UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

isp

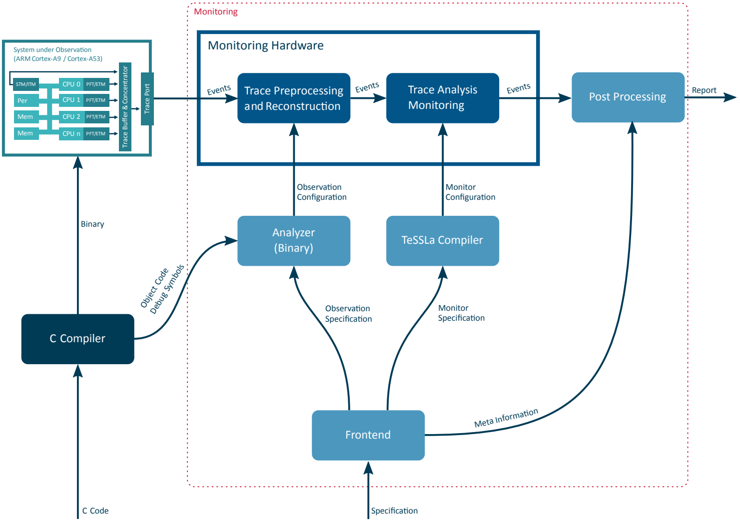
TeSSLa: Temporal Stream-based Specification Language

Lukas Convent Sebastian Hungerecker Torben Scheffel
Malte Schmitz **Daniel Thoma**

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

The 21th Brazilian Symposium on Formal Methods

Motivation: Monitoring Program Flow Trace



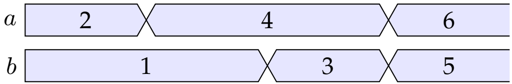
Stream Processing: Technical Prerequisites

- Multi-core CPUs generate large amounts of trace data.
  $\Longrightarrow$ Perform monitoring in hardware.
- FPGAs have limited amount of memory.
  $\Longrightarrow$ Explicit memory usage. Constant memory usage per operator.
- Properties and analyses might become very complex.
  $\Longrightarrow$ Combined monitoring on hardware and in software.
- Timing is crucial in embedded and cyber-physical systems.
  $\Longrightarrow$ Support time as first-class citizen.
- Properties and analyses might require data.
  $\Longrightarrow$ Support analyses and aggregation of data values.

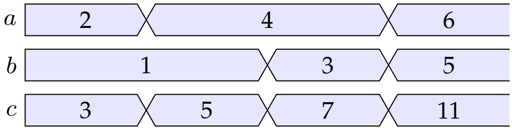
TeSSLa Design Goals

- **Declarative** style: Specification rather than implementation
- **Modularity**: Allowing abstractions based on **few primitives**
- **Time** as first-class citizen
- Abstractions for both **events** and **signals**
- **Recursion** to reason about past
- Implementable with **limited memory**
- Handle both **sparse** and fine **grained** event streams simultaniously

TeSSLa by Example

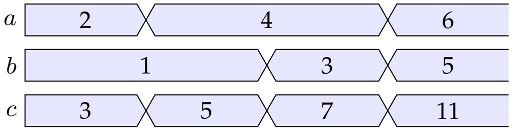


TeSSLa by Example

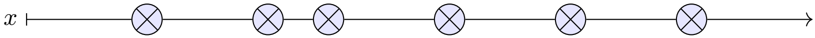



```
def c := a + b
```

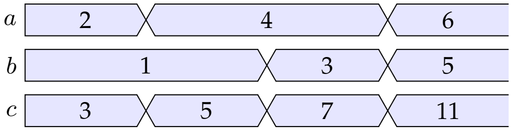
TeSSLa by Example



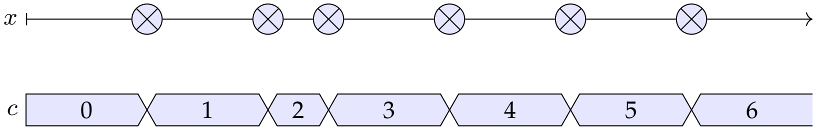

```
def c := a + b
```



TeSSLa by Example

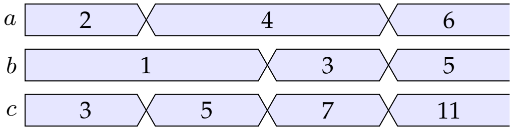



```
def c := a + b
```

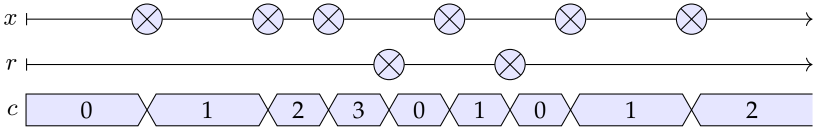



```
def c := eventCount(x)
```

TeSSLa by Example




```
def c := a + b
```




```
def c := eventCount(x, reset = r)
```

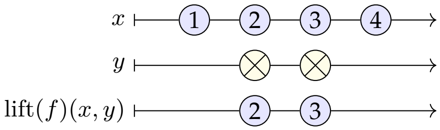
Syntax

$$
\begin{aligned}
e ::= &\ \mathbf{nil} \mid \mathbf{unit} \mid x \mid \\
&\ \mathbf{lift}(f)(e, \ldots, e) \mid \\
&\ \mathbf{time}(e) \mid \\
&\ \mathbf{last}(e, e) \mid \\
&\ \mathbf{delay}(e, e)
\end{aligned}
$$

Core Operators: Lift

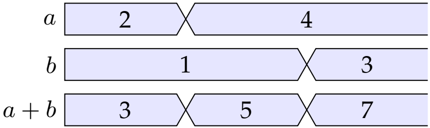
- How to do point-wise operations on streams?

```
def f[A, B](a: Option[A], b: Option[B]): Option[A] :=
  if(isDefined(b)) then a else None
```



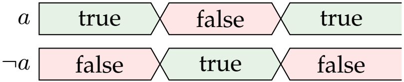
Derived Operators: Signal Lift of Addition

- *Signal lift* allows to lift operations on arbitrary data types to streams.
- E.g. the *addition* on integer numbers can be lifted to streams of integers.



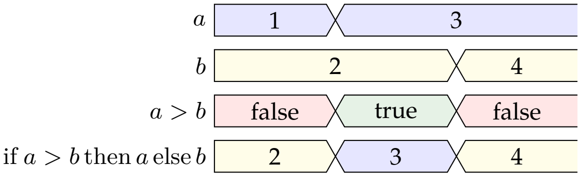
Derived Operators: Signal Lift of Negation

- *Signal lift* allows to lift operations on arbitrary data types to streams.
- E.g. the *negation* of booleans can be lifted to a stream of booleans.



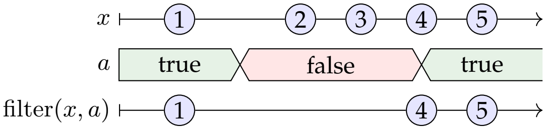
Derived Operators: Signal Lift of If-Then-Else

- *Signal lift* allows to lift operations on arbitrary data types to streams.
- E.g. the ternary *if-then-else* function can be lifted to a stream of booleans and two streams of identical type.



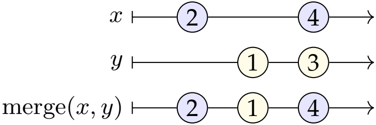
Derived Operators: Filter

- Process streams in an *event-oriented fashion*
- *Filter* the events of one stream based on a second boolean stream interpreted as piecewise constant signal.



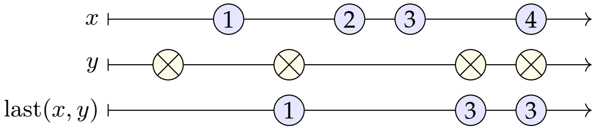
Derived Operators: Merge

- 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.



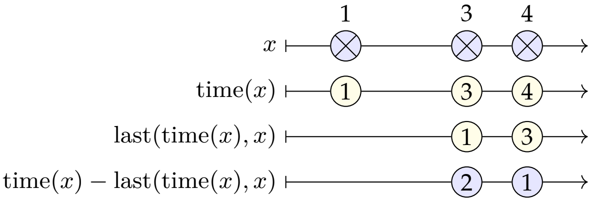
Core Operators: Last

- Needed to define properties over sequences of events.
- Last allows to refer to the values of events on one stream that occurred strictly before the events on another stream



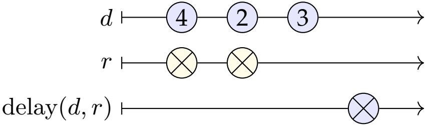
Core Operators: Time

- Provides access to the *timestamps* of events
- Produces events carrying their *timestamps as data value*
- Hence *all operators* for data values can be applied to timestamps.



Core Operators: Delay

- Allows to create new timestamps



Core Operators: Const and Nil

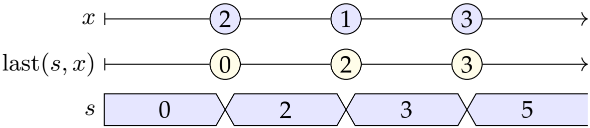
- The constant *nil* for the empty stream
- The operator *const* converting a value to a stream starting with that value at timestamp 0.

Implicit Conversions

- Integer and Boolean constants are converted to streams via *const*.
- Build-in operators on integers and Booleans are lifted to streams via *signal lift*.

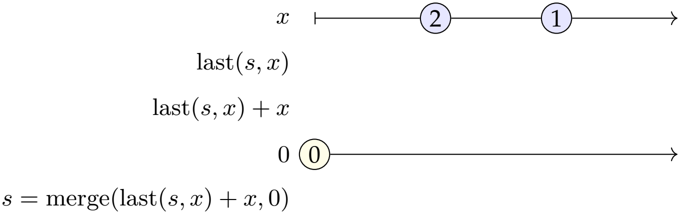
Recursive Equations in TeSSLa

- 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.
- Evaluation algorithm iterates progressing event streams until fixed-point is reached.

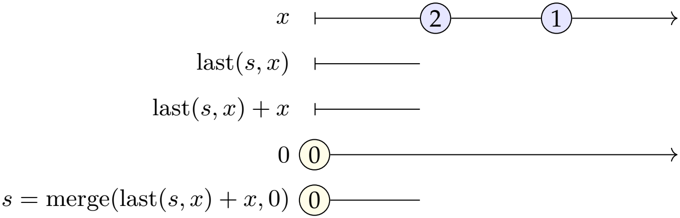



```
def s := merge(last(s, x) + x, 0)
```

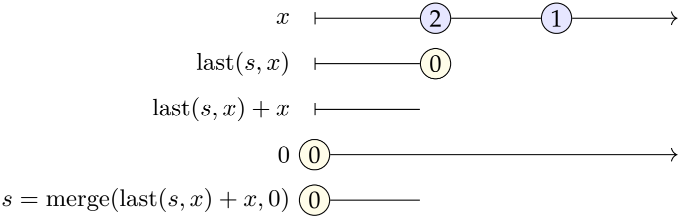
Recursive Equations in TeSSLa: How It Works



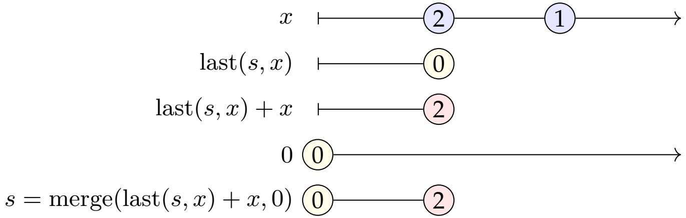
Recursive Equations in TeSSLa: How It Works



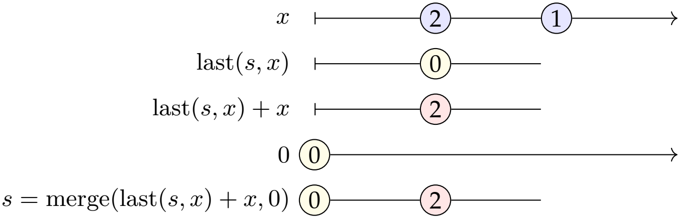
Recursive Equations in TeSSLa: How It Works



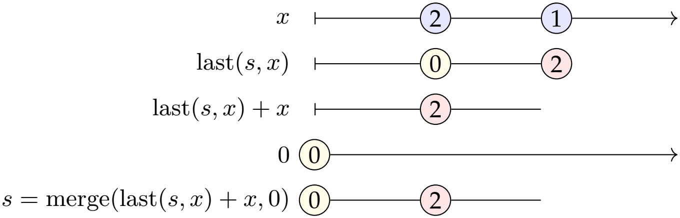
Recursive Equations in TeSSLa: How It Works



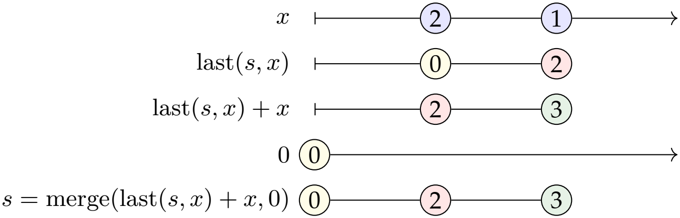
Recursive Equations in TeSSLa: How It Works



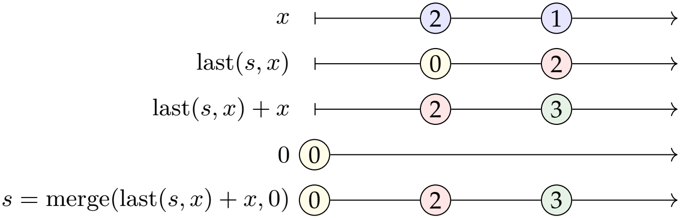
Recursive Equations in TeSSLa: How It Works



Recursive Equations in TeSSLa: How It Works



Recursive Equations in TeSSLa: How It Works



Macros in TeSSLa: eventCount

```
# Count the number of events on `values`.
def eventCount[A,B](values: Events[A]) := {
  def count: Events[Int] := merge(
    # increment counter
    last(count, values) + 1
  , 0)
  count
}
```

Computabilitiy and Well-formed Specifications

- All specifications exhibit a unique least fixed-point
  - Consequence of monotonic and continuous operators
- Computablity:
  - In presence of delay, distances between events might converge
  - We can compute chain of pre-fixed-points, i.e. finite prefixes of output streams
  - Without delay, least fixed-point can always be reached
  - One computation step per operator per progress step
- Well-formed fragement:
  - Recursions guarded by last or delay
  - Only value/delay may be directly recursive
  - For non-well-formed specifications smalles fixed-point undesireable

Properties and Fragments

- Complete TeSSLa is equivalent to
  - continuous and monotone and
  - future independent stream transformations.
- TeSSLa without delay is equivalten to
  - continuous and monotone and
  - future independent and
  - timestamp conservative stream transformations.
- Boolean fragement (inequations on time) is equivalent to finite state transducers
- Timed Fragement (linear constraints on time) is equivalent to deterministic timed transducers

Implementations and Data Types

- JVM based interpreter
- Hardware-implemented interpreter
- Synthesis to FPGA

- TeSSLa is defined agnostically with respect to any time or data domain.
- Different data structures can be used to represent time and data.
- Monitoring in hardware: atomic data types, e.g. int or float.
- Monitoring in software: complex data structures like lists, trees and maps.

Conclusion

- Using TeSSLa we can check
  - event ordering constraints
  - timing constraints
  - complex event patterns
- TeSSLa can be used to aggregate data and compute statistical data
- Can express large classes of stream transformations

- Sparse and fine graned event streams
- Explicit controll over memory requirements

- Relation to well-studied models (finite and timed automata)
- Decidability results for corresponding fragements

- There is a software interpreter (see tessla.io).
- Very small core language (2-3 operators).
- Open to many other applications
- Suited for hardware-based implementations