TESSLA—A Temporal Stream-based Specification Language

June 27, 2016

1 Introduction

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 The Setting

1.2 The Development of TeSSLa

1.3 Related Languages

1.4 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

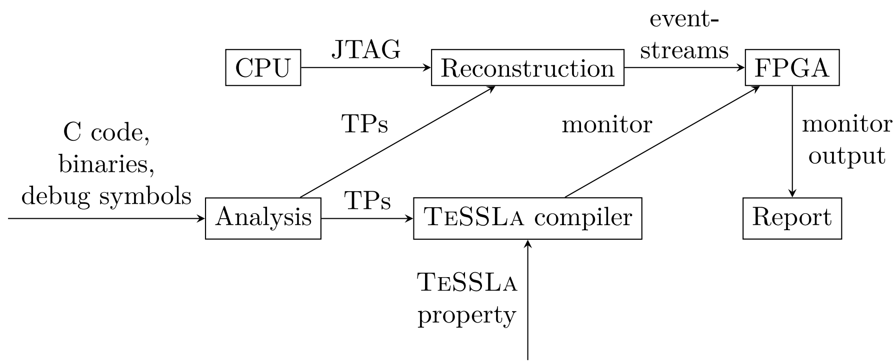

Figure 1: The workflow for reconstructing the program sequence and synthesizing the monitor for a TeSSLa specification.

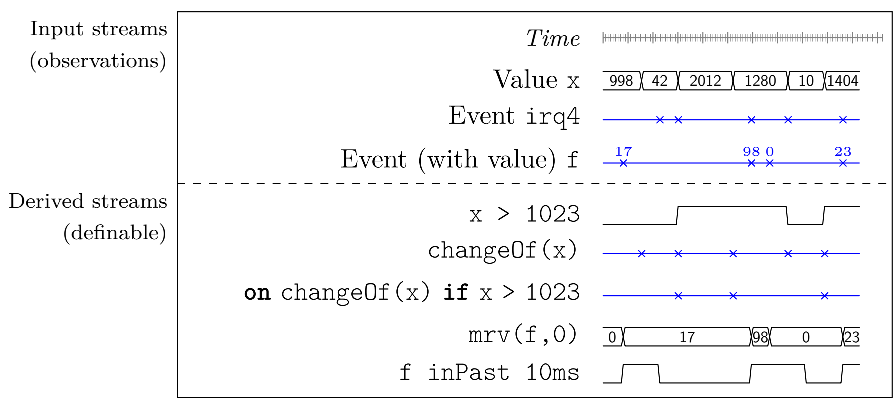

Figure 2: Example streams.

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.

1.5 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 $\mathbb{R}$.[1] Although on the technical level time is mostly 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 provide the semantic 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. This avoids confusion and inconsistencies since the representation of time values is implementation dependent. For example, the values may be scaled according to the processing clock speed and therefore adding or comparing a value $t \in \mathbb{T}$ with some real value $r \in \mathbb{R}$ is not well defined unless considering a specific execution platform with fixed parameters. 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. Values, 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*

- *σ is piece-wise constant,*
- *every segment $I \in \mathsf{seg}(\sigma)$ is left-closed*[2] *and*
- *the set of change points $\Delta(\sigma) := \{\min I \mid I \in \mathsf{seg}(\sigma)\}$ is discrete*[3]*.*

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 domain of definition, called the set of* event points *$E(\eta) := \{t \in \mathbb{T} \mid \eta(t) \in D \text{ defined}\}$, is discrete. The set of all event streams $\eta : \mathbb{T}_{\geq 0} \rightharpoonup D$ is denoted $\mathcal{E}_D$.*

---

[1] Notice that we deliberately choose *the continuum* as a model of real time as is common in philosophy, physics and engineering. For our purposes of specifying timing property, however, also a weaker notion of density would clearly suffice, such as the rational numbers $\mathbb{Q}$.

[2] 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., $\forall_{t \in I} : \sigma(t) = v$.

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

For convenience we may write $\eta(t) = \bot$ to denote that η is not defined at time point $t \in \mathbb{T} \setminus E(\eta)$. To model streams of events that do not carry a value we use a designated type `Unit`. Formally, we let $\texttt{Unit} = \{\top\}$ be a set with one designated element $\top$.

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 (i.e, $\{t \mid (t, v) \text{ occurs on } s_\eta\} = E(\eta)$).

1.6 Defining Streams

TeSSLa allows for defining streams through function applications. Such functions can be applied to signal and 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 function $f : A \to B$ 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 specifications is given by the following grammar.

$$
\begin{aligned}
\textit{spec} &::= \texttt{define}\ \textit{name}[\texttt{:}\ \textit{stype}]\ \texttt{:=}\ \textit{texpr}\ |\\
&\quad\ \ \texttt{out}\ \textit{texpr}\ |\ \textit{spec}\ \textit{spec}\\
\textit{texpr} &:= \textit{expr}[\texttt{:}\ \textit{type}]\\
\textit{expr} &:= \textit{name}\ |\ \textit{literal}\ |\ \textit{name}\texttt{(}\textit{texpr}(\texttt{,}\ \textit{texpr})^{*}\texttt{)}\\
\textit{type} &:= \textit{btype}\ |\ \textit{stype}\\
\textit{stype} &:= \texttt{Signal<}\textit{btype}\texttt{>}\ |\ \texttt{Events<}\textit{btype}\texttt{>}
\end{aligned}
$$

Names are nonempty strings $\textit{name} \in AB^*$ where $A = \{\texttt{A}, \dots, \texttt{Z}, \texttt{a}, \dots, \texttt{z}\}$ 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:

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

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 filter expression, it 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 occurs 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 signals 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
add $: \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_D$	$\texttt{add}(\sigma_1, \sigma_2)(t) := \sigma_1(t) + \sigma_2(t)$	$D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\}$
sub $: \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_D$	$\texttt{sub}(\sigma_1, \sigma_2)(t) := \sigma_1(t) - \sigma_2(t)$	$D \in \{\mathbb{Z}, \mathbb{R}\}$
mul $: \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_D$	$\texttt{mul}(\sigma_1, \sigma_2)(t) := \sigma_1(t) \cdot \sigma_2(t)$	$D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\}$
div $: \mathcal{S}_D \times \mathcal{S}_{D'} \to \mathcal{S}_D$	$\texttt{div}(\sigma_1, \sigma_2)(t) := \frac{\sigma_1(t)}{\sigma_2(t)}$	$D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\}, D' = D \setminus \{0\}$
ge $: \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_{\mathbb{B}}$	$\texttt{ge}(\sigma_1, \sigma_2)(t) := \sigma_1(t) > \sigma_2(t)$	$D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\}$
geq $: \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_{\mathbb{B}}$	$\texttt{geq}(\sigma_1, \sigma_2)(t) := \sigma_1(t) \geq \sigma_2(t)$	$D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\}$
leq $: \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_{\mathbb{B}}$	$\texttt{leq}(\sigma_1, \sigma_2)(t) := \sigma_1(t) \leq \sigma_2(t)$	$D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\}$
eq $: \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_{\mathbb{B}}$	$\texttt{eq}(\sigma_1, \sigma_2)(t) := \sigma_1(t) = \sigma_2(t)$	any D with equality
max $: \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_D$	$\texttt{max}(\sigma_1, \sigma_2)(t) := \max\{\sigma_1(t), \sigma_2(t)\}$	$D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\}$
min $: \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_D$	$\texttt{min}(\sigma_1, \sigma_2)(t) := \min\{\sigma_1(t), \sigma_2(t)\}$	$D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\}$
abs $: \mathcal{S}_D \to \mathcal{S}_D$	$\texttt{abs}(\sigma)(t) := \lvert\sigma(t)\rvert$	$D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\}$
abs $: \mathcal{E}_D \to \mathcal{E}_D$	$\texttt{abs}(\eta)(t) := \begin{cases} \lvert\eta(t)\rvert & \text{if } t \in E(\eta) \\ \perp & \text{otherwise} \end{cases}$	
and $: \mathcal{S}_{\mathbb{B}} \times \mathcal{S}_{\mathbb{B}} \to \mathcal{S}_{\mathbb{B}}$	$\texttt{and}(\sigma_1, \sigma_2)(t) := \sigma_1(t) \wedge \sigma_2(t)$	
or $: \mathcal{S}_{\mathbb{B}} \times \mathcal{S}_{\mathbb{B}} \to \mathcal{S}_{\mathbb{B}}$	$\texttt{or}(\sigma_1, \sigma_2)(t) := \sigma_1(t) \vee \sigma_2(t)$	
implies $: \mathcal{S}_{\mathbb{B}} \times \mathcal{S}_{\mathbb{B}} \to \mathcal{S}_{\mathbb{B}}$	$\texttt{implies}(\sigma_1, \sigma_2)(t) := \sigma_1(t) \Rightarrow \sigma_2(t)$	
not $: \mathcal{S}_{\mathbb{B}} \to \mathcal{S}_{\mathbb{B}}$	$\texttt{not}(\sigma)(t) := \neg\sigma(t)$	
not $: \mathcal{E}_{\mathbb{B}} \to \mathcal{E}_{\mathbb{B}}$	$\texttt{not}(\eta)(t) := \begin{cases} \neg\eta(t) & \text{if } t \in E(\eta) \\ \perp & \text{otherwise} \end{cases}$	

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:

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

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

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

Discrete timporal shifts of event streams can be expressed by the functions

$$\mathbf{delay} : \mathcal{E}_D \to \mathcal{E}_D$$
$$\texttt{delay} : \mathcal{E}_D \times \mathbb{N}_{>0} \to \mathcal{E}_D$$

defined in terms of the sequence representation s_η of event streams $\eta \in \mathcal{E}_D$. For $s_\eta = (t_0, \eta(t_0))(t_1, \eta(t_1))(t_2, \eta(t_2))\ldots$ we define $\texttt{delay}(s_\eta) := (t_1, \eta(t_0))(t_2, \eta(t_1))\ldots$. Notice that in case $|s_\eta| < 2$ then $\texttt{delay}(s_\eta) = \varepsilon$ is empty. A positive natural argument abbrevates iterated application, i.e., $\texttt{delay}(\eta, n) := \texttt{delay}^n(\eta)$.

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

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

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*:

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

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

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

3.3 Synchronisation

We define a function `synchronise` that matches events from two streams within a given time range. Its formal signature is defined as

add example or ref appendix

$$\texttt{synchronise} : \mathcal{E}_{D_1} \times \mathcal{E}_{D_2} \times \mathbb{T} \to \mathcal{E}_{(D_1 + D_2) + (D_1 \times D_2)}.$$

Recall that $A + B := (A \times \{1\}) \cup (B \times \{2\})$[4].

Since the function takes the temporal relation between the events into account it is most convenient to define its semantics inductively based on the sequence representation. For $d \in \mathbb{T}_{\geq 0}$ and sequences $u \in (\mathbb{T}_{\geq 0} \times D_1)^\infty$ and $v \in (\mathbb{T}_{\geq 0} \times D_2)^\infty$ we define the sequence $sync(u, v)$ inductively by

$$sync(u, v) = \begin{cases} \varepsilon & \text{if } u = v = \varepsilon \\ ((a, 1, 1), t_1 + d) \cdot sync(u', v) & \text{if } u = (a, t_1) \cdot u' \text{ and} \\ & v = \varepsilon \text{ or } v = (b, t_2) \cdot v' \text{ with } t_2 > t_1 + d \\ ((b, 2, 1), t_2 + d) \cdot sync(u, v') & \text{if } v = (b, t_2) \cdot v' \text{ and} \\ & u = \varepsilon \text{ or } u = (a, t_1) \cdot u' \text{ with } t_1 > t_2 + d \\ ((a, b, 2), \max\{t_1, t_2\}) \cdot f(u', v') & \text{if } u = (a, t_1) \cdot u',\ v = (b, t_2) \cdot v' \text{ and } |t_1 - t_2| \leq d \end{cases}$$

Based on $sync(u, v)$ we define $\texttt{synchronise}(\eta_1, \eta_2, d) := \eta_3$ where η_3 is the event stream defined by the sequence $s_{\eta_3} = sync(s_{\eta_1}, s_{\eta_2})$. As defined in Section 1.5, the sequences s_{η_1} and s_{η_2} are the sequences representing η_1 and η_2, respectively.

[4]In programming languages, structures $A + B$ are often represented by a type `Either<A,B>` with subtypes `Left<A>` and `Right<B>` and suitable operations to access the wrapped values of type A and B, respectively (e.g., `get()`).

For convenience we further define some lifted functions to access the structural information.

$$\texttt{getLeft} : \mathcal{E}_{D_1 \times D_2} \to \mathcal{E}_{D_1} \qquad \texttt{getLeft}(\eta)(t) := \begin{cases} v_1 & \text{if } t \in E(\eta) \text{ and } \eta(t) = (v_1, v_2) \\ \bot & \text{otherwise} \end{cases}$$

$$\texttt{getRight} : \mathcal{E}_{D_1 \times D_2} \to \mathcal{E}_{D_2} \qquad \texttt{getRight}(\eta)(t) := \begin{cases} v_2 & \text{if } t \in E(\eta) \text{ and } \eta(t) = (v_1, v_2) \\ \bot & \text{otherwise} \end{cases}$$

$$\texttt{getLeft} : \mathcal{E}_{(D_1 + D_2)} \to \mathcal{E}_{D_1} \qquad \texttt{getLeft}(\eta)(t) := \begin{cases} v_1 & \text{if } t \in E(\eta) \text{ and } \eta(t) = (v_1, 1) \\ \bot & \text{otherwise} \end{cases}$$

$$\texttt{getRight} : \mathcal{E}_{(D_1 + D_2)} \to \mathcal{E}_{D_2} \qquad \texttt{getRight}(\eta)(t) := \begin{cases} v_2 & \text{if } t \in E(\eta) \text{ and } \eta(t) = (v_2, 2) \\ \bot & \text{otherwise} \end{cases}$$

$$\texttt{get} : \mathcal{E}_{D_1 + D_2} \to \mathcal{E}_{D_1 \cup D_2} \qquad \texttt{get}(\eta)(t) := \begin{cases} v & \text{if } t \in E(\eta) \text{ and } \eta(t) = (v, i) \text{ for } i \in \{1, 2\} \\ \bot & \text{otherwise} \end{cases}$$

We define a function that indicates whether an event could not be synchronised within the given time bound:

$$\texttt{timeout} : \mathcal{E}_{(D_1 + D_2) + (D_1 \times D_2)} \to \mathcal{E}_{\{\top\}}$$
$$\texttt{timeout}(\eta)(t) := \begin{cases} \top & \text{if } t \in E(\eta) \text{ and } \eta(t) = (v, i, 1) \in (D_1 + D_2) \times \{1\} \\ \bot & \text{otherwise} \end{cases}$$

A "flat" synchronisation simply neglects the origin of a value and gives precedence to the values of the first argument.

$$\texttt{flatSynchronise} : \mathcal{E}_D \times \mathcal{E}_D \times \mathbb{T} \to \mathcal{E}_D$$
$$\texttt{flatSynchronise}(\eta_1, \eta_{,2}, d)(t) := \mathit{get}(\mathit{get}(\texttt{synchronise}(\eta_1, \eta_2, d)))$$

If necessary, we can keep the timeout information.

$$\texttt{flatSynchronise} : \mathcal{E}_D \times \mathcal{E}_D \times \mathbb{T} \to \mathcal{E}_{D \times \mathbb{B}}$$
$$\texttt{flatSynchronise}(\eta_1, \eta_{,2}, d)(t) := \begin{cases} (v, \texttt{true}) & \texttt{if synchronise}(\eta_1, \eta, 2, d)(t) = (v, i, 1) \texttt{ for } i \in \{1, 2\} \\ (v_1, \texttt{false}) & \texttt{if synchronise}(\eta_1, \eta, 2, d)(t) = (v_1, v_2, 2) \\ \bot & \texttt{otherwise} \end{cases}$$

3.4 Aggregations

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

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

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

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

We further define the simple moving average on streams of arithmetic type. Let $\max_1(M) := \{\max M\}$ and $\max_{n+1}(M) := \{\max M\} \cup \max_n(M \setminus (\max M))$ denote the set of the n largest elements of a linearly ordered set M. Then, we define the simple moving average $\texttt{sma} : \mathcal{E}_D \times \mathbb{N}_{>0} \to \mathcal{E}_D$ as

move to preliminaries

$$
\texttt{sma}(\eta, n)(t) := \begin{cases} \frac{\sum_{t' \in \max_n \{t'' \in E(\eta) \mid t'' \leq t\}} \eta(t')}{|\max_n \{t' \in E(\eta) \mid t' \leq t\}|} & \text{if } t \in E(\eta) \\ \bot & \text{otherwise} \end{cases}
$$

Notice that for a fixed n we have that

```
define std_mean_avg := sma(in1, n)
out std_mean_avg
```

is equivalent to

```
define x₁ = in1
define x₂ = delay(x₁)
define x₃ = delay(x₂)
...
define xₙ = delay(x(n − 1))
define sum = add(x₁,add(...add(xₙ₋₁, xₙ)...))
define std_mean_avg := on xₙ yield ifThenElse(sum>0, sum/n, 0)
out std_mean_avg
```

3.5 Selectors/Filters/Conditionals/Combinators

$$\texttt{\textbf{changeOf}} : \mathcal{S}_D \to \mathcal{E}_{\{\top\}} \qquad \texttt{changeOf}(\sigma)(t) := \begin{cases} \top & \text{if } t \in \Delta(\sigma) \\ \bot & \text{otherwise} \end{cases}$$

$$\texttt{\textbf{ifThen}} : \mathcal{E}_{D_1} \times \mathcal{S}_{D_2} \to \mathcal{E}_{D_2} \qquad \texttt{ifThen}(\eta, \sigma)(t) := \begin{cases} \sigma(t) & \text{if } t \in E(\eta) \\ \bot & \text{otherwise} \end{cases}$$

$$\texttt{sample} : \mathcal{S}_{D_1} \times \mathcal{E}_{D_2} \to \mathcal{E}_{D_1} \qquad \texttt{sample}(\sigma, \eta) := \texttt{ifThen}(\eta, \sigma)$$

$$\texttt{\textbf{filter}} : \mathcal{E}_D \times \mathcal{S}_{\mathbb{B}} \to \mathcal{E}_D \qquad \texttt{filter}(\eta, \sigma)(t) := \begin{cases} \eta(t) & \text{if } t \in E(\eta) \\ & \text{and } \sigma(t) = \texttt{true} \\ \bot & \text{otherwise} \end{cases}$$

$$\texttt{ifThenElse} : \mathcal{S}_{\mathbb{B}} \times \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_D \quad \texttt{ifThenElse}(\sigma_1, \sigma_2, \sigma_3)(t) := \begin{cases} \sigma_2(t) & \text{if } \sigma_1(t) = \texttt{true} \\ \sigma_3(t) & \text{otherwise} \end{cases}$$

$$\texttt{merge} : \mathcal{E}_D \times \mathcal{E}_D \to \mathcal{E}_D \qquad \texttt{occurAny}(\eta_1, \eta_2)(t) := \begin{cases} \eta_1(t) & \text{if } t \in E(\eta_1) \\ \eta_2(t) & \text{if } t \in E(\eta_2) \setminus E(\eta_1) \\ \bot & \text{otherwise} \end{cases}$$

$$\texttt{\textbf{occurAny}} : \mathcal{E}_{D_1} \times \mathcal{E}_{D_2} \to \mathcal{E}_{\{\top\}} \qquad \texttt{occurAny}(\eta_1, \eta_2)(t) := \begin{cases} \top & \text{if } t \in E(\eta_1) \cup E(\eta_2) \\ \bot & \text{otherwise} \end{cases}$$

$$\texttt{\textbf{occurAll}} : \mathcal{E}_{D_1} \times \mathcal{E}_{D_2} \to \mathcal{E}_{\{\top\}} \qquad \texttt{occurAll}(\eta_1, \eta_2)(t) := \begin{cases} \top & \text{if } t \in E(\eta_1) \cap E(\eta_2) \\ \bot & \text{otherwise} \end{cases}$$

Notice that `merge` gives precedence to the first argument and that `ifThen` is a restricted form of an on-comprehension.

3.6 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 finite 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,

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

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 $w_t = a_1 \ldots a_m$ with $a_i = \{p_k \in AP \mid \sigma_k(t_i) = \texttt{true}\}$ for $\{t_1 < t_2 < \ldots < t_{|E(\eta)|}\} = E(\eta)$ and $t_m \leq t < t_{m+1}$.

For practical convenience we could introduce further notation such as where explicit inputs are optionals and instead of fixed propositions $p_1 \ldots p_n$ escaped TeSSLa expressions can be used. For example,
`monitor("always (p1 implies eventually not p2)", myFunction(in1, in2), in3)`
could be written as
`monitor "always ( {myFunction(in1, in2)} implies eventually not {in3})".`

3.7 On-comprehension

Let $\eta_1 \in \mathcal{E}_{D_1}, \ldots, \eta_n \in \mathcal{E}_{D_n}$ be event streams, $\sigma_{cnd} \in \mathcal{S}_{\mathbb{B}}$ a Boolean signal and $\sigma_{val} \in \mathcal{S}_D$ a signal of some type D. We define the semantics of on-comprehensions as follows.

$$[\![\texttt{on } \eta_1, \ldots, \eta_n \texttt{ if } \sigma_{cnd} \texttt{ yield } \sigma_{val}]\!](t) := \begin{cases} \sigma_{val}(t) & \text{if } t \in \bigcap_{i=1}^{n} E(\eta_i) \text{ and } \sigma_{cnd}(t) = \texttt{true} \\ \bot & \text{otherwise} \end{cases}$$

Omitting the `yield` part amounts to implicitly take $\sigma_{val} = \sigma_\top$ where $\sigma_\top : \mathbb{T}_{\geq 0} \to \{\top\}$ is the (unique) signal of type `Unit`. Omitting the conditional (`if`) part amount to implicitly take $\sigma_{cnd} = \sigma_{\texttt{true}}$ where $\sigma_{\texttt{true}} : \mathbb{T}_{\geq 0} \to \{\texttt{true}\}$ is the Boolean signal that is always true.

A Examples

TeSSLa specification formally describe transformations of input to output streams where the set of input streams is defined implicitly by names occurring freely in the specification. In practice, however, a more specific control over the inputs is desired and therefore implementations may provide specific input functions, that only depend on constants, e.g., strings describing the stream source.

In the following examples we assume input functions

$$\begin{aligned} \texttt{function\_calls} &: \texttt{String} \to \mathcal{E}_D \\ \texttt{function\_returns} &: \texttt{String} \to \mathcal{E}_D \\ \texttt{instruction\_executions} &: \texttt{String} \to \mathcal{E}_D \end{aligned}$$

Similarly, the representation of time (intervals) is implementation dependent. In the example specification we use integer numbers and suffixes (us,ms,s) to indicate appropriate factors. Ideally, an implementation will convert, e.g., $1s$, $1ms$ and $1us$, into a representation of 1 second, 1 millisecond and 1 microsecond, respectively.

Double hyphens (`--`) indicate commentary until line ending.

A.1 Delay

Property: Whenever an event *source_event* occurs, within the next 1.2 seconds the next event *target_event* must occur. A *target_event* can serve for multiple *source_event* occurrences.

A.1.1 Delay using SALT

```
define source_event := function_calls("main.c:open_door")
define target_event := function_returns("main.c:open_door")

define monitor_output := monitor("
  always (if p1 then next timed[<= 1200] p2)",
  source_event,
  target_event
)

out monitor_output
```

A.1.2 Delay using SALT without real time operators

```
define source_event := function_calls("main.c:open_door")
define target_event := function_returns("main.c:open_door")

define monitor_output := monitor("
  always (if p1 then p2)",
  source_event,
  inFuture(1200ms, target_event)
)

out monitor_output
```

A.2 Strong Delay and Order

Property: For every *source_event* a matching *target_event* must occur within the next 1.2 seconds. Every *target_event* is associated to at most one *source_event*.

```
define source_event := function_calls("main.c:open_door")
define target_event := function_returns("main.c:open_door")

define event_pairs := synchronize(source_event, target_event, 1200ms)
define error := timeout(event_pairs)

out error
```

A.3 Synchronization

Property: Whenever an event occurs, all other events have to occur also within a range of 1.2 seconds.

A.3.1 Synchronization using SALT

```
define event_a := instruction_executions("test.c:23")
define event_b := instruction_executions("test.c:42")
define event_c := instruction_executions("test.c:1729")

define monitor_output := monitor("
  always (
    (if p1 then (next timed[<= 1200] p2 or previous timed[< 1200] p2)) and
    (if p1 then (next timed[<= 1200] p3 or previous timed[< 1200] p3)) and
    (if p2 then (next timed[<= 1200] p1 or previous timed[< 1200] p1)) and
    (if p2 then (next timed[<= 1200] p3 or previous timed[< 1200] p3)) and
    (if p3 then (next timed[<= 1200] p1 or previous timed[< 1200] p1)) and
    (if p3 then (next timed[<= 1200] p2 or previous timed[< 1200] p2)))",
  event_a,
  event_b,
  event_c
)

out monitor_output
```

A.3.2 Synchronization using SALT without real time operators

```
define event_a := instruction_executions("test.c:23")
define event_b := instruction_executions("test.c:42")
define event_c := instruction_executions("test.c:1729")

define monitor_output := monitor("
  always (
    (if p1 then p5) and
    (if p1 then p6) and
    (if p2 then p4) and
    (if p2 then p6) and
    (if p3 then p4) and
    (if p3 then p5))",
  event_a,
  event_b,
  event_c,
  within(-1200ms, 1200ms,event_a),
  within(-1200ms, 1200ms,event_b),
  within(-1200ms, 1200ms,event_c)
)
```

```
out monitor_output
```

A.4 Periodic

Property: There exists an event that occurs periodically (period_event). Whenever period_event occurs, an event (event) has to occurs after at most one second. Between the occurrence of event have to be at least 0.7 seconds.

A.4.1 Periodic using SALT

```
define periodic_event := instruction_executions("test.c:34653")
define event := instruction_executions("test.c:242")

define monitor_output := monitor("
  always (
    (if p1 then next timed[<= 1000] p2) and
    (if p2 then not(next timed[<= 700] p1)))",
  period_event,
  event
)

out monitor_output
```

A.4.2 Periodic using SALT without real time operators

```
define periodic_event := instruction_executions("test.c:34653")
define event := instruction_executions("test.c:242")

define monitor_output := monitor("
  always (
    (if p1 then p3) and
    (if p2 then not(p4)))",
    period_event,
    event,
    inFuture(1000ms,event),
    inFuture(700ms,event)
)

out monitor_output
```

A.5 Use Cases D1-D4

A.5.1 Use Case D1 - Simple Saftey Constraint

Assume an input stream, generated by the application. Only values from the range 0-10 are supposed to occur.

```
define error := on changeOf(APPL_MSG_Value) if geq(APPL_MSG_Value, 11)
out error
```

A.5.2 Use Case D2 - Timing Constraints

Assume an input (events) A and B.

Property D2.1: Event B must not occur within 500ms after any occurrence of event A.

```
define error := on A if inFuture(500ms, B)
out error
```

Alternative:

```
define error := on B if inPast(500ms, A)
out error
```

A Salt formula always($A \to \neg$(next timed[<= 500]B))

```
define monitor_output := monitor("always if p1 then not next timed [<=500ms] p2", A, B)
out monitor_output
```

D2.2: In the environment of 500ms around every event A some B must occur

```
define error := on A if not(within( -500ms, 500ms, B))
out error
```

A.5.3 Use Case D3 - Numerical analysis

Assume as inputs: signals A and B, event stream C

D3.1: Value exceeds limit of 1023

```
define out_of_range := geq(A,1024)
out out_of_range
```

D3.2: Difference between maximum and minimum value exceeds limit of 1023

```
define diff := sub(maximum(A),minimum(A))
out geq(diff,1024)
```

D3.3: Values of events C deviate at most by 1023 from their mean

```
define mean := div(sum(C),max(eventCount(C),1))
    -- assumes implicit initialisation event with value 0
define diffMax := sub(maximum(C, 0), mean)
define diffMean := sub(mean, maximum(C, 0))
define out_of_range := or(geq(diffMax,1024),geq(diffMean,1024))
out on changeOf(out_of_range) if out_of_range
```

D3.4: Error if value of C diverges by more than 1023 from moving average of previous 10 values

```
define assertion := leq(abs(sub(C,sma(C, 10))),1024)
define error := on changeOf(assertion) if not(assertion)
out error
```