# TESSLA—A Temporal Stream-based Specification Language

March 23, 2016

## 1 Introduction

Outline

1. purpose, motivation
    - online processing of data
    - monitoring of trace properties, specifically execution traces of programs
    - Functional reactive programming as related concept
2. What you describe with a TeSSLa specification
    - input, output streams
    - application of functions, composition of function
    - 
3. Modelling data in terms of streams
    - timing model ($\mathbb{R}$,$\mathbb{N}$,$\mathbb{Q}$,...), restrictions to streams with discrete set of time stamps (event streams) or peace-wise constant streams (continuous stream)
    - continuous streams and event streams
4. Functions on streams and desired properties (in general)
    - small examples
    - causality, statefulness, time invariance
    - (composition lemmata)
5. TeSSLa syntax
    - base grammar with functions and type annotations
    - syntactical extensions: infix operators, named arguments, the "on"

6. Types
    - Generic types
    - Coercion
7. Functional semantics of operators, small examples
8. Larger example/case study
    - producer/consumer, ring buffer, . . .

Purpose of TeSSLa.

- analysis of trace data
- specification of failure patterns, correctness properties, transformations
- intuitive, pragmatic means of formulation

Purpose of this document.

- Motivate and describe the language
- reference
- case study and examples from the targeted application area
- description of how to integrate runtime verification methodology based on tessla and its implementation into the development process

Approach

- online processing of data
- monitoring of trace properties, specifically execution traces of programs
- Functional reactive programming as related concept

## 1.1 What you describe with a TeSSLa specification

TeSSLa is conceptually based on streams as a model for data processing and data analysis. The data to be analysed is considered as input streams and a TeSSLa specification essentially describes a set of output streams and how they can be derived from the input. To this end, new streams can be defined by applying some function to existing ones. For example, given an input stream of integral values, a TeSSLa specification could describe the stream that consecutively provides the sum of all previous input values. This is achieved by applying a corresponding function to the input stream and thereby defining a new output stream.

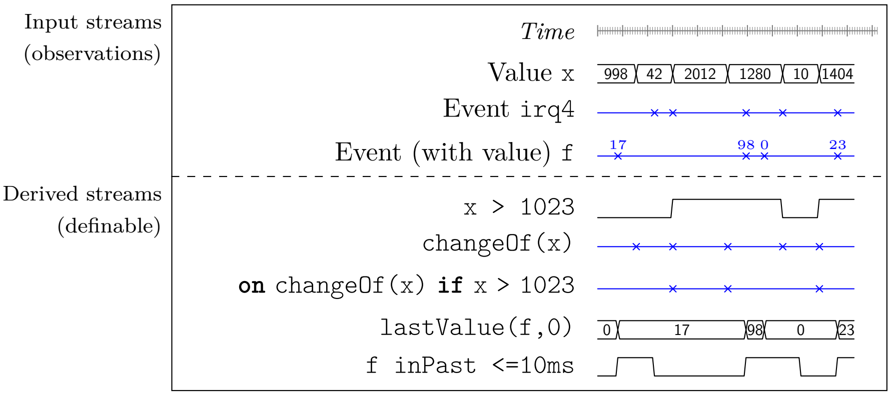

Figure 1: Example streams.

## 1.2 Stream Model

The streams used in TeSSLa specifications model the essential aspects found in computer programs, namely values (e.g., the value of a program variable), events (e.g., the call of a specific function) and time, both, in a qualitative (ordering) and quantitative (duration) sense.

The timing model is based on time stamps $t \in \mathbb{T}$ where we assume $\mathbb{T}$ to be isomorphic to the real numbers $\mathbb{R}$. Although on the technical level time is mosltly quantised in actual systems, real time is a common and intuitive model. In fact, neither specification nor evaluation based on single steps of, e.g., the a CPU core are reasonable. Time is therefore handled, formally and technically, in terms of intervals. In the following, we make the notion of streams precise that prolide the semantical basis of the language. We use $\mathbb{T}$ to denote the time domain to make an explicit distinction between time stamps and, e.g., real values. However, we use common operators and symbols to work with time stamps, such as $+$ (addition), $\leq$ (ordering) and $0$ (neutral element of addition), that are defined as expected.

We consider two types of streams. Value, e.g., of a program variable or stored at some specific memory address, might change over time but can be assumed to always be present. They are represented by continuous streams that we call *signals.*

adde preliminary definitions somewhere, e.g., appendix: piece-wise constant function, segments, intervals, left/right-closed, change points

**Definition 1** (Signals)**.** *Let $D$ be a set of data values. A* signal of type $D$ *is a function $\sigma : \mathbb{T}_{\geq 0} \to D$ such that*

- *$\sigma$ is piece-wise constant,*
- *every segment $I \in \mathsf{seg}(\sigma)$ is left-closed*[1] *and*
- *the set of change points $\Delta(\sigma) := \{\min I \mid I \in \mathsf{seg}(\sigma)\}$ is discrete*[2]*.*

[1] A *segment* of a piece-wise constant function $\sigma : \mathbb{T}_{\geq 0} \to D$ is a maximal interval $I \subset \mathbb{T}$ with constant value $v \in D$, i.e., $\forall_{t \in I} : \sigma(t) = v$.

[2] A subset $M$ of $\mathbb{T}$ is *discrete* if it does not contain bounded infinite subsets.

*The set of all signals $\sigma : \mathbb{T}_{\geq 0} \to D$ is denoted $\mathcal{S}_D$.*

Apart from values that are conveniently modeled to be continuously available, discrete *events*, like function calls, are of interest. These are modelled by the second type of streams *event streams* that provide values (events) only at specific points in time wheras no information about the time between two consecutive events is available.

**Definition 2** (Event streams)**.** *For a set $D$ of data values an* event stream of type $D$ *is a partial function $\eta : \mathbb{T}_{\geq 0} \rightharpoonup D$ such that the set of domain of definition, the* event points $E(\eta) := \{t \in \mathbb{T} \mid \eta(t) \in D$ *defined$\}$, is discrete.*

*The set of event streams $\eta : \mathbb{T}_{\geq 0} \to D \mathbin{\dot\cup} \{\bot\}$ is denoted $\mathcal{E}_D$.*

The discreteness condition reflects the property of actual systems that time stamps cannot converge because only a bounded number of events can happen within a fixed time period.

In addition to the definition in terms of partial functions, an event stream $\eta \in \mathcal{E}_D$ can be naturally represented as a (possibly infinite) sequence $s_\eta = (t_0, \eta(t_0))(t_1, \eta(t_1)) \ldots \in (E(\eta) \times D)^\infty$ ordered by time ($t_i < t_{i+1}$ for $0 \leq i < |s_\eta|$) and containing all event points ($\{t \mid (t, v)$ occurs on $s\} = E(\eta)$).

## 1.3 Defining Streams

TeSSLa allows for defining streams through function applications. Such functions can be applied to signal, event streams, as well as constant values. For example, addition of two (value) streams can be defined as element-wise addition of the values of two signals `s1` and `s2`:

```
define sum := add(in1, in2)
```

Here, $\texttt{add} : \mathcal{S}_\mathbb{N} \times \mathcal{S}_\mathbb{N} \to \mathcal{S}_\mathbb{N}$ is a function that maps a pair of signals $\texttt{in1}, \texttt{in2} \in \mathcal{S}_\mathbb{N}$ with data domain $\mathbb{N}$ to the signal representing their sum at every point in time, i.e., $\texttt{sum}(t) = \texttt{add}(\texttt{in1}, \texttt{in2})(t) = \texttt{in1}(t) + \texttt{in2}(t)$ for any time point $t \in \mathbb{T}$.

The specification above hence describes a rather simple transformation of two input streams into one output stream.

Regarding evaluation it is reasonable, to restrict the functions on streams that can be used in TeSSLa, depending on their application context.

**Definition 3** (Causality, state, time invariance)**.** *Let $A, B$ be sets of signals or event streams. A functions $f : A \to B$ is considered to* respect weak causality *if there is a constant $k \in \mathbb{T}$ such that $f(\sigma)(t)$ is independent of the values $\sigma(t')$ for $t' > t + k$: for all $t \in \mathbb{T}$ and all $\sigma, \sigma' \in A$ we require that $f(\sigma)(t) = f(\sigma')(t)$ if $\sigma(t') = \sigma'(t')$ for all $t' < t + k$.*

*The function $f$ is called* stateless *if for all $t \in \mathbb{T}$ and all $\sigma, \sigma' \in A$ we have $f(\sigma)(t) = f(\sigma')(t)$ if $\sigma(t) = \sigma'(t)$.*

*A stateless function $f$ is called* time invariant *if $\sigma(t) = \sigma(t')$ implies that $f(\sigma)(t) = f(\sigma)(t')$ for all $\sigma \in A$ and all $t \in \mathbb{T}_{\geq 0}$.*

Example: Detecting a delayed action

Assume the control of a device is supposed to react on an input signal within a specific time bound of 10 microseconds. The control program receives a signal when the function `rcv()` is called, needs to process the data and react by calling a function `react()`. Given means to observe function calls during the execution of the program[a] a TeSSLa specification can be used to formulate the timing constraint. The calls to `rcv()` and `react()` can be considered as input event streams `rcv` and `react`.

[a] We will discuss observation approaches in Section **??**.

# 2 Syntax

This section describes the syntax of TeSSLa.

## 2.1 Basic Syntax

The basic syntax of TeSSLa specification is by the following grammar.

$$\begin{aligned}
\mathit{spec} &::= \mathtt{define}\ \mathit{name}[\mathtt{:}\ \mathit{stype}]\ \mathtt{:=}\ \mathit{texpr} \mid \\
&\phantom{::=}\ \mathtt{out}\ \mathit{name} \mid \mathit{spec}\ \mathit{spec} \\
\mathit{texpr} &:= \mathit{expr}[\mathtt{:}\ \mathit{type}] \\
\mathit{expr} &:= \mathit{name} \mid \mathit{literal} \mid \mathit{name}\mathtt{(}\mathit{texpr}(\mathtt{,}\ \mathit{texpr})^{*}\mathtt{)} \\
\mathit{type} &:= \mathit{btype} \mid \mathit{stype} \\
\mathit{stype} &:= \mathtt{Signal<}\mathit{btype}\mathtt{>} \mid \mathtt{Events<}\mathit{btype}\mathtt{>}
\end{aligned}$$

Names are nonempty strings $\mathit{name} \in AB*$ where $A = \{\mathtt{A}, \dots, \mathtt{Z}, \mathtt{a}, \dots, \mathtt{z}\}$ are the alphanumeric characters and $B = A \cup \{\_\}$. Basic types *btype* cover typical ones such as **`Int`**, **`Float`**, **`String`** or **`Bool`**. Literals *literal* denote explicit values, of basic types, such as integers $-1, 0, 1, 2, \dots,$ floating point numbers $0.1, -3.141593$ or strings (enclosed in double quotes). Available basic types and literal representation are implementation dependent.

## 2.2 Syntactical Extensions

For convenience, we consider three additional syntactical elements: **`on`**-*comprehensions*, *infix notation for binary operators* and *named arguments*.

### 2.2.1 On-comprehension

Syntax:

$$\begin{aligned}
\mathit{oncomp} &::= \mathtt{on}\ \mathit{triggers}[\ \mathtt{if}\ \mathit{filterExpr}][\ \mathtt{yield}\ \mathit{valueExpr}] \\
\mathit{triggers} &::= \mathit{name}(\mathtt{,}\ \mathit{name})^{*}
\end{aligned}$$

The *triggers* are a list of names denoting event streams. The filtering Expression $filterExpr$ is an expression of type `Signal<Bool>` where every free name either occurs in the trigger list or denotes a signal. Intuitively, the on-comprehension emits an event at those time points $t \in \mathbb{T}$ where all of the trigger streams emit an event (i.e., are defined) and the filter signal has value `true`. If the `yield` part is omitted, the events do not carry a value, i.e., the stream is of type `Events<Unit>`. Otherwise, the value expression *valueExpr* defines the value of every event. As for the filer expression, in can only contain free names that either occur in the trigger list or are signals.

All functions used in the filter and value expressions are further required to be stateless.

### 2.2.2 Named Arguments

as expected

### 2.2.3 Infix Operators

as expected

# 3 Semantics

The formal semantics of a TeSSLa specification is a function mapping a set of input streams to a set of output streams.

The set of output streams consists of all streams that are explicitly defined in the specification. The set of input streams is defined implicitly by the set of names (and their type) denoting a stream that occur freely in the specification, i.e., without definition.

That way, the semantics of the specification is build from (and depends on) the semantics of the functions used in the specification. In the following we describe a set of convenient functions that could be considered as a „standard library“.

## 3.1 Lifted Functions

A function $f : D_1 \times \ldots \times D_n \to D_{n+1}$ on basic types can easily be lifted to a function $\hat{f} : \mathcal{S}_{D_1} \times \ldots \times \mathcal{S}_{D_n} \to \mathcal{S}_{D_{n+1}}$ on signal with $\hat{f}(\sigma_1, \ldots, \sigma_n)(t) = f(\sigma_1(t), \ldots, \sigma_n(t))$ for all $t \in \mathbb{T}$. This is possible since signals provide a value at every time point.

We list some important lifted functions for arithmetics and boolean operations. They are defined as expected in terms of their scalar counterparts as above.

| Function name, signature | Semantics | Remark |
|---|---|---|
| $\texttt{add} : \mathcal{S}_D \times \mathcal{S}_D \rightarrow \mathcal{S}_D$ | $\texttt{add}(\sigma_1, \sigma_2)(t) := \sigma_1(t) + \sigma_2(t)$ | $D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\}$ |
| $\texttt{sub} : \mathcal{S}_D \times \mathcal{S}_D \rightarrow \mathcal{S}_D$ | $\texttt{sub}(\sigma_1, \sigma_2)(t) := \sigma_1(t) - \sigma_2(t)$ | $D \in \{\mathbb{Z}, \mathbb{R}\}$ |
| $\texttt{mul} : \mathcal{S}_D \times \mathcal{S}_D \rightarrow \mathcal{S}_D$ | $\texttt{mul}(\sigma_1, \sigma_2)(t) := \sigma_1(t) \cdot \sigma_2(t)$ | $D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\}$ |
| $\texttt{geq} : \mathcal{S}_D \times \mathcal{S}_D \rightarrow \mathcal{S}_\mathbb{B}$ | $\texttt{geq}(\sigma_1, \sigma_2)(t) := \sigma_1(t) \geq \sigma_2(t)$ | $D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\}$ |
| $\texttt{leq} : \mathcal{S}_D \times \mathcal{S}_D \rightarrow \mathcal{S}_\mathbb{B}$ | $\texttt{leq}(\sigma_1, \sigma_2)(t) := \sigma_1(t) \leq \sigma_2(t)$ | $D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\}$ |
| $\texttt{eq} : \mathcal{S}_D \times \mathcal{S}_D \rightarrow \mathcal{S}_\mathbb{B}$ | $\texttt{eq}(\sigma_1, \sigma_2)(t) := \sigma_1(t) = \sigma_2(t)$ | any $D$ with equality |
| $\texttt{max} : \mathcal{S}_D \times \mathcal{S}_D \rightarrow \mathcal{S}_D$ | $\texttt{max}(\sigma_1, \sigma_2)(t) := \max\{\sigma_1(t), \sigma_2(t)\}$ | $D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\}$ |
| $\texttt{min} : \mathcal{S}_D \times \mathcal{S}_D \rightarrow \mathcal{S}_D$ | $\texttt{min}(\sigma_1, \sigma_2)(t) := \min\{\sigma_1(t), \sigma_2(t)\}$ | $D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\}$ |
| $\texttt{and} : \mathcal{S}_\mathbb{B} \times \mathcal{S}_\mathbb{B} \rightarrow \mathcal{S}_\mathbb{B}$ | $\texttt{and}(\sigma_1, \sigma_2)(t) := \sigma_1(t) \wedge \sigma_2(t)$ | |
| $\texttt{or} : \mathcal{S}_\mathbb{B} \times \mathcal{S}_\mathbb{B} \rightarrow \mathcal{S}_\mathbb{B}$ | $\texttt{and}(\sigma_1, \sigma_2)(t) := \sigma_1(t) \vee \sigma_2(t)$ | |
| $\texttt{implies} : \mathcal{S}_\mathbb{B} \times \mathcal{S}_\mathbb{B} \rightarrow \mathcal{S}_\mathbb{B}$ | $\texttt{and}(\sigma_1, \sigma_2)(t) := \sigma_1(t) \Rightarrow \sigma_2(t)$ | |
| $\texttt{not} : \mathcal{S}_\mathbb{B} \rightarrow \mathcal{S}_\mathbb{B}$ | $\texttt{not}(\sigma_1)(t) := \neg\sigma_1(t)$ | |

## 3.2 Timing Functions

The function delay shifts a stream by a specific amount of time. We define the function for different signatures and any value domain $D$:

$$\texttt{delay} : \mathcal{S}_D \times \mathbb{T} \times D \rightarrow \mathcal{S}_D \qquad \texttt{delay}(\sigma, d, v)(t) := \begin{cases} \sigma(t-d) & \text{if } t - d \geq 0 \\ v & \text{otherwise} \end{cases}$$

$$\texttt{delay} : \mathcal{S}_D \times \mathbb{T}_{\leq 0} \rightarrow \mathcal{S}_D \qquad \texttt{delay}(\sigma, d)(t) := \sigma(t-d)$$

$$\texttt{delay} : \mathcal{E}_D \times \mathbb{T} \rightarrow \mathcal{E}_D \qquad \texttt{delay}(\eta, d)(t) := \begin{cases} \sigma(t-d) & \text{if } t - d \geq 0 \\ \bot & \text{otherwise} \end{cases}$$

The function timestamp provides the time stamp of an event stream element-wise:

$$\texttt{timestamp} : \mathcal{E}_D \rightarrow \mathcal{E}_\mathbb{T} \qquad \texttt{timestamp}(\eta)(t) := \begin{cases} t & \text{if } t \in E(\eta) \\ \bot & \text{otherwise} \end{cases}$$

The function within serves for checking whether an event occurs within a given (relative) time bound. Further functions inPast and inFuture can be derived from *within*:

$$\texttt{within} : \mathbb{T} \times \mathbb{T} \times \mathcal{E}_D \rightarrow \mathcal{S}_\mathbb{B} \quad \texttt{within}(d_1, d_2, \eta)(t) := \begin{cases} \texttt{true} & \text{if } E(\eta) \cap [t + d_1, t + d_2] \neq \emptyset \\ \texttt{false} & \text{otherwise} \end{cases}$$

$$\texttt{inPast} : \mathbb{T}_{\geq 0} \times \mathcal{E}_D \rightarrow \mathcal{S}_\mathbb{B} \qquad \texttt{inPast}(d, \eta)(t) := \mathit{within}(-d, 0, \eta)(t)$$

$$\texttt{inFuture} : \mathbb{T}_{\geq 0} \times \mathcal{E}_D \rightarrow \mathcal{S}_\mathbb{B} \qquad \texttt{inFuture}(d, \eta)(t) := \mathit{within}(0, d, \eta)(t)$$

TODO
The function synchronise matches events from two streams within a given time range.

$$\texttt{synchronise} : \mathcal{E}_D \times \mathcal{E}_D \times \mathbb{T} \to$$

## 3.3 Aggregations

For any domain $D$ with linear ordering and addition, respectively, e.g., $\mathbb{N}, \mathbb{Z}, \mathbb{R}, \mathbb{Q}$:

$$\begin{aligned}
\texttt{maximum} : \mathcal{E}_D \times D \to \mathcal{S}_D &\quad \texttt{maximum}(\eta, d)(t) := \max(\{d\} \cup \{\eta(t') \mid t' \in E(\eta), t' \leq t\}) \\
\texttt{maximum} : \mathcal{S}_D \to \mathcal{S}_D &\quad \texttt{maximum}(\sigma)(t) := \max\{\eta(t') \mid t' \in \mathbb{T}, t' \leq t\} \\
\texttt{minimum} : \mathcal{E}_D \times D \to \mathcal{S}_D &\quad \texttt{minimum}(\eta, d)(t) := \min(\{d\} \cup \{\eta(t') \mid t' \in E(\eta), t' \leq t\}) \\
\texttt{minimum} : \mathcal{S}_D \to \mathcal{S}_D &\quad \texttt{minimum}(\sigma)(t) := \min\{\eta(t') \mid t' \in \mathbb{T}, t' \leq t\} \\
\texttt{sum} : \mathcal{E}_D \to \mathcal{S}_D &\quad \texttt{sum}(\eta)(t) := \sum_{t' \in E(\eta) \mid t' \leq t} \eta(t')
\end{aligned}$$

Generic functions *eventCount* providing the number of occurred events and *mrv* providing the most recent value of an event stream.

$$\begin{aligned}
\texttt{eventCount} : \mathcal{E}_D \to \mathcal{S}_D &\quad \texttt{eventCount}(\eta)(t) := |t' \in E(\eta) \mid t' \leq t \\
\texttt{mrv} : \mathcal{E}_D \times D \to \mathcal{S}_D &\quad \texttt{mrv}(\eta, d)(t) := \begin{cases} \eta(\max E(\eta) \cap [0, t]) & \text{if } E(\eta) \cap [0, t] \neq \emptyset \\ d & \text{otherwise} \end{cases}
\end{aligned}$$

## 3.4 Selectors/Filters/Conditionals/Combinators

$$\begin{aligned}
\texttt{ifThen} : \mathcal{E}_{D_1} \times \mathcal{S}_{D_2} \to \mathcal{E}_{D_2} &\quad \texttt{ifThen}(\eta, \sigma)(t) := \begin{cases} \sigma(t) & \text{if } t \in E(\eta) \\ \bot & \text{otherwise} \end{cases} \\
\texttt{ifThenElse} : \mathcal{S}_{\mathbb{B}} \times \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_D &\quad \texttt{ifThenElse}(\sigma_1, \sigma_2, \sigma_3)(t) := \begin{cases} \sigma_2(t) & \text{if } \sigma_1(t) = \texttt{true} \\ \sigma_3(t) & \text{otherwise} \end{cases} \\
\texttt{merge} : \mathcal{E}_D \times \mathcal{E}_D \to \mathcal{E}_D &\quad \texttt{merge}(\eta_1, \eta_2)(t) := \begin{cases} \eta_1(t) & \text{if } t \in E(\eta_1) \\ \eta_2(t) & \text{if } t \in E(\eta_2) \setminus E(\eta_1) \\ \bot & \text{otherwise} \end{cases}
\end{aligned}$$

Note that `ifThen` is a restricted form of an on-comprehension.

## 3.5 Monitors

The monitor function can be used to enable the usage of temporal logics within TeSSLa. A monitor is defined by a temporal logic formula and returns an output value that depends on the evaluation status of the given formula at the current point in time. We assume that the temporal logic is defined over propositional variables from a fixed and finite set $AP = \{p_1, \ldots, p_n\}$, e.g. like LTL. Further, the semantics is defined on finites words over the alphabet $\Sigma = 2^{AP}$ and admits truth values from some domain $\mathbb{V}$ (e.g., $\mathbb{B}$). We then let, for arbitrary $D$,

$$\texttt{monitor} : TL \times (\mathcal{S}_{\mathbb{B}})^n \times \mathcal{E}_D \to \mathcal{E}_{\mathbb{V}}$$

with

$$\texttt{monitor}(\varphi, \sigma_1, \ldots, \sigma_n, \eta)(t) := \begin{cases} [\![ w_t \models \varphi ]\!] & \text{if } t \in E(\eta) \\ \bot & \text{otherwise} \end{cases}$$

where $w_t = a_1 \ldots a_{|E(\eta)|}$ with $a_i = \{p_k \in AP \mid \sigma_k(t_i) = \texttt{true}\}$ for $\{t_1 < t_2 < \ldots < t_{|E(\eta)|}\} = E(\eta)$.