \perp \\ a & \text{else} \end{cases}$$

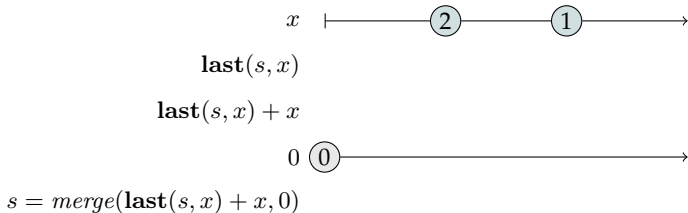
Accessing Previous Events: Last

- ▶ Needed to define properties over sequences of events.
- ▶ Last refers to values of events on one stream occurring strictly before events on another stream



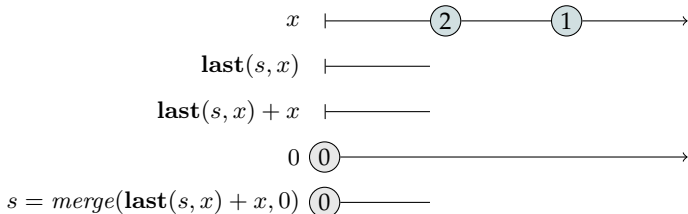
Recursive Equations in TeSSLa

- ▶ The **last** operator allows to write **recursive equations**
- ▶ The **merge** operation allows to **initialize** recursive equations with an initial event from an other stream.
- ▶ Express **aggregation** operations like the **sum** over all values of a stream.



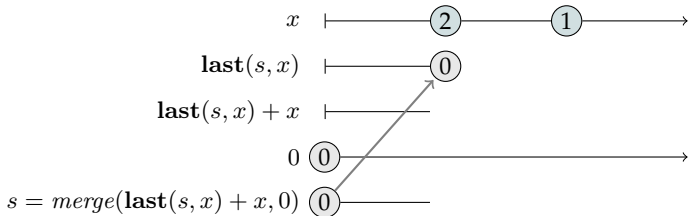
Recursive Equations in TeSSLa

- ▶ The **last** operator allows to write **recursive equations**
- ▶ The **merge** operation allows to **initialize** recursive equations with an initial event from an other stream.
- ▶ Express **aggregation** operations like the **sum** over all values of a stream.



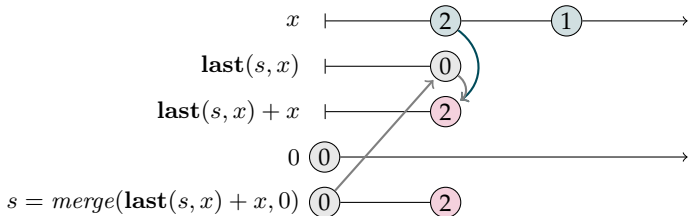
Recursive Equations in TeSSLa

- ▶ The **last** operator allows to write **recursive equations**
- ▶ The **merge** operation allows to **initialize** recursive equations with an initial event from an other stream.
- ▶ Express **aggregation** operations like the **sum** over all values of a stream.



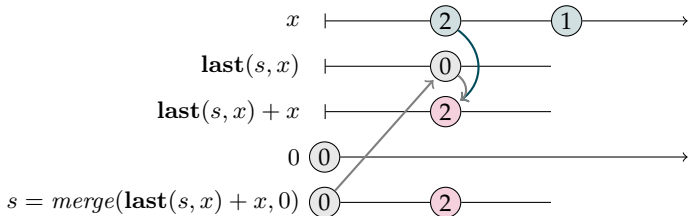
Recursive Equations in TeSSLa

- ▶ The **last** operator allows to write **recursive equations**
- ▶ The **merge** operation allows to **initialize** recursive equations with an initial event from an other stream.
- ▶ Express **aggregation** operations like the **sum** over all values of a stream.



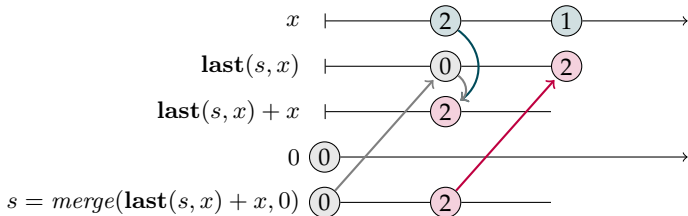
Recursive Equations in TeSSLa

- ▶ The **last** operator allows to write **recursive equations**
- ▶ The **merge** operation allows to **initialize** recursive equations with an initial event from an other stream.
- ▶ Express **aggregation** operations like the **sum** over all values of a stream.



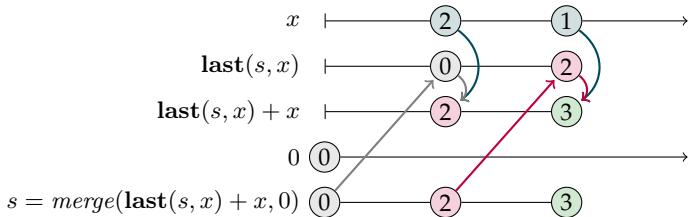
Recursive Equations in TeSSLa

- ▶ The **last** operator allows to write **recursive equations**
- ▶ The **merge** operation allows to **initialize** recursive equations with an initial event from an other stream.
- ▶ Express **aggregation** operations like the **sum** over all values of a stream.



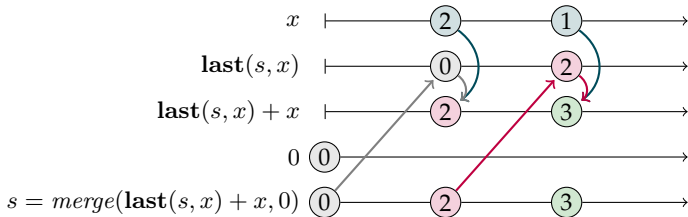
Recursive Equations in TeSSLa

- ▶ The **last** operator allows to write **recursive equations**
- ▶ The **merge** operation allows to **initialize** recursive equations with an initial event from an other stream.
- ▶ Express **aggregation** operations like the **sum** over all values of a stream.



Recursive Equations in TeSSLa

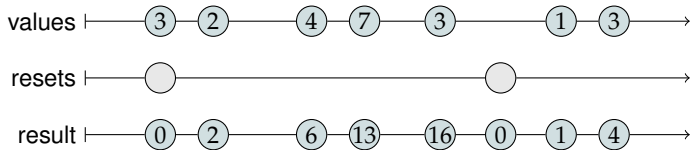
- ▶ The **last** operator allows to write **recursive equations**
- ▶ The **merge** operation allows to **initialize** recursive equations with an initial event from an other stream.
- ▶ Express **aggregation** operations like the **sum** over all values of a stream.



Stream Processing With Gaps

- ▶ Common challenge when applying RV to real-world systems: Incomplete traces with information missing in gaps.
- ▶ Assumption: We know when we stop getting information and when the trace becomes reliable again.
- ▶ Abstract event streams represent the set of all possible traces that could have occurred during gaps in the input trace.

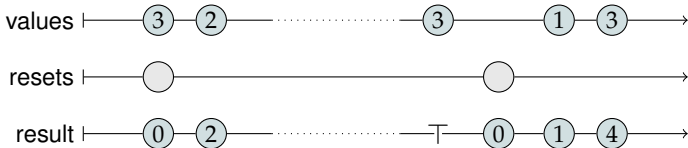
$$\text{result} = \text{sum}(\text{values}, \text{resets})$$



Stream Processing With Gaps

- ▶ Common challenge when applying RV to real-world systems: Incomplete traces with information missing in gaps.
- ▶ Assumption: We know when we stop getting information and when the trace becomes reliable again.
- ▶ Abstract event streams represent the set of all possible traces that could have occurred during gaps in the input trace.

$$\text{result} = \text{sum}^\#(\text{values}, \text{resets})$$



Abstract Lift

- ▶ $\mathbf{lift}^\#(f^\#)(s_1, \dots, s_n)$ is similar to $\mathbf{lift}(f)(s_1, \dots, s_n)$
- ▶ $f^\# : \mathbb{D}_{1\perp}^\# \times \dots \times \mathbb{D}_{n\perp}^\# \rightarrow \mathbb{D}_\perp^\#$ is abstraction of $f : \mathbb{D}_{1\perp} \times \dots \times \mathbb{D}_{n\perp} \rightarrow \mathbb{D}_\perp$
- ▶ $f^\#$ takes care of the gaps

Abstract Lift

- ▶ $\mathbf{lift}^\#(f^\#)(s_1, \dots, s_n)$ is similar to $\mathbf{lift}(f)(s_1, \dots, s_n)$
- ▶ $f^\# : \mathbb{D}_{1\perp}^\# \times \dots \times \mathbb{D}_{n\perp}^\# \rightarrow \mathbb{D}_\perp^\#$ is abstraction of $f : \mathbb{D}_{1\perp} \times \dots \times \mathbb{D}_{n\perp} \rightarrow \mathbb{D}_\perp$
- ▶ $f^\#$ takes care of the gaps

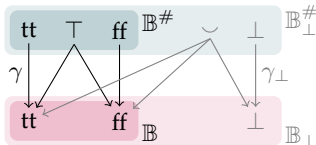
If $\mathbb{D}^\#$ is a data abstraction of \mathbb{D}

with an associated concretisation function γ ,

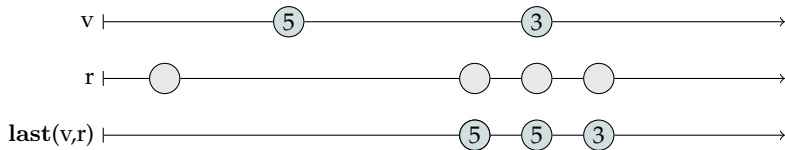
then $\mathbb{D}_\perp^\# = \mathbb{D}^\# \cup \{\perp, \smile\}$ is a data abstraction of \mathbb{D}_\perp

with an associated concretisation function $\gamma_\perp : \mathbb{D}_\perp^\# \rightarrow 2^{\mathbb{D} \cup \{\perp\}}$:

$$\gamma_\perp(d) = \begin{cases} \perp & \text{if } d = \perp \\ \mathbb{D} \cup \{\perp\} & \text{if } d = \smile \\ \gamma(d) & \text{if } d \in \mathbb{D}^\# \end{cases}$$



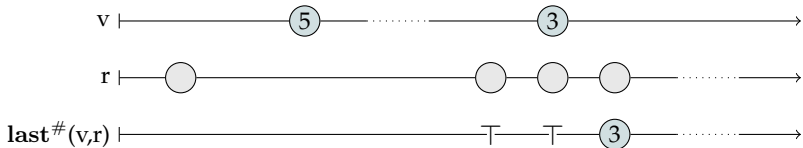
Last



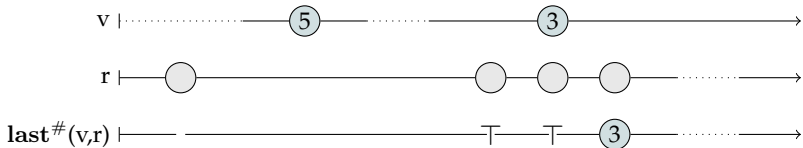
Abstract Last



Abstract Last



Abstract Last



Theoretical Results

Theorem

Every abstract TeSSLa operator is a perfect abstraction of its concrete counterpart.

Theorem

The abstract TeSSLa operators can be defined in terms of the concrete operators.

Theoretical Results

Theorem

Every abstract TeSSLa operator is a perfect abstraction of its concrete counterpart.

Theorem

The abstract TeSSLa operators can be defined in terms of the concrete operators.

Problem

Perfectness is not compositional.

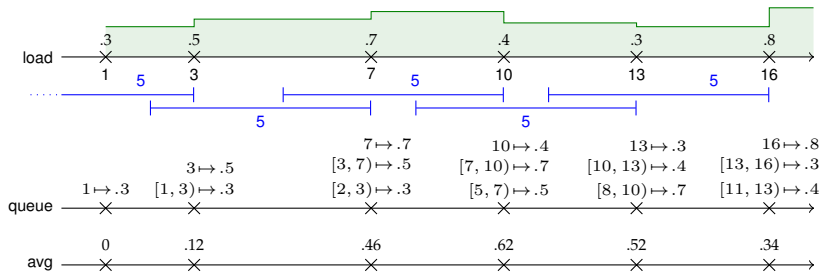
Theorem

If $f^\#$ is a perfect abstraction of f then $\mathbf{slift}(f^\#)^\#$ is a perfect abstraction of $\mathbf{slift}(f)$.

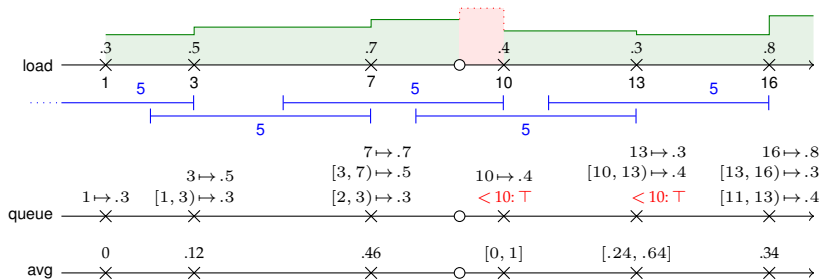
Theorem

$\mathbf{lastTime}^\#$ is a perfect abstraction of $\mathbf{lastTime}$.

Sliding Windows



Abstractions for Sliding Windows



Abstractions for Sliding Windows

The screenshot displays the TeSSLa IDE interface with the following components:

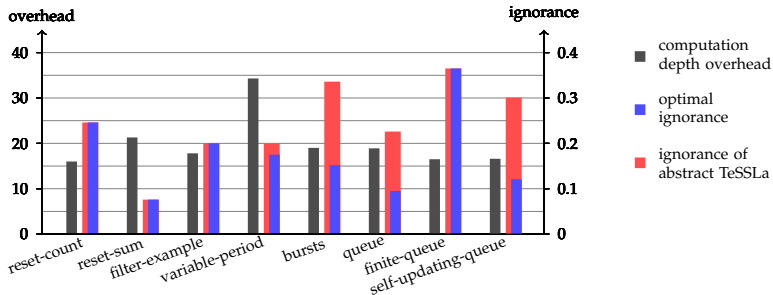
- Trace:** A list of 10 numbered steps showing the state of variables `ld` and `l`.

```
1 0: ld = true
2 1: l = 300
3 3: l = 500
4 7: l = 700
5 8: ld = false
6 9: ld = true
7 10: l = 400
8 13: l = 300
9 16: l = 800
10
```
- Concrete Specification:** A code block defining `qLen`, `stripped`, `enqueue`, `queue`, and `enq` functions.

```
7 def qLen := 5
8
9 def stripped: EventsA[QueueA[Int]] :=
10   sllfca
11     timeA(load),
12     mergeA(lastA(queue, load), sigAT(queueA_empty[Int])),
13     fun(tOpt: Option[Int], q: Option[QueueA[Int]]) =>
14       if isNone(tOpt) then None[QueueA[Int]] else Some({
15         def t := getSome(tOpt)
16         return remOrderA(t - qLen, remReverseA(t, q))
17       })
18
19 def queue: EventsA[QueueA[Int]] :=
20   liftTot3AT(timeA(load), load, stripped,
21   fun(t: Int, d: Int, q: QueueA[Int]) => enqA(t, d, q))
```
- Status and Compiler Output:** An empty text area with a small '1' at the top left.
- TeSSLa Output:** A visualization of the sliding window abstraction. It shows a timeline from 0.0 to 6.5 with values 300, 500, 700, 400, 300, 800. Below this, it shows the state of `ld` (true/false) and `a` (true/false) over time, with green bars indicating true states and red circles indicating false states.

tessla-a.isp.uni-luebeck.de

Empirical Results



Conclusion

1. We replace the basic operators of stream processing with abstract counterparts. We obtain a framework where we can specify with respect to complete traces and automatically evaluate for partially known traces.
2. The abstract operators can be encoded in TeSSLa, allowing existing software- and hardware-based evaluation engines to be reused.
3. The sliding window example demonstrates how complex data structures like queues can be abstracted.
4. Evaluating the abstract specification typically only increases the computational cost by a constant factor.



`www.tessla.io`