TESSLA—A Temporal Stream-based Specification Language

August 8, 2016

1 Introduction

1.1 The Setting

TeSSLa is developed in the scope of the CONIRAS project as a language for specifying properties of multicore systems. The goal is to observe the execution of C programs on multicore systems with the processor's debug interfaces and check if these executions fulfill a certain correctness property specified in TeSSLa. The correctness properties are synthesized as a monitor on an FPGA which also gets the data from the debug interfaces. Then the FPGA can check by using the monitor if the properties are violated of satisfied.

In the CONIRAS project the goal was to use runtime verification techniques early in the development process for debugging multicore systems. Because synthesizing a monitor on an FPGA can take a huge amount of time, it was an important requirement that the elements of a monitor on the FPGA do not have to be synthesized every time a new monitor is created. Instead, a general bunch of elements should be synthesized on the FPGA and only if the elements needed for a TeSSLa specification and their connections do not match to the ones available on the FPGA a new synthesis process has to be started.

The whole workflow for creating a monitor from a TeSSLa specification and to monitor a C program is given in Figure 1. At first the observable points, called tracepoints, have to be extracted from the compiled C code with debug symbols. Tracepoints can be various lines in the assembler code, for example jumps, a write to a variable or similar things. These tracepoints are needed in TeSSLa specifications to relate to certain lines in the C code and thus they are also needed for the TeSSLa-compiler such that it knows which lines of code in the C program can be observed when they are reached during execution. After a TeSSLa specification is compiled the compiler outputs a directed acyclic graph (DAG) which contains all the functions and constants used in the specification. Afterwards, if this DAG matches to the elements and connections available on the FPGA, the FPGA is configured to work as the specified monitor. The data which the monitor works on comes from the debug interfaces of the CPU. Besides the information about which operation was executed, the data contains

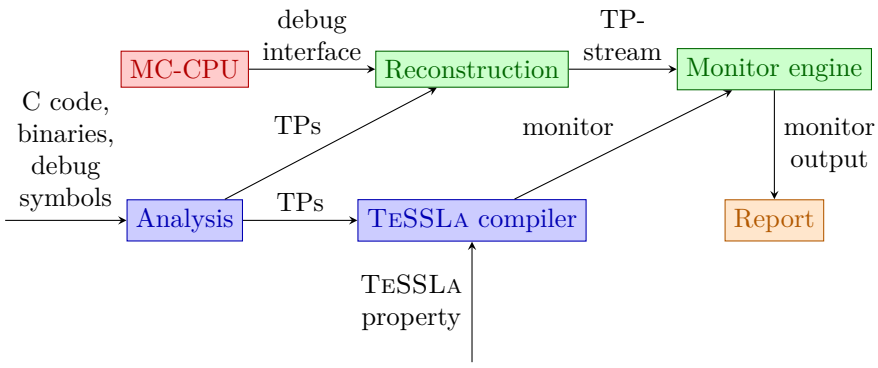

Figure 1: The workflow for reconstructing the program sequence and synthesizing the monitor for a TeSSLa specification. The blue nodes represent the part needed for the compilation of a TeSSLa specification, the red node represents the multicore CPU, the green nodes represent the parts that are synthesized on the FPGA and the orange node represents the output of the verification process.

timestamps and an ownership ID which relates to the core that executed the operation. But before giving the data into the monitor engine on the FPGA the tracepoints have to be reconstructed. Therefore an Instruction Reconstruction (IR) unit is synthesized on the FPGA in front of the monitor engine, hence all data that comes out of the CPU debug interface first goes through this IR unit. The IR unit matches the incoming data to the tracepoints and sends an event to the monitor engine every time a tracepoint was executed by the CPU. All the data that does not related to a tracepoint or does not contain general important information will be filtered out by the IR unit to reduce the amount of data that reaches the monitor.

1.2 Targeted Systems

TeSSLa was developed as a specification language for multicore systems, more precisely for systems on chip (SoC). This is a very rough classification because there are various types of such systems. Hence in this section the systems under scrutiny are described more concrete.

Generally, for the system under scrutiny it is assumed that...

- OS / no OS
- Memory (shared or not)
- Paging?

	Asynchronous	Real Time	Arithmetic	Stream Ops
LOLA	×	×	?	?
STL	?	×	?	?
TIMMO	?	✓	?	✓
QRE	?	?	?	?

Figure 2: Overview over the characteristics of different temporal logics.

1.3 Taxonomy of Temporal Languages

Different temporal languages have different characteristics which influence the expressiveness of the language and the way correctness properties can be specified. The characteristics of temporal logics lead from the usages of different types of input streams and the possibility to specify real time values to the availability of arithmetic operations on the input values. In this section different characteristics of temporal logics are considered and various logics that use streams as inputs are investigated for their characteristics.

The first one is how the input looks like. For streams, there are two different possibilities: They can be synchronized, hence there is a certain step frequency after which on every input stream synchronously an event occurs. The counterpart to this are asynchronous streams which means that on every stream an event can happen at any point in time independent from the other streams.

Another characteristic is if real time properties can be expressed with the logic. A real time property describes a property of the system that somehow involves a connection to real time time units like seconds or microseconds. Generally some real time bound has to be fulfilled in a real time property.

The third important aspect of temporal logics are arithmetic operators. These are needed to derive new streams from existing ones by applying an arithmetic function to existing streams. Examples for such operators are the calculation of the average of the values of a stream or simply comparing the values on a stream to a constant.

The last characteristic we consider is the availability of stream manipulation operators. These operators allow for manipulating, filtering or merging streams for certain events. They are needed, for example, if a stream contains data from different cores but only the data of certain cores is of interest. Then the stream needs to be filtered for the important events.

Different temporal logics cover different of these characteristics. An overview is given in Table 2. As it can be seen, none of the logics covers all of the considered characteristics but as shown in the next section, all of them are needed. That is why none of the mentioned temporal logics could be used in the CONIRAS project and TeSSLa was developed to cover all of these characteristics.

1.4 The Design of TeSSLa

1.4.1 General Requirements

Various requirements lead to the form TeSSLa has now. Some requirements directly result from setting: because the clock speed of the FPGA is much lower than the one of the CPU cores, an asynchronous processing of the incoming data is necessary which leads to the model of asynchronous streams used in TeSSLa. Furthermore, the fact that the systems under scrutiny are multicore systems and hence data comes from multiple core in arbitrary order and amount fits the model of asynchronous streams well.

1.4.2 Timing Requirements

Besides the requirements that result directly from the setting, there are different requirements which come from the properties that should be specifiable. By viewing the TIMMO specification language used in the AUTOSAR project it becomes clear that real time constraints are really important in the scope of multicore systems. Additionally the difference of the timestamps of two events can be arbitrarily small which leads to the need of a continuous time model with the constraint that in a finite time frame only a finite number of events, and thus timestamps, can occur. Otherwise no monitoring is possible because an infinite number of events has to be processed in a finite amount of time.

Among others these include properties like the following:

1. *Every time an event e_1 occurs, at most n seconds after (before respectively) e_1 another event e_2 has to occur.*
2. *For every event in the event stream E_1, at most n seconds after (before respectively) an Event has to occur on the stream E_2. Every event that occurs on E_2 can only be matched to at most one from E_1.*

This results in functions being available in TeSSLa that work on real time units.

1.4.3 Arithmetic Requirements

Other important properties of program executions on CPUs require to count events that are happening and also to apply different arithmetic operations on the counted values. This is needed to recognize buffer overflows, check fairness properties on, for example, variables access or resource usage, prove constraints on the number of function calls in the C code and other properties. Another use case for arithmetics is to make calculations on variable values, for example medians.

Among others these include properties like the following:

1. *The difference of the number of write events and the number of delete events for a buffer of size 5 should never exceed 5.*

2. *The median of the ten last values a temperature sensor delivers should never exceed 20.*

These are typical properties that are interesting for the industry and hence to be able to specify those properties, TeSSLa needs arithmetic functions.

1.4.4 Stream Operation Requirements

Not only but also because of the necessity of counting events, it is necessary to apply operations directly on the input streams. These can transform the data of the events of an input stream or can even add or take out whole events. Typical examples for these operations are if-then constructs or filter and merge functions which are needed in TeSSLa.

Among others these include properties like the following:

1. *There exists a system with three cores. Both core one and core two should be able to access a certain resource at least once every ten seconds.*
2. *A certain threshold value should not be exceeded by a variable more than three times in the last ten values of that variable.*

1.4.5 Temporal Logic Requirements

Besides the properties that are mentioned above an important class of properties is missing: the properties that are specifiable with typical temporal logics such as the LTL or SALT. These have proven to be interesting in a lot of systems. Furthermore the different semantics available for such logics are needed to detect final output values and to detect those as soon as possible such that errors are acknowledged very early. To be able to use those logics it has to be possible to transform formulas of these logics into state machines which leads to the need to detect where such logics are used. Because of this it is the easiest way to include them into TeSSLa in a separated function. Another positive side effect is that state machines can be synthesized and executed extremely efficient on the FPGA monitor engine.

Properties for which temporal logics can be used are the following:

1. *When the only process that writes values to the buffer is deactivated, no write operations should happen on the buffer until the process is activated again.*
2. *Every time a occurs a b has to be occurred 2 to 5 seconds before.*

1.5 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

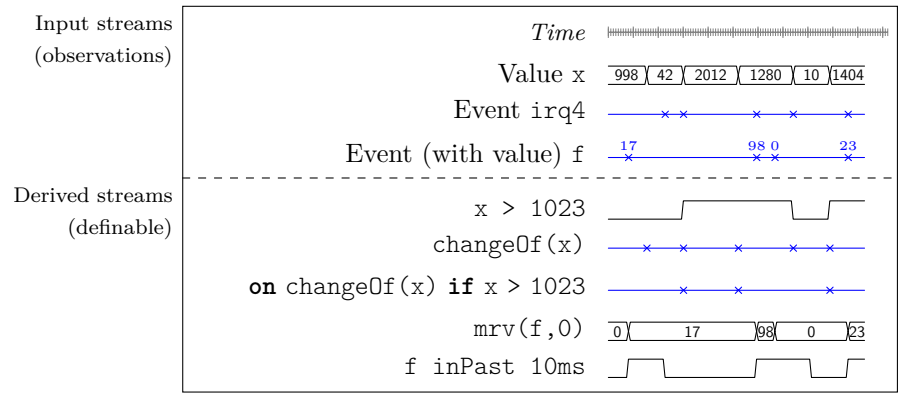

Figure 3: Example streams.

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.

1.6 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.

[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 $\mathbb{Q}$.

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 $\mathcal{E}_D$.*

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

1.7 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`:

[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.

```
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 **??**.

1.8 Monitoring Engine

The TeSSLa compiler translates a specification into a DAG of functions. This DAG is then synthesized on an FPGA in order to execute such a specification. On the FPGA both signals and event streams are represented as tuples of data and timestamps. In case of signals each value change corresponds to such a tuple. On the FPGA data and timestamps are transferred through seperated lines with sufficient bus with. Data and the corresponding timestamp are always kept in sync. An n-ary function in the DAG is translated into a function block with n data inputs, one timestamp input, one data output and one timestamp output. Note that every function block has only one timestamp input, which may require additional synchronization in front of it: If you want to combine n not yet synchronized streams in one function block, a synchronization block is added which takes n value inputs and the corresponding n timestamp inputs and has n value outputs and only one timestamp output. This block buffers the

incoming values and timestamps until all values for a common timestamp are known, and then emits one timestamp together with n values.

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.

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

Names are nonempty strings $\mathit{name} \in AB^*$ where $A = \{\texttt{A}, \ldots, \texttt{Z}, \texttt{a}, \ldots, \texttt{z}\}$ 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 $-1, 0, 1, 2, \ldots$, 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:

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

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

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:

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

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

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

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

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

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 $\mathtt{delay}(s_\eta) := (t_1, \eta(t_0))(t_2, \eta(t_1))\ldots$. Notice that in case $|s_\eta| < 2$ then $\mathtt{delay}(s_\eta) = \varepsilon$ is empty. A positive natural argument abbrevates iterated application, i.e., $\mathtt{delay}(\eta, n) := \mathtt{delay}^n(\eta)$.

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 1.6, 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., $\mathbb{N}, \mathbb{Z}, \mathbb{R}, \mathbb{Q}$:

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

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

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

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

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

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

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

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

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

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

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

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

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,

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

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 $p_1 \dots p_n$ 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 $\eta_1 \in \mathcal{E}_{D_1}, \dots, \eta_n \in \mathcal{E}_{D_n}$ 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.

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

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.

3.8 Related Languages

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($A \rightarrow \neg$(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
```