UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

isp

Runtime Verification for Timed Event Streams with Partial Information

Martin Leucker[1] César Sánchez[2] Torben Scheffel[1] **Malte Schmitz[1]**
Daniel Thoma[1]

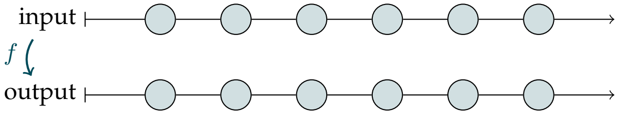
[1] Institute for Software Engineering and Programming Languages,
University of Lübeck, Germany

[2] IMDEA Software Institute, Spain

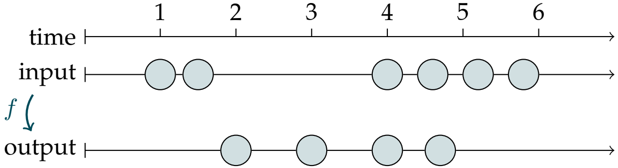
The 19th International Conference on Runtime Verification

Event Sequence

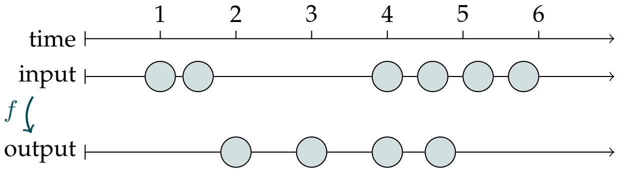
Stream Transformation



Non-Synchronized Stream Transformation



Non-Synchronized Stream Transformation



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

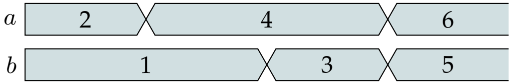
can be represented as

recursive equation system of stream transforming functions:

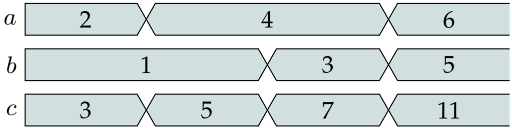
- **lift**
  lifts functions on values to functions on streams
- **last**
  gives access to values of previous events
- **delay**
  generates events with additional timestamps

TeSSLa

Stream Processing by Example

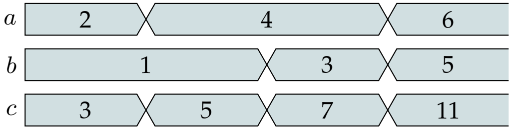


Stream Processing by Example

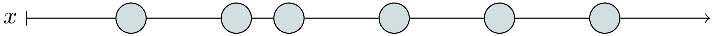


$$c = a + b$$

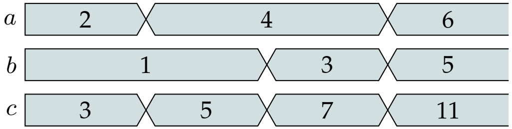
Stream Processing by Example



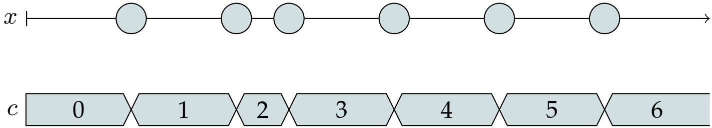
$$c = a + b$$



Stream Processing by Example

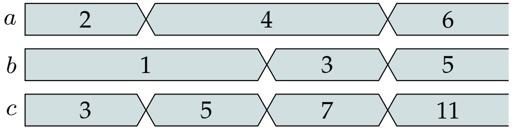


$$c = a + b$$

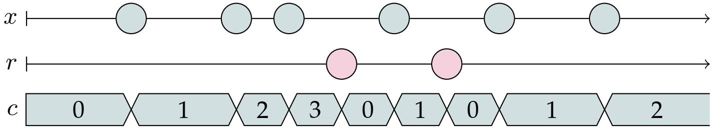


$$c = count(x)$$

Stream Processing by Example



$$c = a + b$$

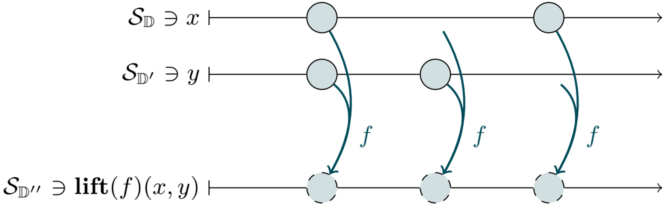


$$c = count(x, r)$$

Lift

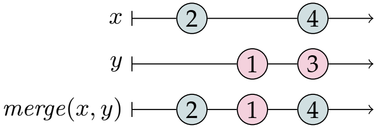
$$\mathbf{lift} : (\mathbb{D}_\perp \times \mathbb{D}'_\perp \to \mathbb{D}''_\perp) \to (\mathcal{S}_\mathbb{D} \times \mathcal{S}_{\mathbb{D}'} \to \mathcal{S}_{\mathbb{D}''})$$

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



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.



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

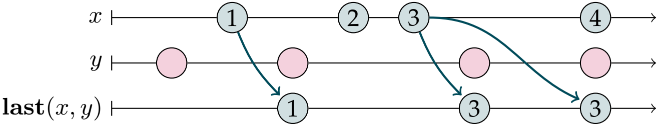
$$f : \mathbb{D}_\perp \times \mathbb{D}_\perp \to \mathbb{D}_\perp$$

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

Accessing Previous Events: Last

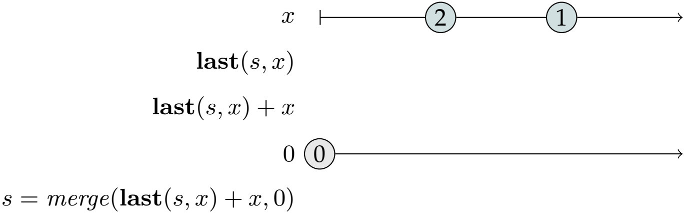
- Needed to define properties over sequences of events.
- Last refers to values of events on one stream
  occurring strictly before events on another stream

$$\mathbf{last} : \mathcal{S}_{\mathbb{D}} \times \mathcal{S}_{\mathbb{D}'} \to \mathcal{S}_{\mathbb{D}}$$



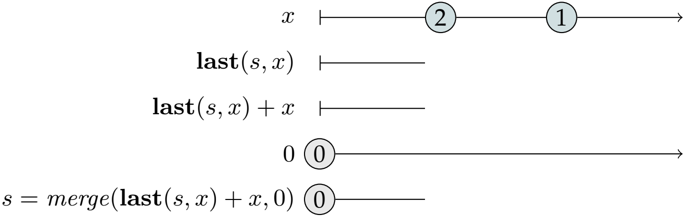
Recursive Equations in TeSSLa

- 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.



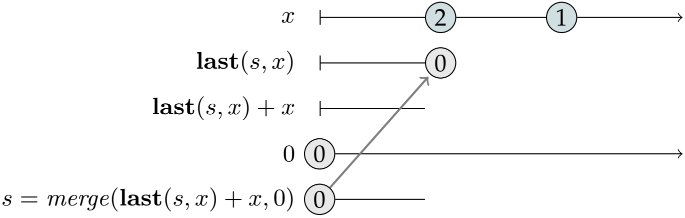
Recursive Equations in TeSSLa

- 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.



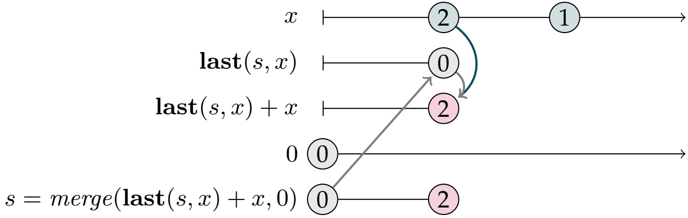
Recursive Equations in TeSSLa

- 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.



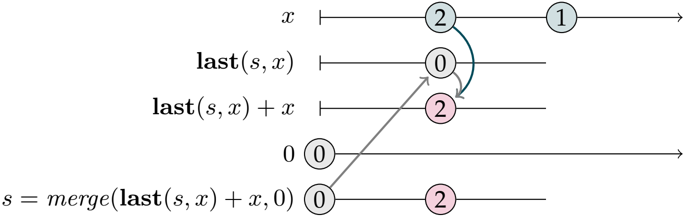
Recursive Equations in TeSSLa

- 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.



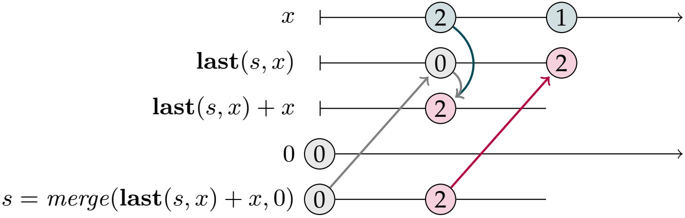
Recursive Equations in TeSSLa

- 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.



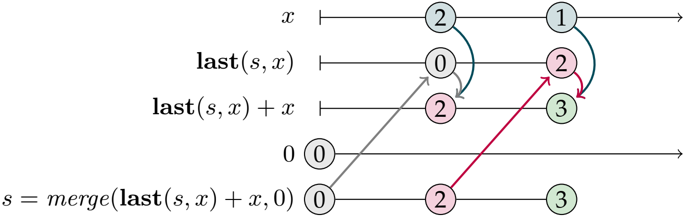
Recursive Equations in TeSSLa

- 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.



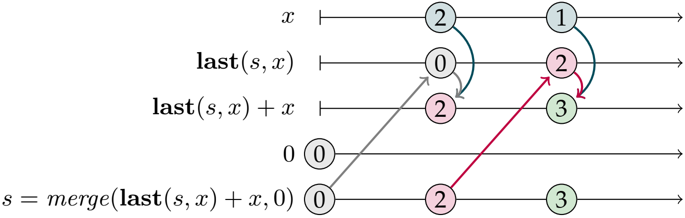
Recursive Equations in TeSSLa

- 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

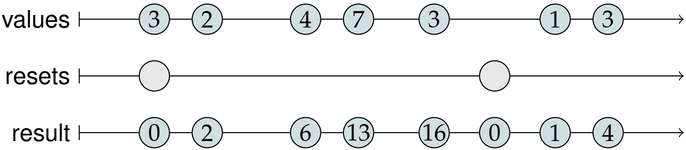
- The last operator allows to write recursive equations
- The merge operation allows to initialize recursive equations with an initial event from an other stream.
- Express aggregation operations like the sum over all values of a stream.



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.

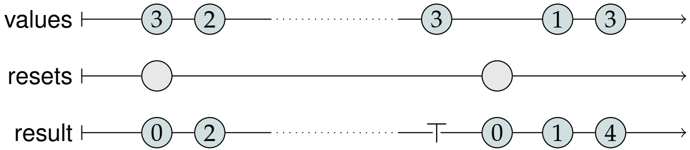
$$\text{result} = sum(\text{values}, \text{resets})$$



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.

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



Abstract Lift

- $\mathbf{lift}^{\#}(f^{\#})(s_1, \ldots, s_n)$ is similar to $\mathbf{lift}(f)(s_1, \ldots, s_n)$
- $f^{\#} : {\mathbb{D}_1}_{\perp}^{\#} \times \cdots \times {\mathbb{D}_n}_{\perp}^{\#} \to \mathbb{D}_{\perp}^{\#}$ is abstraction of $f : {\mathbb{D}_1}_{\perp} \times \cdots \times {\mathbb{D}_n}_{\perp} \to \mathbb{D}_{\perp}$
- $f^{\#}$ takes care of the gaps

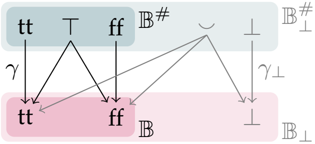
Abstract Lift

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

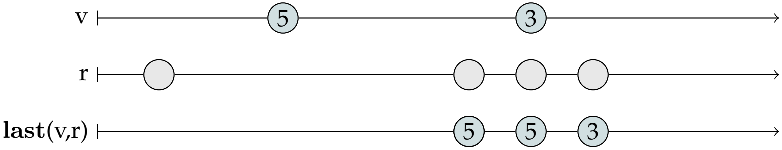
If $\mathbb{D}^{\#}$ is a data abstraction of $\mathbb{D}$
with an associated concretisation function γ,

then $\mathbb{D}_{\bot}^{\#} = \mathbb{D}^{\#} \cup \{\bot, \smile\}$ is a data abstraction of $\mathbb{D}_{\bot}$
with an associated concretisation function $\gamma_{\bot} : \mathbb{D}_{\bot}^{\#} \to 2^{\mathbb{D} \cup \{\bot\}}$:

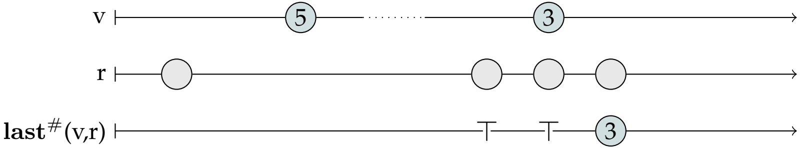
$$\gamma_{\bot}(d) = \begin{cases} \bot & \text{if } d = \bot \\ \mathbb{D} \cup \{\bot\} & \text{if } d = \smile \\ \gamma(d) & \text{if } d \in \mathbb{D}^{\#} \end{cases}$$



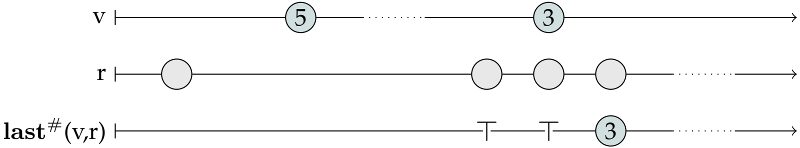
Last



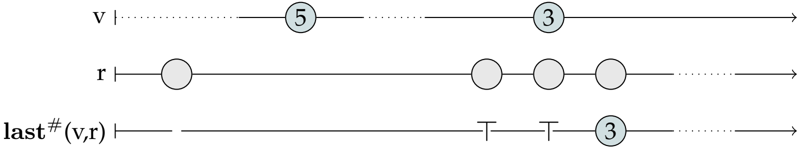
Abstract Last



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.

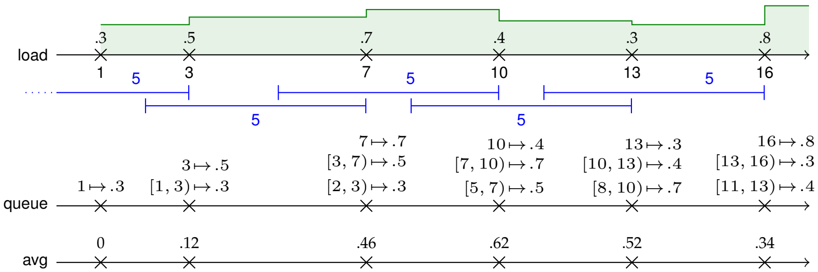
Theorem

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

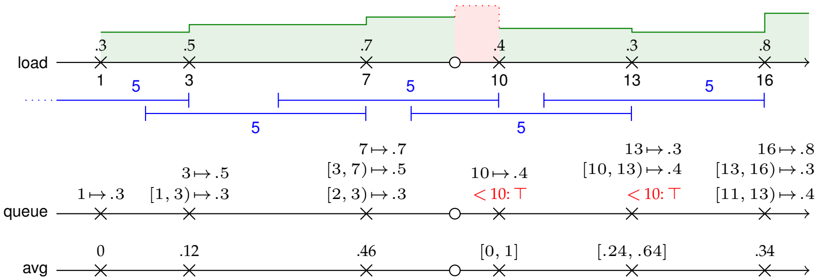
Theorem

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

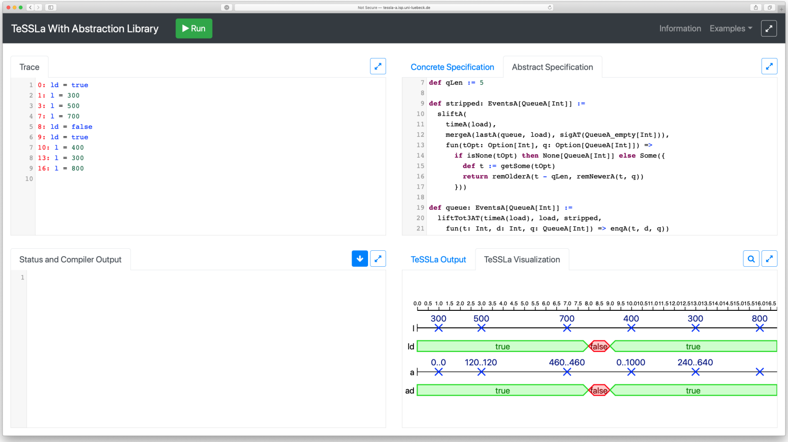
Sliding Windows



Abstractions for Sliding Windows

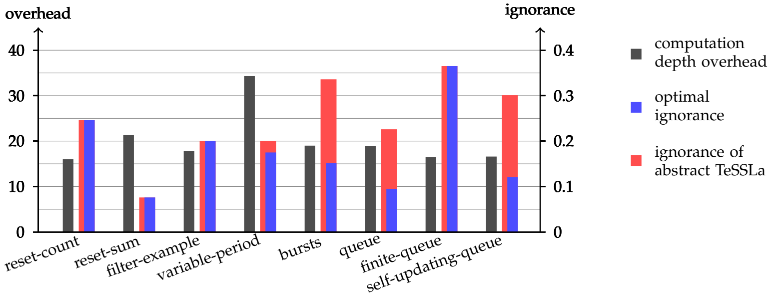


Abstractions for Sliding Windows



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.

TeSSLa

www.tessla.io