TeSSLa: Runtime Verification of Non-synchronized Real-Time Streams*

Martin Leucker
ISP, Univ. of Lübeck, Germany
leucker@isp.uni-luebeck.de

César Sánchez
IMDEA Software Inst., Spain
cesar.sanchez@imdea.org

Torben Scheffel
ISP, Univ. of Lübeck, Germany
scheffel@isp.uni-luebeck.de

Malte Schmitz
ISP, Univ. of Lübeck, Germany
schmitz@isp.uni-luebeck.de

Alexander Schramm
IMDEA Software Inst., Spain
alexander.schramm@imdea.org

ABSTRACT

We present TeSSLa, a specification language based on stream runtime verification, designed for monitoring a specific class of real-time signals. Our monitors can observe concurrent systems with a shared clock, but where each component reports observations as signals that arrive to the monitor at different speeds and with different and varying latencies. The signals and streams that TeSSLa supports (including inputs and final verdicts) are not restricted to be Booleans but can be data from richer domains, including integers and reals with arithmetic operations and aggregations. Consequently, TeSSLa can be used both for checking logical properties, and for computing statistics and general numeric temporal metrics (and properties on these richer metrics). We present an online evaluation algorithm for TeSSLa specifications and show a formal proof of the correctness of concurrent implementations of the evaluation algorithm. Finally, we report an empirical evaluation of a highly concurrent Erlang implementation of the monitoring algorithm.

CCS CONCEPTS

• **Theory of computation → Logic and verification**; *Modal and temporal logics*; *Online algorithms*; Semantics and reasoning;

KEYWORDS

Runtime Verification, Online Monitoring, Stream Processing

ACM Reference Format:
Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, and Alexander Schramm. 2018. TeSSLa: Runtime Verification of Non-synchronized Real-Time Streams. In *Proceedings of SAC 2018: Symposium on Applied Computing (SAC 2018).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3167132.3167338

*This work is supported in part by EU COST Action IC1402 "ArVi", the BMBF projects ARAMIS II with funding ID 01 IS 16025 and CONIRAS with funding ID 01 IS 13029, the EU H2020 project COEMS under num. 732016, the EU H2020 project Elastest under num. 731535 and by Spanish MINECO Project "RISCO (TIN2015-71819-P)".

SAC 2018, April 9–13, 2018, Pau, France

ACM ISBN 978-1-4503-5191-1/18/04.
https://doi.org/10.1145/3167132.3167338

1 INTRODUCTION

Runtime verification (RV) is an applied formal technique for verifying, analyzing and supporting software reliability. In contrast to static verification, in RV only one trace of the system under scrutiny is considered. Thus, RV sacrifices completeness guarantees to obtain an immediately applicable and formal extension of testing and debugging. A central problem in runtime verification is how to generate monitors from formal specifications (see [17, 23] for RV surveys).

In this paper we study how to perform runtime verification on concurrent systems that have a shared global clock but whose concurrent components emit events to the monitor at different speeds and with different delays. This assumption is common, for example, when observing embedded systems or when observing execution traces of software running on multi-core processors. At the low-level software analysis, the signals that these systems emit are real-time signals that remain constant between two observations, also known as *piece-wise constant* signals.

We are interested in *online* monitoring performed while the system is running (as opposed to offline monitoring through post-mortem analysis of dumped traces). Our target application also requires *non-intrusive* monitoring, meaning that the monitoring activity cannot perturb the execution of the system under observation. We use some hardware capabilities to obtain run-time information while the concurrent system executes. This information is dispatched to an external monitoring infrastructure that executes *outline* (as opposed to inlining the monitors within the system itself). See [22] for a definition and classification of these RV concepts.

In a nutshell, the goals of this paper are (1) to study how to describe sophisticated properties of continuous piece-wise constant signals, and (2) to efficiently monitor these properties against systems where each component is an event source that can emit events at different speeds and with different latencies. We say that these systems emit "non-synchronized in-order streams". In our setting it is relevant to distinguish between the *system time* and *monitor time*. System time refers to the moments at which events are produced within the observed system. These instants are captured by the synchronized global clock that is used to time-stamp events. Monitor time refers to the instants at which events arrive at the monitor and when these events are processed in order to produce verdicts.

Event streams from hardware processors come at very high speeds, which imposes the additional requirement of crafting highly efficient monitoring implementations. We explore here how to

exploit parallelism and distributed implementations using multi-core platforms, while still formally guaranteeing the correctness of the monitors.

We propose here the specification language TeSSLa to achieve these goals. TeSSLa stands for Temporal Stream-based Specification Language. TeSSLa is based on Stream Runtime Verification (SRV) and has already been used for creating monitors in FPGA hardware in [10] without providing a concrete definition or further theoretical background. Early specification languages for RV were based on their counterparts in static verification, typically logics like LTL [25] or past LTL adapted for finite paths [5, 12, 18]. Similar formalisms proposed are based on regular expressions [31], timed regular expressions [2], rule based languages [3], or rewriting [28]. Stream runtime verification, pioneered by the tool LOLA [9], is an alternative to define monitors using streams. In SRV one describes the dependencies between input streams of values (observable events from the system under analysis) and defined streams (alarms, errors and output diagnosis information). These dependencies can relate the current value of a depending stream with the values of the same or other streams at the present moment, in past instants (like in past temporal formulas), or in future instants. In SRV there is a clean separation between the evaluation algorithms—that exploit the explicit dependencies between streams—and the data manipulation—expressed by each individual operation. SRV allows to generalize well-known evaluation algorithms from runtime verification to perform collections of numeric statistics from input traces.

SRV resembles synchronous languages [8]—like Esterel [6], Lustre [16] or Signal [14]—but these systems are causal because their intention is to describe systems and not observations, while SRV removes the causality assumption allowing to refer to future values. Another related area is Functional Reactive Programming (FRP) (see [13]), where reactive behaviors are defined using functional programs as building blocks to express reactions. As with synchronous languages, FRP is a programming paradigm and not a monitoring specification language, so future dependencies are not allowed in FRP. On the other hand SRV, was initially conceived for monitoring synchronous systems. See [7, 15, 26] for further developments on SRV. The semantics of temporal logics can also be defined using declarative dependencies between streams of values. For example, temporal testers [27] defined these dependencies for LTL. Likewise, the semantics of Signal Temporal Logic (STL) [11, 24] is defined in terms of the relation between a defined signal and the signals for its sub-expressions, based on Metric Interval Temporal Logic [1].

Here we extend SRV to real-time piece-wise constant signals, and study how to deal with the non-synchronized arrival of events. All previous approaches to SRV assume synchronous sampling and synchronous arrivals of events in all input streams. It is theoretically feasible, in some cases, to reduce the setting in this paper to the synchronous SRV, for example by assuming that all samples are made at instants multiple of a minimum delay, and executing the specification synchronously after every delay. However, the fast arrival of events would render such an approach impractical due to the large number of processing steps that would be required.

STL has also been used to create monitors on FPGAs [20] and for monitoring in different application areas (see for example [21, 30]). However, the assumptions of STL on the signals is different, because the goal of STL is to analyze arbitrary continuous signals and not necessarily changes from digital circuits with accurate clocks. Sampling ratios and sampling instants are important issues in STL, while our signals are accurately represented by the stream of events at the changing points of the signal. In timed regular expressions (TRE) [2] the signals are also assumed to be piece-wise constant. However, our framework can handle much richer data domains than TREs and STL[1]. TREs have been combined with STL [29] to get the advantages of both domains but again the signals analyzed are not necessarily piece-wise constant. Consequently, the results are approximate and sampling becomes, again, an important issue.

Contributions. In summary, the contributions of this paper are:

(1) TeSSLa, an SRV-based specification language for real-time piece-wise constant signals. The syntax and semantics of TeSSLa are presented in Section 2, including the numerous core and library functions.
(2) A method for the systematic generation of parallel and asynchronous online monitors for software monitoring TeSSLa specifications. These monitors handle the non-synchronized arrival of events from different input stream sources. In Section 3 we introduce a computational model for asynchronous concurrent monitors which allows to implement an online evaluation algorithm for TeSSLa specifications. Section 4 describes a prototype implementation developed in Erlang and an early empirical evaluation.

Finally, Section 5 concludes. Missing proofs appear in the appendix.

2 SYNTAX AND SEMANTICS OF TESSLA

We introduce in this section the real-time specification language TeSSLa[2]. We first present some preliminaries, and then introduce the syntax and semantics of TeSSLa.

2.1 Preliminaries

We use two types of stream models as underlying formalism: piece-wise constant signals and event streams. We use $\mathbb{T}$ for the time domain (which can be $\mathbb{N}$, $\mathbb{Q}$, $\mathbb{R}$, etc), and D for the collection of data domains (Booleans, integers, reals, etc). Values from these data domains model observations and the output produced by the monitors.

Definition 2.1 (Event stream). An *event stream* is a partial function $\eta : \mathbb{T} \rightharpoonup D$ such that $E(\eta) := \{t \in \mathbb{T} \mid \eta(t) \text{ is defined}\}$ does not contain bounded infinite subsets. The set of all event streams is denoted by $\mathcal{E}_D$.

The set $E(\eta)$ is called the set of *event points* of η. When η is not defined at a time point t, that is $t \in \mathbb{T} \setminus E(\eta)$, we write $\eta(t) = \bot$. We use $\top$ as the "unit" value (the singleton domain). A finite event stream η can be naturally represented as a timed word, that is, a sequence $s_\eta = (t_0, \eta(t_0))(t_1, \eta(t_1)) \cdots \in (E(\eta) \times D)^*$ ordered by time ($t_i < t_{i+1}$) that contains a D value at all event points.

[1] In the synchronous non-real-time case, [7] contains a thorough theoretical comparison of SRV versus temporal logics, regular expressions, etc. A similar comparison for real-time piece-wise signals is out of the scope of this paper.
[2] TeSSLa is available at www.isp.uni-luebeck.de/tessla

The second type of stream model that we consider is piece-wise constant signals, which have a value at every point in time. These signals change value only at a discrete set of positions, and remain constant between two change points.

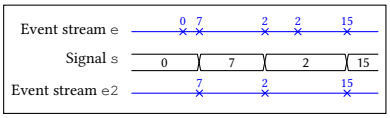
Definition 2.2 (Signal). A *signal* is a total function $\sigma : \mathbb{T} \to D$ such that the set of *change points*

$$\Delta(\sigma) := \{t \in \mathbb{T} \mid \nexists t' < t : \forall t''.t' < t'' < t : \sigma(t) = \sigma(t'')\}$$

does not contain bounded infinite subsets. The set of all signals is denoted by $\mathcal{S}_D$.

Every piece-wise constant signal can be exactly represented by an event stream that contains the change points of the signal as events, and whose value is the value of the signal after the change point. Hence, one can convert signals into event streams and vice-versa. Note that while in STL sampling provides an approximation of fully continuous signals, in TeSSLa event streams represent piece-wise constant signals with perfect accuracy.

Example 2.3. Consider the following streams `e`, `s` and `e2`, where `e` and `e2` are interpreted as event streams and `s` as a signal.



The signal `s` has been created from `e` by using the value of the last event on `e` as value, with a default value 0. In turn, stream `e2` is defined as the changes in value of `s`. When converting an event stream into a signal, only events that represent actual changes are generated. ■

We define next the syntax and semantics of TeSSLa.

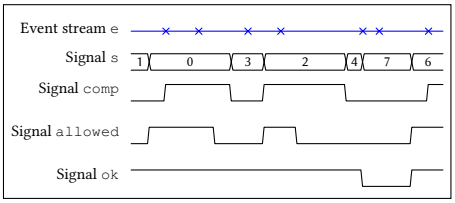
2.2 Syntax of TeSSLa

We begin with an example to illustrate a simple TeSSLa specification. Specifications are written by defining streams in terms of other streams (and ultimately in terms of input streams). Streams marked as **`out`** are the verdict of the monitor and will be reported to the user.

Example 2.4. Consider the following TeSSLa specification:

```
in e: Events<Unit>
in s: Signal<Int>
define comp := eventCount(e) > s
define allowed := within(-1, 1, filter(e, comp))
define ok := implies(s > 5, allowed)
out ok
```

The first two lines define two input streams, `e` (an event stream without values) and `s` (a signal of integers). The Boolean signal `comp` is *true* if the number of events of `e` (denoted by `eventCount(e)`) is greater than the current value of `s`, and *false* otherwise. The Boolean signal `allowed` is *true* when there is an event that has not been filtered out from e in the interval $[-1, +1]$ around the current instant. The function `filter` eliminates an event if the Boolean signal as the second parameter is *false*. Finally, the Boolean signal `ok` is *false* whenever s is greater than 5 and `allowed` is *false*. Consider the input shown in the box below.



When `allowed` is *true*, so will be `ok`. The signal `ok` will also be *true* as long as `s` is lower than 6. When `s` becomes 7, not enough events have happened on `e` and then `comp` is *false*. Consequently, no event is left through the filter and `allowed` is *false* too. But because `s` is greater than 5, `ok` becomes *false*. When `s` is set back to 6 and more events on `e` have happened, `allowed` becomes *true* again. ■

The basic syntax of a TeSSLa specification *spec* is

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

A *name* is a nonempty string. Basic types *btype* cover typical types found in programming and verification like **`Int`**, **`Float`**, **`String`** or **`Bool`**. One of the main contributions of SRV is to generalize existing monitoring algorithms for logics (that produce Boolean verdicts) to algorithms that compute values from richer domains. The production **`in`** introduces input stream variables, and **`define`** introduces defined stream variables (also called output variables). Given a specification φ we use I for the set of input variables and O for the set of output variables, and write $\varphi(I, O)$. For example, in Example 2 above, $I = \{\texttt{e}, \texttt{s}\}$ and $O = \{\texttt{comp}, \texttt{allowed}, \texttt{ok}\}$. The marker **`out`** is used to denote those output variables that are the result of the specification and will be reported to the user. Each defined variable x is associated with a *defining equation* E_x given by the expression on the right hand side of the **`:=`** symbol. 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 "foo", "bar" (enclosed in double quotes).

We expand the syntax of basic TeSSLa by adding builtin functions, user defined macros and timing functions.

$$\begin{aligned} \mathit{name} &::= \mathit{defName} \mid \mathit{timingFun} \mid \mathit{builtinFun} \mid \mathit{macro} \\ \mathit{timingFun} &::= \mathit{delay} \mid \mathit{shift} \mid \mathit{within} \end{aligned}$$

A *defName* is simply a name of a previously defined stream or constant. Timing functions allow to describe timing dependencies between streams. The function *delay* delays the values of a signal (or events of an event stream) by a certain amount of time. The function *shift* shifts the values of an event stream one unit into the future, that is, the first event becomes the second event, etc. The function *within* defines a signal which is *true* as long as some event of the given stream exists within the specified interval.

Macros are user defined functions identified by the construct **`fun`**. Macros can be expanded at compile time using their definition on a purely syntactical level because macros are not recursive.

Example 2.5. An example of a macro has already been used in the Example 2.4 because *implies* is not a builtin function, but the following macro:

```
fun implies(x, y) := or(not(x),y)
```

The full expressivity of TeSSLa is obtained using a set of *builtin functions*. Before describing the builtin functions, we define the temporal core of TeSSLa (which allows to define real time relations between streams) and the general semantics of TeSSLa.

2.3 Semantics

Semantics of Timing Functions. There are three timing functions *delay*, *shift* and *within*. The function *delay* is overloaded for signals

$$delay : \mathcal{S}_D \times \mathbb{T} \times D \to \mathcal{S}_D$$

$$delay(\sigma, d, v)(t) = \begin{cases} \sigma(t-d) & \text{if } t-d \geq 0 \\ v & \text{otherwise} \end{cases}$$

and for event streams

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

$$delay(\eta, d)(t) = \begin{cases} \eta(t-d) & \text{if } t-d \geq 0 \\ \bot & \text{otherwise} \end{cases}$$

The function *delay* delays a signal or an event stream by a given amount of time units. Since signals must always carry a value, a value v is provided as default. For event streams, the occurrence of each event is delayed by the indicated amount of time.

The *shift* function receives an event stream and produces the event stream that results from moving the value of each event to the next event. We use the following notation. Let

$$s_\eta = (t_0, \eta(t_0))(t_1, \eta(t_1))(t_2, \eta(t_2)) \ldots,$$

we use $s_\eta^{\rightarrow}$ for the stream $(t_1, \eta(t_0))(t_2, \eta(t_1)) \ldots$. The signature and interpretation of *shift* is:

$$shift : \mathcal{E}_D \to \mathcal{E}_D$$

$$shift(s_\eta) = \begin{cases} \varepsilon & \text{if } s_\eta = (t_0, \eta(t_0)) \text{ or } s_\eta = \varepsilon \\ s_\eta^{\rightarrow} & \text{otherwise} \end{cases}$$

The last timing function is *within*, which already appeared in Example 2.4. It produces a Boolean valued signal that captures whether there is an event within a timing window:

$$within : \mathbb{T} \times \mathbb{T} \times \mathcal{E}_D \to \mathcal{S}_{\mathbb{B}}$$

$$within(a, b, \eta)(t) = \begin{cases} true & \text{if } E(\eta) \cap [t+a, t+b] \neq \emptyset \\ false & \text{otherwise} \end{cases}$$

Semantics of TeSSLa. We define the semantics of TeSSLa in terms of evaluation models, as commonly done in SRV [9]. The intended meaning of TeSSLa specifications is to define output signals and event streams from input signals and event streams.

Consider a TeSSLa specification over input variables I and defined variables O. A *valuation* of a signal variable x of type D is an element of $\mathcal{S}_D$. Similarly, a valuation of a stream variable y of type D is an element of $\mathcal{E}_D$. We extend valuations to sets of variables in the usual way. If σ_I and σ_O are valuations of sets of variables I and O with $I \cap O = \emptyset$ then we use $\sigma_I \cup \sigma_O$ for the valuation of $I \cup O$ that coincides with σ_I on I and σ_O on O.

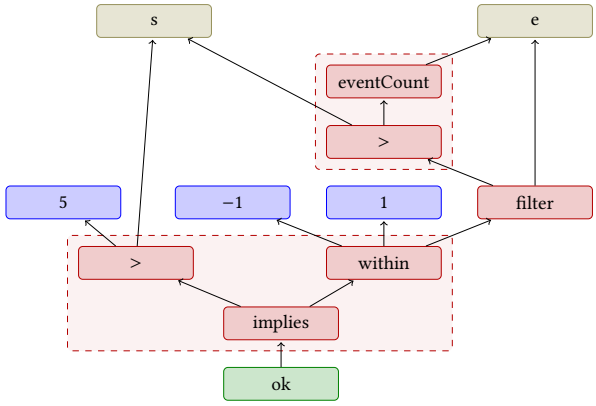


Figure 1: The dependency graph for the spec in Example 2. Inputs are shown in brown, constants in blue, outputs in green, computation nodes in red and some possible merges of computation nodes in dashed red.

Let $[\![l]\!]$ be the value of a literal l, which is an element of its corresponding domain. Also, given a function name f we use $[\![f]\!]$ for the mathematical function that gives an interpretation of f (that is, a map from elements of the domain to an element of the co-domain). Given a valuation σ for each of the variables $I \cup O$ of a specification $\varphi(I, O)$, we can extend the valuation to expressions E over variables I and O, written $[\![E, \sigma]\!]$, recursively as follows:

- *variable name* ($E = name$):

$$[\![name, \sigma]\!] = \sigma(name);$$

- *literal* ($E = l$):

$$[\![l, \sigma]\!] = [\![l]\!];$$

- *function application* ($E = f(e_1, \ldots, e_n)$):

$$[\![E, \sigma]\!] = [\![f]\!]([\![e_1, \sigma]\!], \ldots, [\![e_n, \sigma]\!])$$

An *evaluation model* of a specification $\varphi(I, O)$ is a valuation σ for variables I and O for every $x \in O$ with defining equation E_x: $[\![x, \sigma]\!] = [\![E_x, \sigma]\!]$. Informally, a valuation σ is an evaluation model whenever, for every defined variable x, the value that results when evaluating x and when evaluating its defining expression E_x coincide. We say that a specification $\varphi(I, O)$ is *well-defined* whenever for every valuation σ_I of input variables I there is a unique valuation σ_O of output variables O such that $\sigma_I \cup \sigma_O$ is an evaluation model of φ.

Non-recursive specifications. In order to guarantee that every specification is well-defined, we restrict TeSSLa specifications such that no variable x can depend on itself. More formally, given a specification $\varphi(I, O)$ we say that a variable x directly depends on a variable y if y appears in the defining equation E_x, and we write $x \to y$. We say that x depends on y if $x \to^+ y$ (where $\to^+$ is the transitive closure of $\to$). The dependency relation $x \to^+ y$ gives a necessary condition for y to affect in any way the value of x. The dependency graph has variables as nodes and the dependency relation as edges. Note that input variables and constants are leafs

in the dependency graph. The dependency graph of legal TeSSLa specifications must be non-recursive (i.e. for every x, $x \not\twoheadrightarrow^+ x$), which is easily checkable at compile time. If this is the case, the dependency graph is a DAG and a reverse topological order gives an evaluation order to compute the unique evaluation model. If all variables y preceding x have been assigned a valuation (the only one for which $[\![y]\!] = [\![E_y]\!]$) then $[\![E_x]\!]$ can be evaluated, which is the only possible choice for x.
Hence, this restriction guarantees that all TeSSLa specifications are well-defined. Figure 1 shows the dependency graph of the specification from Example 2.4.

Note also that if one merges a node n and the nodes n directly depends on, and replaces the function of n with the composition of the functions of the merged nodes, the resulting graph is still a DAG, and the streams computed will be the same. For example in Figure 1 nodes > and *eventCount* could be merged. Such a node is called *computation node* or *node* for short.

TeSSLa Library of Builtin Functions. There are five types of functions in TeSSLa, apart from logical functions: arithmetic functions, aggregations, stream manipulators, timing functions (explained above) and temporal property functions. Tables 1 and 2 show a representative set of the functions provided by TeSSLa and the semantics of some of these functions. See Section A in the appendix for the complete collection of functions with their semantics.

Simple arithmetic functions provide capabilities for performing arithmetic operations on streams. In general, these functions take a set of signals as input and output another signal. Examples include basic arithmetic operations like *add*, *mul*, etc. More complex calculation functions in TeSSLa are *aggregations*, which take event streams as input and output a signal. Examples are *sum* that computes the sum of all events that happened on an event stream, and *eventCount* that counts the events. Another important aggregation function is $mrv : \mathcal{E}_D \times D \to \mathcal{S}_D$ which converts an event stream into a signal that receives the most recent value in the event stream (or a default value of type D provided as second argument).

Sampling functions convert a signal into an event stream. The function $changeOf : \mathcal{S}_D \to \mathcal{E}_D$ returns an event stream with an event at the point in time at which the signal changes. The function $sample : \mathcal{S}_D \times \mathcal{E}_D \to \mathcal{E}_D$ samples a signal by an event stream and returns an event stream with the values obtained from the signal.
Stream manipulators allow to process event streams. Examples include a *filter* operator which allows to delete events and *merge* which fuses two event streams.

The *monitor* function delivers a scope for specifying properties in temporal logics like LTL or SALT [4] with classical or three-valued semantics [5]. Input properties are then compiled into a state machine which can be executed using Boolean signals as propositions and an event stream to step the monitor every time an event happens on this stream (to model discrete inputs).

3 ONLINE EVALUATION OF EFFICIENTLY MONITORABLE SPECIFICATIONS

The semantics provided in the previous section is denotational in the sense that it associates for each complete input valuation, the unique output valuation. We develop now an operational semantics for online monitoring TeSSLa specifications. In this section *we restrict TeSSLa specifications to refer to present and past values only.* These specifications are known as efficiently monitorable [9], and satisfy that the values of an output variable x can immediately be resolved to their unique possible values (by evaluating E_x on the variables lower in the dependency graph) when a new input is processed. This fact leads to an online algorithm, which is implemented in the evaluation engines presented in this section. TeSSLa specifications are compiled into a single monitor that receives multiple inputs from the system under observation (each input is called a source and is associated with an input stream of the TeSSLa specification). Recall that each source can send events at different speeds and with different delays.

We represent behaviors of monitors as transducers on timed finite words, whose input is the stream of events arriving from any source. Let A be an input alphabet and B be an output alphabet (which correspond to the domains of input and output streams). A timed input letter is an element of $(A \times \mathbb{T})$ and a timed output letter is an element of $(B \times \mathbb{T})$. Given a timed letter a we use $t(a)$ to denote its time component. We use $source(a)$ to represent the source of an input letter. We reserve the special symbol \$ (not in A or B) to denote the end of a timed word $a_0 \dots a_n\$$. We use Σ for $(A \times \mathbb{T})$ and Γ for $(B \times \mathbb{T})$, and ε for the empty word. Given a word w we use $L(w)$ for the letters occurring in w and $pos(a)$ for the position of a in w. The time-stamps of letters model the *system time*, while the position of a letter in the world models the *monitoring time*. We will show that every execution of a TeSSLa specification generates the exact same output if each input stream is the same even if the streams are non-sychronized, as long as these streams are in-order.

Definition 3.1 (In-order & Synchronized Inputs). We say that an input word w is
- *in-order* whenever for every $a, b \in L(w)$ if $pos(a) < pos(b)$ and $source(a) = source(b)$ then $t(a) \leq t(b)$.
- *synchronized* whenever for every $a, b \in L(w)$ if $pos(a) < pos(b)$ then $t(a) \leq t(b)$.

Example 3.2. Consider two input sources, one receives (T, 0)(T, 3) and another receives (F, 1)(F, 6). The following two inputs
- w_1 : (T, 0)(F, 1)(T, 3)(F, 6) and
- w_2 : (T, 0)(F, 1)(F, 6)(T, 3)

are in-order inputs for these sources. However, w_1 is synchronized but w_2 is not, because in w_2 (T, 3) is received after (F, 6). The input w_2 is in-order because the input sources are different for the letters received in reverse order of time-stamps. ■

3.1 Evaluation Engines

We introduce *evaluation engines* to define the operational semantics and reason about the correctness of different implementations of TeSSLa online monitors. Consider a dependency graph G for a given specification $\varphi(I, O)$, and let N be the collection of nodes of G. Nodes will be implemented by separate execution entities, possibly executing concurrently and asynchronously. Nodes that read some input stream are called *sources nodes*. Nodes communicate by sending timed letters, which we call (internal) *events*. Events are sent along the reversed edges of G. Evaluation engines equip each node with one queue for each of the node's inputs, which stores the non-processed events in that input. Queues support the standard

Arithmetics		Aggregations	Sampling & Filter
$add : \mathcal{S}_N \times \mathcal{S}_N \to \mathcal{S}_N$	$max : \mathcal{S}_N \times \mathcal{S}_N \to \mathcal{S}_N$	$sma : \mathcal{E}_N \times N \to \mathcal{E}_N$	$timestamps : \mathcal{E}_D \to \mathcal{E}_{\mathbb{T}}$
$sub : \mathcal{S}_N \times \mathcal{S}_N \to \mathcal{S}_N$	$min : \mathcal{S}_N \times \mathcal{S}_N \to \mathcal{S}_N$	$maximum : \mathcal{E}_N \times N \to \mathcal{S}_N$	$changeOf : \mathcal{S}_D \to \mathcal{E}_{\{\top\}}$
$mul : \mathcal{S}_N \times \mathcal{S}_N \to \mathcal{S}_N$	$abs : \mathcal{S}_N \to \mathcal{S}_N$	$maximum : \mathcal{S}_N \to \mathcal{S}_N$	$ifThen : \mathcal{E}_{D_1} \times \mathcal{S}_{D_2} \to \mathcal{E}_{D_2}$
$div : \mathcal{S}_N \times \mathcal{S}_{N^+} \to \mathcal{S}_N$	$abs : \mathcal{E}_N \to \mathcal{E}_N$	$minimum : \mathcal{E}_N \times N \to \mathcal{S}_N$	$sample : \mathcal{S}_{D_1} \times \mathcal{E}_{D_2} \to \mathcal{E}_{D_1}$
$gt : \mathcal{S}_N \times \mathcal{S}_N \to \mathcal{S}_{\mathbb{B}}$	$and : \mathcal{S}_{\mathbb{B}} \times \mathcal{S}_{\mathbb{B}} \to \mathcal{S}_{\mathbb{B}}$	$minimum : \mathcal{S}_N \to \mathcal{S}_N$	$filter : \mathcal{E}_D \times \mathcal{S}_{\mathbb{B}} \to \mathcal{E}_D$
$geq : \mathcal{S}_N \times \mathcal{S}_N \to \mathcal{S}_{\mathbb{B}}$	$or : \mathcal{S}_{\mathbb{B}} \times \mathcal{S}_{\mathbb{B}} \to \mathcal{S}_{\mathbb{B}}$	$sum : \mathcal{E}_N \to \mathcal{S}_N$	$ifThenElse : \mathcal{S}_{\mathbb{B}} \times \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_D$
$leq : \mathcal{S}_N \times \mathcal{S}_N \to \mathcal{S}_{\mathbb{B}}$	$not : \mathcal{S}_{\mathbb{B}} \to \mathcal{S}_{\mathbb{B}}$	$eventCount : \mathcal{E}_{D_1} \times \mathcal{E}_{D_2} \to \mathcal{S}_{\mathbb{N}}$	$occursAny : \mathcal{E}_{D_1} \times \mathcal{E}_{D_2} \to \mathcal{E}_{\{\top\}}$
$eq : \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_{\mathbb{B}}$	$neg : \mathcal{E}_{\mathbb{B}} \to \mathcal{E}_{\mathbb{B}}$	$mrv : \mathcal{E}_D \times D \to \mathcal{S}_D$	$occursAll : \mathcal{E}_{D_1} \times \mathcal{E}_{D_2} \to \mathcal{E}_{\{\top\}}$
$N \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\},\ N^+ = N \setminus \{0\}$			$merge : \mathcal{E}_D \times \mathcal{E}_D \to \mathcal{E}_D$

Table 1: A representative collection of the set of builtin functions.

$$add(\sigma_1, \sigma_2)(t) = \sigma_1(t) + \sigma_2(t)$$
$$div(\sigma_1, \sigma_2)(t) = \frac{\sigma_1(t)}{\sigma_2(t)}$$
$$ge(\sigma_1, \sigma_2)(t) = \sigma_1(t) > \sigma_2(t)$$
$$max(\sigma_1, \sigma_2)(t) = \max\{\sigma_1(t), \sigma_2(t)\}$$
$$not(\sigma)(t) = \neg\sigma(t)$$
$$neg(\eta)(t) = \begin{cases} \neg\eta(t) & \text{if } t \in E(\eta) \\ \bot & \text{otherwise} \end{cases}$$
$$timestamps(\eta)(t) = \begin{cases} t & \text{if } t \in E(\eta) \\ \bot & \text{otherwise} \end{cases}$$
$$mrv(\eta, d)(t) = \begin{cases} \eta(\max E(\eta) \cap [0, t]) & \text{if } E(\eta) \cap [0, t] \neq \emptyset \\ d & \text{otherwise} \end{cases}$$
$$changeOf(\sigma)(t) = \begin{cases} \top & \text{if } t \in \Delta(\sigma) \\ \bot & \text{otherwise} \end{cases}$$
$$sample(\sigma, \eta) = ifThen(\eta, \sigma)$$
$$filter(\eta, \sigma)(t) = \begin{cases} \eta(t) & \text{if } t \in E(\eta) \text{ and } \sigma(t) = true \\ \bot & \text{otherwise} \end{cases}$$
$$merge(\eta_1, \eta_2)(t) = \begin{cases} \eta_1(t) & \text{if } t \in E(\eta_1) \\ \eta_2(t) & \text{if } t \in E(\eta_2) \setminus E(\eta_1) \\ \bot & \text{otherwise} \end{cases}$$
$$occursAny(\eta_1, \eta_2)(t) = \begin{cases} \top & \text{if } t \in E(\eta_1) \cup E(\eta_2) \\ \bot & \text{otherwise} \end{cases}$$

Table 2: Semantics for some of the representative builtin functions.

operation for lists to get the head, the tail or to append to the end of the queue.

Formally, an *evaluation engine* is a transition system $E : \langle S, T, s_0 \rangle$, where the set of states S consists of the internal state of each node n, together with the state of each queue. In the initial state $s_0 \in S$ all queues are empty and all internal states of the nodes are set to their initial values.

An evaluation engine can be fed with input events from the input word w, which are placed into the input queues of the corresponding source nodes. During execution the evaluation engine can produce output events in the streams marked as output streams. A *transition* $\tau \in T$ involves the execution of exactly one node. A transition is *enabled* if there are events present in every input queue of the node. *Firing* a transition follows the operational semantics of the TeSSLa operation associated with the node, and consumes at least one event from some of the node's input queues, producing events into the output queues and updating the internal state of the node. The events produced are pushed to the corresponding input queues of the nodes directly depending on the executing node. We add the special transition $\lambda \in T$ for the empty transition where no event is consumed. We use $apply : S \times T \to S$ for the application of a transition to a state of the evaluation engine. The function $node : T \setminus \{\lambda\} \to N$ returns the node involved in a transition. A run is obtained by the repeated application of transitions.

Definition 3.3 (Run). A run of an evaluation engine

$$r = (\lambda, s_0)(\tau_1, s_1)(\tau_2, s_2) \ldots \in (T \times S)^*$$

is a sequence of transitions and states such that for every $i > 1$, $node(\tau_i)$ is enabled at state s_{i-1} and $apply(s_{i-1}, \tau_i) = s_i$.

Given an input $w \in \Sigma^*$ and a run we get the output of the evaluation engine by concatenating all the output events produced in the run.

It is possible that more than one node is enabled in a given state. A *scheduler* chooses a transition to fire among the enabled transitions. While output events produced by any given node are ordered, outputs produced by different nodes may not be ordered, so concatenating output streams does not necessarily lead to a timed order sequence.

Lemma 3.4 (All Runs are Finite). *Let E be an evaluation engine and $w \in \Sigma^*$ be an input. All runs $r \in (T \times S)^*$ of E on w are of finite length.*

3.1.1 Output Completeness. An event carries information about its occurrence, but there is so far no means to convey information about the absence of an event. We introduce extra events, called *progress events* whose only purpose is to inform nodes downstream about absences of events. A node n of an evaluation engine is called *output complete* if whenever n fires it produces at least one event in its output queue, either a real event or a progress event. In order to guarantee that all queues are emptied after a run finishes we extend the operational semantics of all TeSSLa operators to be output complete, while still implement the intended function. To see the importance of output completeness, consider a node n that is not output complete. This node could just consume all inputs without producing any events in its output (e.g a filter whose condition is always false). Every other node m depending directly on n will never be enabled, because the input queue of m coming from n will always be empty. If m has other input queues filled with some events, these queues will not be emptied at the end of the run.

Lemma 3.4 guarantees that the run will terminate (because no node is enabled) but not necessarily that all queues are empty.

Progress events indicate that an input has progressed up to the time-stamp in the progress event, with no value change. It is easy to see that with output complete building blocks all nodes consume at least one input and all nodes generate exactly one output.

3.2 Timed Transducers

In order to prove properties of evaluation engines, in particular that for any given input the scheduler does not affect the events emitted in the output, we introduce a theory of timed asynchronous transducers. The main result is Theorem 3.12 which states that all runs of an engine are observationally equivalent, independently of the scheduler.

A "classical" synchronous transducer is an element of $(\Sigma \times \Gamma)^*$. However, we wish to model asynchronous transducers because we want to decouple the rate of arrival at input sources from the internal execution of the evaluation engine. A *timed transducer* F is $F \subset \Sigma^* \times \Gamma^*$. Our timed transducers will relate every input to some output (possibly ε). A timed transducer F is *complete* if for all $w \in \Sigma^*$ there is some $v \in \Gamma^*$ such that $(w, v) \in F$.

Definition 3.5. A timed transducer F is *strictly deterministic* if for all $w \in \Sigma^*$, and for all $v, v' \in \Gamma^*$, if $(w, v) \in F$ and $(w, v') \in F$, then $v = v'$.

As an example consider a transducer that delays every input letter by one time instant. Such a transducer is complete and strictly deterministic and would translate $(a, 0)(b, 1)(c, 2)$ into $(a, 1)(b, 2)(c, 3)$. However, strict determinism is too fine grained for our purposes, because we want to allow output letters to be produced out of order. We introduce a *timed reordering* function $timed : \Gamma^* \to \Gamma^*$ which reorders a word according to the time-stamps:

$$timed(b_0 \dots b_{i-1} b_i b_{i+1} \dots b_n) = b_i \cdot timed(b_0 \dots b_{i-1} b_{i+1} \dots b_n) \quad \text{if } t(b_i) < t(b_j) \text{ for all } j \neq i$$

Since words are finite, this recursive definition is well-defined. The following notion of asynchronous determinism captures more precisely the deterministic nature of asynchronous evaluation engines.

Definition 3.6 (Asynchronous determinism). A timed transducer F is called asynchronous deterministic if for all $w \in \Sigma^*$ and for all $v, v' \in \Gamma^*$ with $(w, v) \in F$ and $(w, v') \in F$, $timed(v) = timed(v')$.

Asynchronous determinism allows non-deterministic transducers to produce different outputs for the same input as long as the outputs are identical up-to reordering. Another important notion is asynchronous causality. We use $w \sqsubseteq w'$ to denote that w is a prefix of w'.

Definition 3.7 (Asynchronous causality). A timed transducer F is called asynchronous causal if for all $w, w' \in \Sigma^*$ with $w \sqsubseteq w'$ and $v, v' \in \Gamma^*$ with $(w, v) \in F$ and $(w', v') \in F$, $timed(v) \sqsubseteq timed(v')$.

Finally, we introduce observational equivalence between transducers.

Definition 3.8 (Observational Equivalence). Let F and G be two timed transducers over the same input and output alphabets, and let $w \in \Sigma^*$. We say that F and G are observational equivalent, and we write $F \equiv_O G$ whenever for all $v, u \in \Gamma^*$ with $(w, v) \in F$ and $(w, u) \in G$, $timed(v) = timed(u)$.

It is easy to see that observational equivalence is an equivalence relation for asynchronous deterministic transducers, because the definition of $\equiv_O$ is symmetric and transitive, and if F is asynchronous deterministic then $F \equiv_O F$.

3.3 Correctness

Before we give the main result we need some preliminary lemmas.

Lemma 3.9 (Persistence of Enabledness). *Consider a run $r \in (T \times S)^*$. If a node n is enabled in a state s_i of r, then n stays enabled until it gets scheduled. In particular, the run contains a transition which involves n.*

We say that a node n is *independent* of a node m if n is not a descendant of m in the dependency graph. Similarly, let $\tau_1, \tau_2 \in T$ be two transitions. We say that τ_1 is independent of τ_2 whenever $node(\tau_1)$ is independent of $node(\tau_2)$.

Lemma 3.10 (Exchange of Independent Transitions). *Let τ be independent of τ', then the following holds:*
If

$$r_1 = (\lambda, s_0) \dots (\tau_{i-1}, s_{i-1})(\tau\ , s\ )(\tau', s'')(\tau_{i+1}, s_{i+1}) \dots (\tau_l, s_l)$$

is a run then

$$r_2 = (\lambda, s_0) \dots (\tau_{i-1}, s_{i-1})(\tau', s')(\tau\ , s'')(\tau_{i+1}, s_{i+1}) \dots (\tau_l, s_l)$$

is a run.

Definition 3.11 (Distance between Runs). Let $r, r' \in (T \times S)^*$ be two runs of an engine for the same input. Let p be the common prefix between r and r' and let τ be the transition taken in r after p (that is, the first different transition). We define the *distance* between r and r' as $\delta(r, r') = (|r| - |p|, j)$ where j is the position in r' at which τ is taken after p.

Note that this is well-defined because two runs for the same input: (1) are of the same length, because it takes exactly the same number of transitions to empty all the queues in all cases and (2) contain exactly the same transitions, but possibly in different order. For any run $r \in (T \times S)^*$ we get $\delta(r, r) = (0, 0)$.

Let $r = r_0 \dots r_n$ be a run of an engine for a given input w. The *output* of r is $output(r) = o_1 \dots o_n$ where o_i is the output produced by taking τ_i at s_{i-1} (or ε otherwise). The timed transducer defined by an engine E is:

$$\{(w, v) \in (\Sigma^* \times \Gamma^*) \mid \text{there is a run } r \text{ of } E \text{ on } w \text{ with } output(r) = v\}.$$

We are now ready to prove the main result.

Theorem 3.12. *Let E be an engine, $w \in \Sigma^*$ an input and r and r' two runs of E on w. Then $timed(output(r)) = timed(output(r'))$.*

The proof of Theorem 3.12 proceeds by induction on the distance δ between runs defined above proving that all runs are equivalent using Lemma 3.10.

Corollary 3.13. *All evaluation engines are asynchronous deterministic and asynchronous causal.*

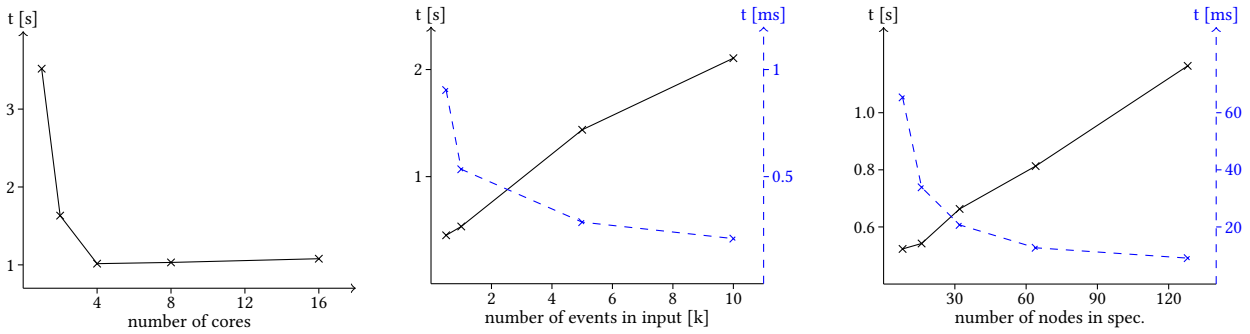


Figure 2: Benchmarks. The black solid plot indicates the total time and the blue dashed line indicates the relative time per event or node, resp.

Theorem 3.12 allows to conclude that the scheduler has no impact on the output of an engine, up-to timed reorderings of the output. Hence, one can take a very deterministic scheduler to reason about the outcome, because every other scheduler is guaranteed to produce an observationally equivalent output (for the same input). For example, a simple reverse topological order allows to easily prove that the resulting deterministic transducer corresponds to the intended denotational semantics of the corresponding TeSSLa specification.

4 IMPLEMENTATION AND EVALUATION

We report here an empirical evaluation of an implementation of the TeSSLa evaluation engine[3] . Our implementation consists of two parts. First, a *compiler* translates a TeSSLa specification into a dependency graph (and performs type checking, macro expansion, and type inference for the defined streams, and also checks that the specification is non-recursive). Then, the *evaluation engine*, written in Elixir, takes an input trace and the dependency graph generated by the compiler and produces an output trace. Elixir, built on top of Erlang, is based on the actor model [19] which allows to deploy code over multiple cores or even distributed systems. Our implementation maps nodes to actors. Theorem 3.12 guarantees that the outcome of an execution is independent on the concrete execution of the Erlang scheduler. Additionally, we also implemented auxiliary tools to use TeSSLa for the runtime verification of C programs based on a software instrumentation. See Section C in the appendix for more details. To evaluate the performance of our implementation we created several artificial benchmarks. We measured the execution time in relation to the number of processor cores, the length of the input trace and the size of the specification. All benchmarks were performed on the same machine with up to 16 cores and 48GB of RAM. The results displayed in Figure 2 are the average of 20 runs (details appear in Appendix D).

Number of cores. For this benchmark we created a TeSSLa specification with 16 computation nodes, consisting of a chain of *abs* functions. The input trace contains exactly 10, 000 events. The results show that the execution time decreases drastically with an increase on the number of cores available, suggesting that our implementation is able to make an effective use of parallelism even though the dependency graph is completely linear. Our asynchronous implementation is able to exploit parallelism automatically in a pipeline fashion.

Number of events in the input. To study the dependency of the execution time with the input length we modified the specification with different input sizes. We also display (with a dash line) the average time per event. With an increase on the input length, the static overhead is amortized quickly and the length of the trace becomes less relevant for the average event time. Consequently, the relative time shows a decay as more events are added.

Number of nodes in the specification. The execution time also grows linearly in the size of the specification. For this benchmark we used input traces of 1, 000 events and increased the number of computation nodes in the specification by adding more calls to *abs* to the composition chain. Again, the relative time per event shows a decay as more nodes are added, because the static overhead becomes less relevant.

5 CONCLUSION

We presented TeSSLa, a stream-based runtime verification language for non-synchronized streams of piece-wise constant real-time signals. We defined the operational semantics of TeSSLa in terms of evaluation engines. We defined a version of timed transducers to prove that all evaluations of a TeSSLa specification produce the same output up-to timed reordering, independently of the scheduler. This result enables different evaluation engines, including asynchronous evaluation engines based on actors—which allow to exploit multi-core parallelism—and evaluation engines implemented in FPGAs which enable the utilization of massive hardware parallelism. We report in this paper an implementation of an evaluation engine written in Elixir/Erlang.

For simplicity, our timed transducers operate on finite words, but all definitions and results can be extended to infinite words under the assumption of a fair scheduler. Similarly, the interleaving semantics of the evaluation engine can be extended to true concurrency between independent nodes. We are currently working on an extension of the TeSSLa language that allows to define the time dependencies between streams explicitly (in a controlled way) and a library of builtin functions on top of the resulting core language.

[3]Tools available at www.isp.uni-luebeck.de/tessla

ACKNOWLEDGMENTS

We thank Jannis Harder and Sebastian Hungerecker for their work on TeSSLa and its compiler.

REFERENCES

[1] Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. 1996. The benefits of relaxing punctuality. *J. ACM* (1996).
[2] Eugene Asarin, Paul Caspi, and Oded Maler. 2002. Timed regular expressions. *J. ACM* 49, 2 (2002), 172–206.
[3] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. 2004. Rule-Based Runtime Verification. In *Proc. of VMCAI'04 (LNCS 2937)*. Springer, 44–57.
[4] Andreas Bauer and Martin Leucker. 2011. The Theory and Practice of SALT. In *NASA Formal Methods (NFM)*. Springer, 13–40.
[5] Andreas Bauer, Martin Leucker, and Chrisitan Schallhart. 2011. Runtime Verification for LTL and TLTL. *ACM T. Softw. Eng. Meth.* 20, 4 (2011), 14.
[6] Gérard Berry. 2000. *Proof, language, and interaction: essays in honour of Robin Milner*. MIT Press, Chapter The foundations of Esterel, 425–454.
[7] Laura Bozelli and César Sánchez. 2014. Foundations of Boolean Stream Runtime Verification. In *In Proc. RV'14 (LNCS)*, Vol. 8734. Springer, 64–79.
[8] Paul Caspi and Marc Pouzet. 1996. Synchronous Kahn Networks. In *Proc. of ICFP'96*. ACM Press, 226–238.
[9] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. 2005. LOLA: Runtime Monitoring of Synchronous Systems. In *Proc. of TIME'05*. IEEE, 166–174.
[10] Normann Decker, Philip Gottschling, Christian Hochberger, Martin Leucker, Torben Scheffel, Malte Schmitz, and Alexander Weiss. 2017. Rapidly Adjustable Non-Intrusive Online Monitoring for Multi-core Systems. In *20th Brazilian Symposium on Formal Methods (SBMF 2017)*. Springer.
[11] Alexandre Donzé, Oded Maler, Ezio Bartocci, Dejan Nickovic, Radu Grosu, and Scott A. Smolka. 2012. On Temporal Logic and Signal Processing. In *In Proc. of ATVA'12 (LNCS)*, Vol. 7561. Springer, 92–106.
[12] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. 2003. Reasoning with Temporal Logic on Truncated Paths. In *Proc. of CAV'03 (LNCS 2725)*, Vol. 2725. Springer, 27–39.
[13] Conal Eliot and Paul Hudak. 1997. Functional Reactive Animation. In *Proc. of ICFP'07*. ACM, 163–173.
[14] Thierry Gautier, Paul Le Guernic, and Löic Besnard. 1987. SIGNAL: A declarative language for synchronous programming of real-time systems. In *Proc. of FPCA'87 (LNCS 274)*. Springer, 257–277.
[15] Alwyn E. Goodloe and Lee Pike. 2010. *Monitoring distributed real-time systems: A survey and future directions*. Technical Report. NASA Langley Research Center.
[16] Nicolas Halbwachs, Paul Caspi, D. Pilaud, and J.A. Plaice. 1987. Lustre: a declarative language for programming synchronous systems. In *Proc. of POPL'87*. ACM Press, 178–188.
[17] Klaus Havelund and Allen Goldberg. 2005. Verify your runs. In *Proc. of VSTTE'05 (LNCS 4171)*. Springer, 374–383.
[18] Klaus Havelund and Grigore Roşu. 2002. Synthesizing Monitors for Safety Properties. In *Proc. of TACAS'02 (LNCS 2280)*. Springer, 342–356.
[19] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. *IJCAI* (1973), 235–245.
[20] Stefan Jaksic, Ezio Bartocci, Radu Grosu, Reinhard Kloibhofer, Thang Nguyen, and Dejan Nickovic. 2015. From signal temporal logic to FPGA monitors. In *Proc. of MEMOCODE 2015*. 218–227.
[21] Stefan Jaksic, Ezio Bartocci, Radu Grosu, and Dejan Nickovic. 2016. Quantitative Monitoring of STL with Edit Distance. In *Prov. of RV'16 (LNCS)*, Vol. 10012. 201–218.
[22] Martin Leucker. 2011. Teaching Runtime Verification. In *Proc. of RV'11 (LNCS)*. Springer, 34–48.
[23] Martin Leucker and Christian Schallhart. 2009. A Brief Account of Runtime Verification. *J. Logic Algebr. Progr.* 78, 5 (2009), 293–303.
[24] Oded Maler and Dejan Nickovic. 2004. Monitoring Temporal Properties of Continuous Signals. In *FTRTFT*. 152–166.
[25] Zohar Manna and Amir Pnueli. 1995. *Temporal Verification of Reactive Systems: Safety*. Springer, New York.
[26] Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. 2010. Copilot: A Hard Real-Time Runtime Monitor. In *Proc. of RV'10 (LNCS 6418)*. Springer.
[27] Amir Pnueli and Aleksandr Zaks. 2006. PSL Model Checking and Run-Time Verification Via Testers. In *Proc. of FM'06 (LNCS 4085)*. Springer, 573–586.
[28] Grigore Roşu and Klaus Havelund. 2005. Rewriting-Based Techniques for Runtime Verification. *Automated Software Engineering* 12, 2 (2005), 151–197.
[29] Konstantin Selyunin, Stefan Jaksic, Thang Nguyen, Christian Reidl, Udo Hafner, Ezio Bartocci, Dejan Nickovic, and Radu Grosu. 2017. Runtime Monitoring with Recovery of the SENT Communication Protocol. In *Proc. of CAV'17 (LNCS)*, Vol. 10426. Springer, 336–355.
[30] Konstantin Selyunin, Thang Nguyen, Ezio Bartocci, and Radu Grosu. 2016. Applying Runtime Monitoring for Automotive Electronic Development. In *Proc. of RV'16 (LNCS)*, Vol. 10012. 462–469.
[31] Koushik Sen and Grigore Roşu. 2003. Generating Optimal Monitors for Extended Regular Expressions. *ENTCS* 89, 2 (2003), 226–245.

A FULL SEMANTICS OF TESSLA LIBRARY OF BUILTIN FUNCTIONS

A.1 Denotational Semantics

Note that the semantics for the timing functions *delay*, *shift* and *within* were already discussed in the paragraph *Semantics of Timing Functions* in Section 2 and are not repeated in the following list.

$$\begin{aligned} add &: \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_D, \quad D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\} \\ add &(\sigma_1, \sigma_2)(t) := \sigma_1(t) + \sigma_2(t) \end{aligned}$$

$$\begin{aligned} sub &: \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_D, \quad D \in \{\mathbb{Z}, \mathbb{R}\} \\ sub &(\sigma_1, \sigma_2)(t) := \sigma_1(t) - \sigma_2(t) \end{aligned}$$

$$\begin{aligned} mul &: \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_D, \quad D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\} \\ mul &(\sigma_1, \sigma_2)(t) := \sigma_1(t) \cdot \sigma_2(t) \end{aligned}$$

$$\begin{aligned} div &: \mathcal{S}_D \times \mathcal{S}_{D'} \to \mathcal{S}_D, \quad D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\}, \quad D' = D \setminus \{0\} \\ div &(\sigma_1, \sigma_2)(t) := \frac{\sigma_1(t)}{\sigma_2(t)} \end{aligned}$$

$$\begin{aligned} gt &: \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_{\mathbb{B}}, \quad D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\} \\ ge &(\sigma_1, \sigma_2)(t) := \sigma_1(t) > \sigma_2(t) \end{aligned}$$

$$\begin{aligned} geq &: \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_{\mathbb{B}}, \quad D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\} \\ geq &(\sigma_1, \sigma_2)(t) := \sigma_1(t) \geq \sigma_2(t) \end{aligned}$$

$$\begin{aligned} leq &: \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_{\mathbb{B}}, \quad D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\} \\ leq &(\sigma_1, \sigma_2)(t) := \sigma_1(t) \leq \sigma_2(t) \end{aligned}$$

$$\begin{aligned} eq &: \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_{\mathbb{B}}, \quad \text{any } D \text{ with equality} \\ eq &(\sigma_1, \sigma_2)(t) := \sigma_1(t) = \sigma_2(t) \end{aligned}$$

$$\begin{aligned} max &: \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_D, \quad D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\} \\ max &(\sigma_1, \sigma_2)(t) := \max\{\sigma_1(t), \sigma_2(t)\} \end{aligned}$$

$$\begin{aligned} min &: \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_D, \quad D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\} \\ min &(\sigma_1, \sigma_2)(t) := \min\{\sigma_1(t), \sigma_2(t)\} \end{aligned}$$

$$\begin{aligned} abs &: \mathcal{S}_D \to \mathcal{S}_D, \quad D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\} \\ abs &(\sigma)(t) := |\sigma(t)| \end{aligned}$$

$$\begin{aligned} abs &: \mathcal{E}_D \to \mathcal{E}_D, \quad D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\} \\ abs &(\eta)(t) := \begin{cases} |\eta(t)| & \text{if } t \in E(\eta) \\ \bot & \text{otherwise} \end{cases} \end{aligned}$$

$$\begin{aligned} and &: \mathcal{S}_{\mathbb{B}} \times \mathcal{S}_{\mathbb{B}} \to \mathcal{S}_{\mathbb{B}} \\ and &(\sigma_1, \sigma_2)(t) := \sigma_1(t) \wedge \sigma_2(t) \end{aligned}$$

$$\begin{aligned} or &: \mathcal{S}_{\mathbb{B}} \times \mathcal{S}_{\mathbb{B}} \to \mathcal{S}_{\mathbb{B}} \\ or &(\sigma_1, \sigma_2)(t) := \sigma_1(t) \vee \sigma_2(t) \end{aligned}$$

$$\begin{aligned} not &: \mathcal{S}_{\mathbb{B}} \to \mathcal{S}_{\mathbb{B}} \\ not &(\sigma)(t) := \neg\sigma(t) \end{aligned}$$

$$\begin{aligned} neg &: \mathcal{E}_{\mathbb{B}} \to \mathcal{E}_{\mathbb{B}} \\ neg &(\eta)(t) := \begin{cases} \neg\eta(t) & \text{if } t \in E(\eta) \\ \bot & \text{otherwise} \end{cases} \end{aligned}$$

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

$$\begin{aligned} maximum &: \mathcal{E}_D \times D \to \mathcal{S}_D \\ maximum &(\eta, d)(t) := \max(\{d\} \cup \{\eta(t') \mid t' \in E(\eta), t' \leq t\}) \end{aligned}$$

$$\begin{aligned} maximum &: \mathcal{S}_D \to \mathcal{S}_D \\ maximum &(\sigma)(t) := \max\{\eta(t') \mid t' \in \mathbb{T}, t' \leq t\} \end{aligned}$$

$$\begin{aligned} minimum &: \mathcal{E}_D \times D \to \mathcal{S}_D \\ minimum &(\eta, d)(t) := \min(\{d\} \cup \{\eta(t') \mid t' \in E(\eta), t' \leq t\}) \end{aligned}$$

$$\begin{aligned} minimum &: \mathcal{S}_D \to \mathcal{S}_D \\ minimum &(\sigma)(t) := \min\{\eta(t') \mid t' \in \mathbb{T}, t' \leq t\} \end{aligned}$$

$$\begin{aligned} sum &: \mathcal{E}_D \to \mathcal{S}_D \\ sum &(\eta)(t) := \Sigma_{\{t' \in E(\eta) \mid t' \leq t\}} \eta(t') \end{aligned}$$

$$\begin{aligned} eventCount &: \mathcal{E}_{D_1} \times \mathcal{E}_{D_2} \to \mathcal{S}_{\mathbb{N}} \\ eventCount &(\eta_1, \eta_2)(t) := \\ &|\{t' \in E(\eta_1) \mid t' \leq t \wedge \forall t'' \leq t \in E(\eta_2) : t' > t''\}| \end{aligned}$$

$$\begin{aligned} mrv &: \mathcal{E}_D \times D \to \mathcal{S}_D \\ mrv &(\eta, d)(t) := \\ &\begin{cases} \eta(\max E(\eta) \cap [0, t]) & \text{if } E(\eta) \cap [0, t] \neq \emptyset \\ d & \text{otherwise} \end{cases} \end{aligned}$$

$$sma : \mathcal{E}_D \times \mathbb{N} \to \mathcal{E}_D, \quad D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\}$$

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

$$changeOf : \mathcal{S}_D \to \mathcal{E}_{\{\top\}}$$

$$changeOf(\sigma)(t) := \begin{cases} \top & \text{if } t \in \Delta(\sigma) \\ \bot & \text{otherwise} \end{cases}$$

$$ifThen : \mathcal{E}_{D_1} \times \mathcal{S}_{D_2} \to \mathcal{E}_{D_2}$$

$$ifThen(\eta, \sigma)(t) := \begin{cases} \sigma(t) & \text{if } t \in E(\eta) \\ \bot & \text{otherwise} \end{cases}$$

$$sample : \mathcal{S}_{D_1} \times \mathcal{E}_{D_2} \to \mathcal{E}_{D_1}$$

$$sample(\sigma, \eta) := ifThen(\eta, \sigma)$$

$$filter : \mathcal{E}_D \times \mathcal{S}_{\mathbb{B}} \to \mathcal{E}_D$$

$$filter(\eta, \sigma)(t) := \begin{cases} \eta(t) & \text{if } t \in E(\eta) \\ & \text{and } \sigma(t) = true \\ \bot & \text{otherwise} \end{cases}$$

$$ifThenElse : \mathcal{S}_{\mathbb{B}} \times \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_D$$

$$ifThenElse(\sigma_1, \sigma_2, \sigma_3)(t) := \begin{cases} \sigma_2(t) & \text{if } \sigma_1(t) = true \\ \sigma_3(t) & \text{otherwise} \end{cases}$$

$$merge : \mathcal{E}_D \times \mathcal{E}_D \to \mathcal{E}_D$$

$$merge(\eta_1, \eta_2)(t) := \begin{cases} \eta_1(t) & \text{if } t \in E(\eta_1) \\ \eta_2(t) & \text{if } t \in E(\eta_2) \setminus E(\eta_1) \\ \bot & \text{otherwise} \end{cases}$$

$$occursAny : \mathcal{E}_{D_1} \times \mathcal{E}_{D_2} \to \mathcal{E}_{\{\top\}}$$

$$occursAny(\eta_1, \eta_2)(t) := \begin{cases} \top & \text{if } t \in E(\eta_1) \cup E(\eta_2) \\ \bot & \text{otherwise} \end{cases}$$

$$occursAll : \mathcal{E}_{D_1} \times \mathcal{E}_{D_2} \to \mathcal{E}_{\{\top\}}$$

$$occursAll(\eta_1, \eta_2)(t) := \begin{cases} \top & \text{if } t \in E(\eta_1) \cap E(\eta_2) \\ \bot & \text{otherwise} \end{cases}$$

A.2 Operational Semantics

For the operational semantics we use the following definition of a queue, which stores incoming events. We represent an event as an object with attributes *time* and *value*, where value can be either be a concrete value or $\top$ as only instance of Unit or *progress* to indicate a progress event.

Definition A.1 (Queue). A *queue* Q can be accessed with the following functions
- *enqueue*(Q, a) adds a to Q,
- *dequeue*(Q) removes the next element from the queue and returns it,
- *peek*(Q) returns the same as *dequeue*(Q) without changing Q,
- *last*(Q) returns the last element which was dequeued from Q.

Note that our queues remember the last element dequeued, which can be accessed using *last*.

We now present the operational semantics of the TeSSLa functions listed in the last section. The code is executed if the associated computation node is enabled, i.e. if for all input queues Q we have *peek*$(Q) \neq$ *nil*. We use the statement *emit*(a) to indicate that a is send to the output, that is a is enqueued in the corresponding input queues of nodes corresponding to functions which are directly depending on the running node in the dependency graph.

A.2.1 Binary Arithmetic Functions. For $add : \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_D$ we have two input queues A and B and the following code:

```
1  if peek(A).time = peek(B).time then
2    a = peek(A).value, b = peek(B).value
3    t = peek(A).time
4    if a = progress then a = last(A).value
5    if b = progress then b = last(B).value
6    dequeue(B)
7  else if peek(A).time < peek(B).time then
8    a = peek(A).value, b = last(B).value
9    t = peek(A).time
10   if a = progress then a = last(A).value
11   dequeue(A)
12 else
13   a = last(A).value, b = peek(B).value
14   t = peek(B).time
15   if b = progress then b = last(B).value
16   dequeue(B)
17 emit(value = a + b, time = t)
```

Note that for the progress events we assume an automatic storage and proper initialization of the last value, e.g. for a queue Q iff *peek*(Q).*value* = *progress* we have the following implicit behavior for a call of *dequeue*(Q):

```
tmp = last(Q).value
dequeue(Q)
last(Q).value = tmp
```

Furthermore, for signals the command *emit* does only emit the first event if called multiple times in a row with exactly the same event. Otherwise the above implementation of processing progress events would lead to multiple emission of the same event. Finally, *emit* converts events with a value into progress events if the value is the same as the value of the last emitted event.

By changing the applied arithmetics, the code above can be applied to the following binary operators on signals as well: *sub*, *mul*,

div, *gt*, *geq*, *leq*, *eq*, *max*, *min*, *and* and *or*. This is simply achieved by replacing line 14 as follows.

- For *sub*:

```
14 emit(value = a − b, time = t)
```

- For *mul*:

```
14 emit(value = a · b, time = t)
```

- For *div*:

```
14 emit(value = a/b, time = t)
```

- For *gt*:

```
14 emit(value = a > b, time = t)
```

- For *geq*:

```
14 emit(value = a ≥ b, time = t)
```

- For *leq*:

```
14 emit(value = a ≤ b, time = t)
```

- For *eq*:

```
14 emit(value = a = b, time = t)
```

- For *max*:

```
14 emit(value = max(a, b), time = t)
```

- For *min*:

```
14 emit(value = min(a, b), time = t)
```

- For *and*:

```
14 emit(value = a ∧ b, time = t)
```

- For *or*:

```
14 emit(value = a ∨ b, time = t)
```

A.2.2 Unary Arithmetic Functions. For $abs : \mathcal{S}_D \to \mathcal{S}_D$ we have one input queue A and the following code:

```
1 if peek(A).value = progress then
2     emit(value = progress, time = peek(A).time)
3 else
4     emit(value = |peek(A).value|, time = peek(A).time)
5 dequeue(A)
```

By changing the applied arithmetics, the code above can be applied to the following unary operators as well: $abs : \mathcal{E}_D \to \mathcal{E}_D$, $not : \mathcal{S}_{\mathbb{B}} \to \mathcal{S}_{\mathbb{B}}$, $neg : \mathcal{E}_{\mathbb{B}} \to \mathcal{E}_{\mathbb{B}}$, $timestamps : \mathcal{E}_D \to \mathcal{E}_{\mathbb{T}}$, $mrv : \mathcal{E}_D \times D \to \mathcal{S}_D$ and $changeOf : \mathcal{S}_D \to \mathcal{E}_{\{\top\}}$. This is accomplished by changing line 4 in the code of *abs* with the corresponding operation.

A.2.3 Aggregation Functions. For $sum : \mathcal{E}_D \to \mathcal{S}_D$ we have one input queue A, the internal $state \in D$ initialized with 0 and the following code:

```
1 if peek(A).value = progress then
2     emit(value = progress, time = peek(A).time)
3 else
4     state = state + peek(A).value
5     emit(value = state, time = peek(A).time)
6 dequeue(A)
```

By changing the applied arithmetics, the code above can be applied to the following aggregating operators as well: $maximum : \mathcal{E}_D \times D \to \mathcal{S}_D$, $maximum : \mathcal{S}_D \to \mathcal{S}_D$, $minimum : \mathcal{E}_D \times D \to \mathcal{S}_D$, $minimum : \mathcal{S}_D \to \mathcal{S}_D$ and $sma : \mathcal{E}_D \times D$. This is accomplished by replacing line 4 with the right operation.

For $eventCount : \mathcal{E}_{D_1} \times \mathcal{E}_{D_2} \to \mathcal{S}_{\mathbb{N}}$ we have two input queues A and B, the internal $state \in \mathbb{N}$ initialized with 0 and the following code:

```
1  if peek(A).time = peek(B).time then
2      if peek(A).value = progress and
3          peek(B).value = progress then
4          emit(value = progress, time = peek(A).time)
5      else if peek(B).value = progress then
6          state = state + 1
7          emit(value = state, time = peek(A).time)
8      else
9          state = 0
10         emit(value = 0, time = peek(A).time)
11     dequeue(A), dequeue(B)
12 else if peek(A).time < peek(B).time then
13     if peek(A).value = progress then
14         emit(value = progress, time = peek(A).time)
15     else
16         state = state + 1
17         emit(value = state, time = peek(A).time)
18     dequeue(A)
19 else
20     if peek(B).value = progress then
21         emit(value = progress, time = peek(B).time)
22     else
23         state = 0
24         emit(value = state, time = peek(A).time)
25     dequeue(B)
```

A.2.4 Filtering Functions. For $ifThen : \mathcal{E}_{D_1} \times \mathcal{S}_{D_2} \to \mathcal{E}_{D_2}$ we have two input queues A and B and the following code:

```
1 if peek(A).time = peek(B).time then
2     if peek(A).value = progress then
3         emit(value = progress, time = peek(A.time)
4     else if peek(B).value = progress then
5         emit(value = last(B).value, time = peek(A).time)
6     else
7         emit(value = peek(B).value, time = peek(A).time)
8     dequeue(A), dequeue(B)
9 else if peek(A).time < peek(B).time then
```

```
    if peek(A).value = progress then
      emit(value = progress, time = peek(A.time)
    else
      emit(value = last(B).value, time = peek(A).time)
    dequeue(A)
  else
    emit(value = progress, time = peek(B.time))
    dequeue(B)
```

For *filter* : $\mathcal{E}_D \times \mathcal{S}_{\mathbb{B}} \to \mathcal{E}_D$ we have two input queues A and B and the following code:

```
if peek(A).time = peek(B).time then
  if (peek(B).value = progress and last(B).value) or
      peek(B).value then
    emit(value = peek(A).value, time = peek(A).time)
  else
    emit(value = progress, time = peek(A.time)
  dequeue(A), dequeue(B)
else if peek(A).time < peek(B).time then
  if last(B).value then
    emit(value = peek(A).value, time = peek(A).time)
  else
    emit(value = progress, time = peek(A.time)
  dequeue(A)
else
  emit(value = progress, time = peek(B.time))
  dequeue(B)
```

For *ifThenElse* : $\mathcal{S}_{\mathbb{B}} \times \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_D$ we have three input queues A, B and C, but apart from more cases the operative semantics is very similar to *ifThen* and *filter* above.

For *merge* : $\mathcal{E}_D \times \mathcal{E}_D \to \mathcal{E}_D$ we have two input queues A and B and the following code:

```
if peek(A).time = peek(B).time then
  if peek(A).value = progress then
    emit(value = peek(B).value, time = peek(A).time)
  else
    emit(value = peek(A).value, time = peek(A).time)
  dequeue(A), dequeue(B)
else if peek(A).time < peek(B).time then
  emit(value = peek(A).value, time = peek(A).time)
  dequeue(A)
else
  emit(value = peek(B).value, time = peek(B).time)
  dequeue(B)
```

With slight modifications the code above can be applied to *occursAny* : $\mathcal{E}_{D_1} \times \mathcal{E}_{D_2} \to \mathcal{E}_{\{\top\}}$ and *occursAll* : $\mathcal{E}_{D_1} \times \mathcal{E}_{D_2} \to \mathcal{E}_{\{\top\}}$ as well.

A.2.5 Timing Functions. For *delay* : $\mathcal{S}_D \times \mathbb{T} \times D \to \mathcal{S}_D$ we have one input queues A and a constant $d \in \mathbb{T}$ and the following code:

```
emit(value = peek(A).value, time = peek(A).time + d)
dequeue(A)
```

Taking into account the additional default value this can be extended to *delay* : $\mathcal{E}_D \times \mathbb{T} \to \mathcal{E}_D$ on signals, too.

For *shift* : $\mathcal{E}_D \to \mathcal{E}_D$ we only have one input queue A and the following code:

```
emit(value = last(A).value, time = peek(A).time)
dequeue(A)
```

For *within* : $\mathbb{T} \times \mathbb{T} \times \mathcal{E}_D \to \mathcal{S}_{\mathbb{B}}$ we have one input queue X belonging to a stream $x \in \mathcal{E}_D$ and the two constants $a, b \in \mathbb{T}$, such that we compute *within*(a, b, x). For the operative semantics we assume $a < b < 0$. We then have the following code:

```
if last(X).time + b < peek(X).time + a then
  emit(value = false, time = last(X).time + b)
  emit(value = true, time = peek(X).time + a)
emit(value = progress, time = peek(X).time)
dequeue(X)
```

All the constructs have no loops, so it is easy to see that all constructs consume at least one event from at least one input queue, and produce one event. Moreover, the inputs are consumed in increasing time-stamps and the outputs are also generated in increasing time-stamps. All constructs are output-complete. The operational semantics for all functions except the timing functions only use time-stamps for the input in their outputs. To show that the operational semantics adhere to the denotational semantics one can show that for each building block, the output signal or event stream encodes (with legal progress events) the output of the denotation function on a given input stream. The detail proof then proceeds by induction on the size of the specification, but it is out of the scope of this paper.

B MISSING PROOFS

Lemma B.1 (All Runs are Finite). *Let E be an evaluation engine and let $a \in (I \times \mathbb{T})^*$ be an input. All possible runs $r \in (T \times S)^\infty$ of E on a are finite.*

Proof. Consider an evaluation order $<$ (a reverse topological order of the dependency graph). We create a lexicographic order as tuple with one entry per queue ordered according to $<$. Each entry represents the size of the corresponding queue. For a given input word a the initial value consists of the sizes of the input queues according to a, with all the other entries being 0 (empty queues). Firing a transition consists on decreasing an entry (the size of the queue corresponding to the node) and increasing only entries in lower positions of the lexicographic order. Hence every firing decreases a (well-founded) lexicographic order and the firing relation is terminating. □

Lemma B.2 (Persistence of Enabledness). *Let $r \in (T \times S)^*$ be a run. A node n which is enabled in a state s_i of r stays enabled at least until it gets scheduled, i.e. the run contains a transition which involves n.*

Proof. Let $(\tau_x, s_x) = r_{i+1}$ be the next element in the run after r_i. If $node(\tau_x) \neq n$, then the only influence that τ_x can have on the internal state of n is by appending some event to one of the input queues. Since the input queues of n can only grow, n will stay enabled. By induction on the distance from i to the end of the trace,

n stays enabled at every later step until a transition involving n is taken. □

Lemma B.3 (Exchange of Independent Transitions). *Let τ be independent of τ', then the following holds: If*

$$r_1 = (\lambda, s_0)\dots(\tau_{i-1}, s_{i-1})(\tau\ , s\ )(\tau', s'')(\tau_{i+1}, s_{i+1})\dots(\tau_l, s_l)$$

is a run then

$$r_2 = (\lambda, s_0)\dots(\tau_{i-1}, s_{i-1})(\tau', s')(\tau\ , s'')(\tau_{i+1}, s_{i+1})\dots(\tau_l, s_l)$$

is a run.

Proof. Because τ' is independent of τ, τ' is guaranteed to be enable after s (otherwise it would not be take in r_1. Similarly, τ is enabled at s' because τ' cannot turn false the enabling condition of τ. We just need to show that the state s'' does not change by moving τ' before τ. By definition we know that applying a transition τ only modifies the internal state of $node(\tau)$ and only the queues of $node(\tau)$ and those dependent of $node(\tau)$. Because τ' is independent of τ, we get

$$s_j = apply(apply(s_{i-1}, \tau), \tau') = apply(apply(s_{i-1}, \tau'), \tau).$$

and the results follows. □

Theorem B.4. *Let E be an engine, $a \in \Sigma^*$ an input and r and r' two runs of E on a. Then $timed(output(r)) = timed(output(r'))$.*

Proof. If r and r' are identical we are done. Otherwise, let p be the longest common prefix of r and r' and let τ be the transition in r taken after p. Let j be the first position after p at which τ is taken in r' (this guaranteed to happen because τ is continuously enabled in r' after p). The distance between r and r' is $(|r| - |p|, j)$. We create a run r_3 from r' by swapping (τ, s_{j-1}) and (τ', s_{j-2}), which produces a legal run of E on a by Lemma 3.10. Moreover, the $timed(output(r')) = timed(Output(r''))$ because the queues for τ and τ' are the same in r' and r'', and the states before $j-2$ and after j are the same in both runs. Also, $\delta(r, r'') < \delta(r, r')$. Since δ is a well-founded it follows, repeating the same argument by induction (on $\delta(r, r')$) shows that $timed(output(r)) = timed(output(r'))$ as desired. □

C INSTRUMENTATION & TOOL CHAIN

As mentioned in the introduction, TeSSLa is designed to simplify FPGA based implementations of the evaluation engine. Additionally, we also implemented auxiliary tools to use TeSSLa for the runtime verification of C programs, like a simple software instrumentation tool realized as a compiler pass of the LLVM Compiler Infrastructure. We added the TeSSLa functions *function_call* and *function_return* which generate an input stream with an event every time the function is called or returns, respectively, during the run of the program. This additional TeSSLa functions are preprocessed into a list of functions that need to be instrumented and replaced with input streams for further processing of the specification. This leads to the following tool chain from a TeSSLa specification φ:

(1) Process φ into dependency graph G and identify the functions to instrument.
(2) Compile the C code to LLVM IR code.
(3) Instrument the LLVM IR code and compile the result to an executable.
(4) Run the executable, which generates the input trace r.
(5) Run the TeSSLa evaluation engine with G and r as inputs.

The integrated tool chain is available as an Atom plugin,[4] which is shown in Figure 3 on the next page.

D RUNTIME BENCHMARK DATA

All data from the benchmarks is shown on the next pages in Table 3, Table 4 and Table 5.

[4] www.isp.uni-luebeck.de/tessla

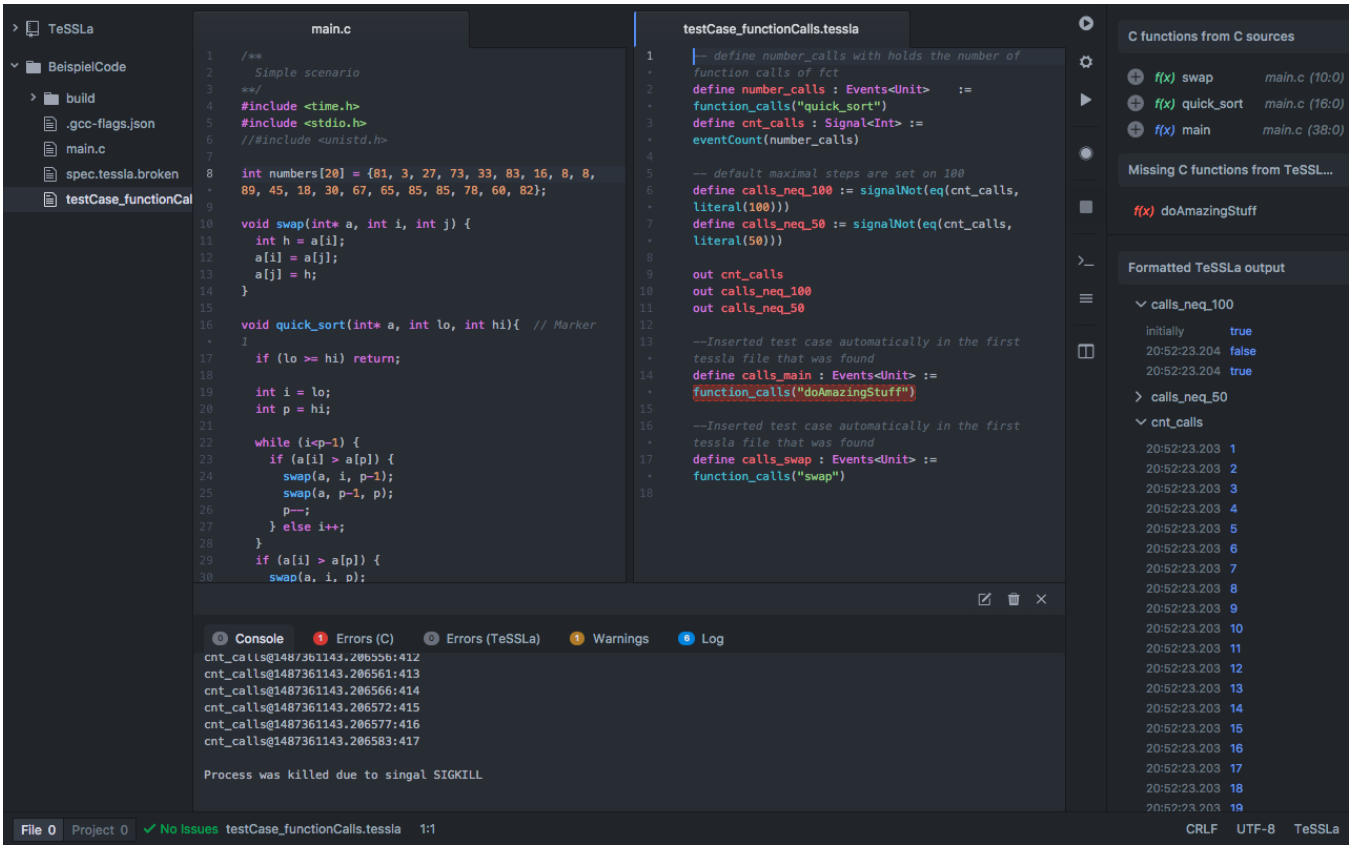


Figure 3: Atom plugin for TeSSLa based runtime verification of C programs.

Run #	1	2	4	8	16
1	3.620	1.598	1.024	0.948	1.010
2	3.483	1.650	0.964	0.974	0.971
3	3.488	1.753	0.989	0.959	0.989
5	3.439	1.597	1.009	1.033	1.082
6	3.521	1.651	1.013	0.931	1.151
7	3.448	1.725	0.976	0.987	0.982
8	3.550	1.648	0.994	0.979	1.098
9	3.496	1.624	1.098	0.967	1.137
10	3.586	1.615	1.039	1.011	1.042
11	3.504	1.634	1.092	1.075	1.101
12	3.443	1.580	0.944	1.003	1.060
13	3.840	1.609	1.017	1.009	1.071
14	3.605	1.579	0.964	1.083	1.189
15	3.499	1.535	0.981	0.997	1.023
16	3.535	1.608	0.989	1.287	1.121
17	3.446	1.739	1.015	1.007	1.129
18	3.505	1.544	1.047	1.175	1.161
19	3.586	1.680	1.013	1.035	1.067
20	3.377	1.611	1.008	1.096	1.100

Table 3: Execution time in seconds of multiple runs with 10 000 input events with different number of used processor cores

Run #	500	1000	5000	10000
1	0.4415	0.5358	1.4555	1.9343
2	0.4569	0.5471	1.4538	1.9907
3	0.4549	0.4996	2.8283	1.9546
5	0.4569	0.4985	1.8618	1.9731
6	0.4603	0.5482	1.3788	1.9725
7	0.4610	0.5489	1.1817	2.0493
8	0.4585	0.5302	1.1973	1.9670
9	0.4407	0.5581	1.3172	2.2681
10	0.4414	0.5407	1.3002	2.0725
11	0.4392	0.5349	1.2226	1.9567
12	0.4652	0.5557	1.2435	1.9850
13	0.4436	0.5462	1.2375	2.0038
14	0.4458	0.5240	1.2271	2.0334
15	0.4480	0.5093	1.2287	1.9577
16	0.4508	0.5071	1.8341	2.0005
17	0.4498	0.5390	1.5943	2.1850
18	0.4820	0.5252	1.4755	2.4828
19	0.4365	0.5576	1.2696	2.5152
20	0.4379	0.5334	1.2343	2.3986

Table 4: Execution time in seconds of multiple runs of with different number of input events

Run #	8	16	32	64	128
1	0.534	0.548	0.611	0.784	1.027
2	0.510	0.533	0.598	0.741	1.039
3	0.501	0.546	0.631	0.718	1.057
5	0.494	0.552	0.594	0.731	1.293
6	0.489	0.563	0.598	0.747	1.231
7	0.521	0.539	1.041	0.737	1.236
8	0.496	0.548	0.676	0.721	1.221
9	0.507	0.541	0.618	0.745	1.262
10	0.483	0.559	0.579	0.748	1.229
11	0.507	0.554	0.579	1.271	1.342
12	0.487	0.539	0.836	0.739	1.060
13	0.507	0.526	0.574	0.720	1.174
14	0.545	0.530	0.586	0.782	1.274
15	0.518	0.563	0.597	0.888	1.218
16	0.561	0.528	0.606	1.214	1.278
17	0.605	0.528	0.602	1.006	1.247
18	0.550	0.557	0.651	0.738	1.043
19	0.550	0.523	0.645	0.750	1.027
20	0.541	0.534	1.051	0.735	0.987

Table 5: Execution time in seconds of multiple runs of in regard to different amount of nodes in a specification