Runtime Verification of Non-synchronized Real-Time Event Streams

Martin Leucker · César Sánchez · Torben Scheffel · Malte Schmitz · Alexander Schramm

Received: date / Accepted: date

Abstract We study the problem of online runtime verification of real-time event streams. 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. We start from specifications in a fragment of the TeSSLa specification language, where streams (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. Specifications 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 the specification language and a concurrent implementation of the evaluation algorithm. The algorithm can tolerate and exploit the asynchronous arrival of events without synchronizing the inputs.

Martin Leucker
Institute for Software Engineering and Programming Languages,
University of Lübeck, Germany
E-mail: leucker@isp.uni-luebeck.de

César Sánchez
IMDEA Software Institute, Spain
E-mail: cesar.sanchez@imdea.org

Torben Scheffel
Institute for Software Engineering and Programming Languages,
University of Lübeck, Germany
E-mail: scheffel@isp.uni-luebeck.de

Malte Schmitz
Institute for Software Engineering and Programming Languages,
University of Lübeck, Germany
E-mail: schmitz@isp.uni-luebeck.de

Alexander Schramm
IMDEA Software Institute, Spain
E-mail: alexander.schramm@imdea.org
Then, we introduce a theory of asynchronous transducers and show a formal proof of the correctness such that every possible run of the monitor implements the semantics. Finally, we report an empirical evaluation of a highly concurrent Erlang implementation of the monitoring algorithm.

1 Introduction

We study the online Runtime Verification of real-time event streams and signals that arrive at different speeds and with different and varying delays to the monitor. Runtime verification (RV) is an applied formal technique for 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. Central problems in runtime verification are (1) how to generate monitors from formal specifications, and (2) how to evaluate these monitors against input traces from the running system. See [21,28] for RV surveys and the recent book [4].

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 and varying delays. This assumption is common, for example, when observing embedded systems or when observing low-level 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 or timed event-streams.

We are interested in online monitoring, which is performed while the system is running (as opposed to offline monitoring through post-mortem analysis of dumped traces). The target application of low-level software analysis of embedded systems also requires non-intrusive monitoring, meaning that the monitoring activity cannot perturb the execution of the system under observation. To achieve non-intrusiveness, the monitoring infrastructure uses some hardware capabilities to obtain run-time information while the concurrent system executes. This information is dispatched to an external monitoring executing infrastructure that executes outline (as opposed to inlining the monitors within the system itself which is common in runtime verification for high-level software). See [26] for a definition and classification of these RV concepts.

The main goal of this paper is to study how to monitor sophisticated properties of continuous piece-wise constant signals efficiently, particularly against systems where each component is a source that can emit events at different speeds and with different latencies. We say that these systems emit “non-synchronized in-order streams”. Since the order and time at which events are processed are both important, we introduce here the distinction between 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, which is used to time-stamp events by the instru-
imentation of the system under analysis. Monitor time refers to the instants at which events arrive at the monitor and when these events are processed in order to produce verdicts. This order depends not only on the implementation of the monitor but also on the arrival time and order of the events.

Event streams from hardware processors come at very high speeds, which imposes the additional requirement of crafting highly efficient monitoring implementations. We explore here software monitors that can exploit the parallelism available in multi-core platforms, while still formally guaranteeing the correctness of the monitors. These monitors must tolerate non-synchronized arrival of events and progress as much as possible with only events from some sources.

Stream Runtime Verification (SRV) is very appealing as an approach to a specification language for our purposes, because the dependencies between streams allow to decompose specifications into components that can be executed concurrently and asynchronously. We use here an acyclic fragment of the TeSSLa specification language [9], an incarnation of SRV for real-time event streams. We call this fragment TeSSLa_{a}. TeSSLa stands for Temporal Stream-based Specification Language, and has already been used for creating monitors in FPGA hardware in [11,12]. Our acyclic fragment TeSSLa_{a} restricts TeSSLa to non-recursive specifications, which means that no cycles are allowed in a specification. The functionality that recursion allows with a few core operators in TeSSLa is encapsulate in TeSSLa_{a} in a collection of building blocks. This allows us to build a simpler asynchronous evaluation algorithm, which results in an efficient evaluation of specifications in TeSSLa_{a}.

Related work. Early specification languages for RV were based on their counterparts in static verification, typically logics like LTL [30] or past LTL adapted for finite paths [5,11,22]. Similar formalisms proposed are based on regular expressions [39], timed regular expressions [2], rule based languages [3], or rewriting [33]. Stream runtime verification, pioneered by the tool LOLA [10], 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 [20] or Signal [17]—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) [15], where reactive behaviors are defined using func-
tional 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, 18, 31] 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 [32] defined these dependencies for LTL. Likewise, the semantics of Signal Temporal Logic (STL) [13, 29] 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].

The specification language TeSSLa that we use extends SRV with support for real-time piece-wise constant signals. Most previous approaches to SRV assume synchronous sampling and synchronous arrivals of events in all input streams. It is theoretically feasible, at least in some cases, to reduce the setting in this paper to synchronous SRV, for example by assuming that all samples are made at instants multiple of a minimum quantum 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. That is, the monitor must be able to execute efficiently both at times of spread events and also under fast bursts. There are extensions of SRV for real-time signals, most notably RTLola [16] and Striver [19]. However, all the monitoring algorithms proposed and implemented for these logics, similarly to full TeSSLa, are not able to exploit concurrency and asynchronous evaluation. Moreover, the correctness of the operational semantics of these formalisms requires synchronous arrivals.

STL has also been used to create monitors on FPGAs [24] and for monitoring in different application areas (see for example [25, 35]). However, the assumptions of STL on the signals is different than ours, 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 the signals we assume here 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. Additionally, our framework can handle much richer data domains of data and verdicts than TREs and STL. TREs have been combined with STL [33] 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) A method for the systematic generation of parallel and asynchronous online monitors for software monitoring TeSSLa specifications. These mon-
itors handle the non-synchronized arrival of events from different input stream sources.

(2) A computational model for proving correctness of asynchronous concurrent monitors, introduced in Section 3, which enables to study a concurrent online evaluation algorithm for TeSSLa specifications.

(3) A prototype implementation developed in Erlang, described in Section 4, and an empirical evaluation.

(4) The precise syntax and semantics of TeSSLaa, an acyclic fragment of TeSSLaa, presented in Section 2, including the core and library functions.

Journal Version. An earlier version of this paper appeared in [27], in the Proceedings of the 33rd Symposium on Applied Computing (SAC’18). This paper contains the following additional contributions: Section 2 now contains a full description of the TeSSLaa language, as well as the formal semantics of each operator and the operational semantics of the implementation of each building block. Section 3 now contains a revisited and extended version of the model of computation that can now handle unbounded streams as \( \omega \)-words. New theorems and full proofs of all results are now presented, including the proof that a fair scheduler is all that is needed to guarantee that all concurrent monitor executions preserve the semantics. The empirical evaluation in Section 4 has also been extended to a larger study that illustrates how the implementation allows to exploit parallelism automatically. Finally, the tool chain is now described in Section 4.2.

2 Syntax and Semantics of TeSSLaa

We describe in this section the real-time specification language TeSSLaa.\(^2\) We first present some preliminaries, and then introduce the syntax and semantics.

2.1 Preliminaries

We use two types of stream models as underlying formalism: piece-wise constant signals and event streams. We use \( T \) 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 verdicts produced by the monitors. In this manner output verdicts can be numerical statistics or complex data collected from the trace.

Definition 1 (Event stream) An event stream is a partial function \( \eta : T \to D \) such that \( E(\eta) := \{ t \in T \mid \eta(t) \text{ is defined} \} \) does not contain bounded infinite subsets.

\(^2\) TeSSLaa is available at [https://www.tessla.io/acyclic](https://www.tessla.io/acyclic)
The set of all event streams is denoted by $E_D$. The set $E(\eta)$ is called the set of event points of $\eta$. When $\eta$ is not defined at a time point $t$, that is $t \in T \setminus E(\eta)$, we write $\eta(t) = \bot$. We use $\top$ as the “unit” value (the only value in a singleton domain). A finite event stream $\eta$ 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.

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. This allows a smoother definition of the functions available in TeSSLa and delivers a more convenient model to the user.

**Definition 2 (Signal)** A signal is a total function $\sigma : T \rightarrow D$ such that the set of change points $\Delta(\sigma) := \{t \in T \mid \exists t' < t : \forall t''. t' < t'' < t : \sigma(t) = \sigma(t'')\}$ does not contain bounded infinite subsets.

The set of all signals is denoted by $S_D$. 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 1** Consider the following streams $e$, $s$ and $e_2$, where $e$ and $e_2$ are interpreted as event streams and $s$ as a signal.

<table>
<thead>
<tr>
<th>Event stream $e$</th>
<th>Events at times</th>
<th>7</th>
<th>2</th>
<th>2</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal $s$</td>
<td>Events at times</td>
<td>0</td>
<td>7</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>Event stream $e_2$</td>
<td>Events at times</td>
<td>7</td>
<td>2</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

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 $e_2$ 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 want to define monitors that run concurrently and asynchronously with the system under analysis, and we want to reason about their correctness. Then, it is very important to distinguish between the time at which an event $e$ happens and that is used to time-stamp the event, and the time at which a given monitor receives the event. We use $t(e)$ for the time of the occurrence of the event in the system, and $rt(e)$ for the time at which $e$ reaches the specified monitor.

**Definition 3 (In-order streams)** A stream $s$ is called in-order whenever for every two events $e$ and $e'$, $t(e) < t(e') \Rightarrow rt(e) < rt(e')$ holds.
2.2 Syntax of TeSSLa,

We begin with an example to illustrate a simple TeSSLa, specification. Spec-
ifications are declarative, defining streams in terms of other streams, and ul-
timately in terms of input streams. Streams marked as out are the verdict of
the monitor and their values will be reported to the user.

**Example 2** Consider the following TeSSLa, specification:

```plaintext
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.

```
Event stream e: 1 0 3 2 4 7 6
Signal s: 5 3 2 0 7 6
Signal comp: ———— ———— ———— ———— ———— ————
Signal allowed: ———— ———— ———— ———— ———— ————
Signal ok: ———— ———— ———— ———— ———— ————
```

When allowed is true, so will be ok. The signal ok will also be true as long
as s is lower than 5. 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

$$spec ::= \text{define } \text{name}[: \text{stype}] ::= \text{teexpr} | \text{out} \text{name} |$$

$$\text{in} \text{name} : \text{stype} | \text{spec} \text{spec}$$

$$\text{teexpr} ::= \text{expr}[: \text{type}]$$

$$\text{expr} ::= \text{name} | \text{literal} | \text{name}(\text{teexpr}, \text{teexpr}^*)$$

$$\text{type} ::= \text{btype} | \text{stype}$$

$$\text{stype} ::= \text{Signal}<\text{btype}> | \text{Events}<\text{btype}>$$

A name is a nonempty string. Basic types $\text{btype}$ cover typical types found in programming and verification like $\text{Int}$, $\text{Float}$, $\text{String}$ or $\text{Bool}$. 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 $\text{in}$ introduces input stream variables, and $\text{define}$ introduces defined stream variables (also called output variables). Given a specification $\varphi$ 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, $I = \{e, s\}$ and $O = \{\text{comp}, \text{allowed}, \text{ok}\}$. The marker $\text{out}$ 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 $\text{literal}$ 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). Available basic types and literal representation are implementation dependent. A flat specification is one such that every defining equation $E_x$ is either a name, a literal or an expression of the form $f(x_1, \ldots, x_n)$ where $x_i$ are all stream names. Every specification can be transformed into an equivalent flat specification by introducing additional variables for each sub-expression.

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

$$name ::= \text{defName} | \text{timingFun} | \text{builtinFun} | \text{macro}$$

$$\text{timingFun} ::= \text{delay} | \text{shift} | \text{within}$$

A $\text{defName}$ is simply a name of a previously defined stream or constant. Timing functions allow to describe timing dependencies between streams. The function $\text{delay}$ delays the values of a signal (or events of an event stream) by a certain amount of time. The function $\text{shift}$ shifts the values of an event stream one unit into the future, that is, the first event becomes the second event, etc. The function $\text{within}$ 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 $\text{fun}$. Macros can be expanded at compile time using their definition on a purely syntactical level because macros are not recursive. Macros can be defined with the following production, which is added to $spec$:

$$\text{macro} ::= \text{fun} \text{name}(\text{name}, \text{name}^*) ::= \text{teexpr} | \text{macro} \text{macro}$$
where the `texpr` can use the names of the macro arguments.

**Example 3** An example of a macro has already been used in the Example 2 because `implies` is not a builtin function. Instead, `implies` is defined by the following macro:

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

The expressivity of TeSSL_a is obtained by the use a set of *builtin functions*. We first define in Section 2.3 the semantics of the temporal core of TeSSL_a, which is enough to define the semantics of TeSSL_a in Section 2.4. Then we give semantics of the non-temporal building blocks in Section 2.5.

### 2.3 Semantics of Timing Functions

There are three timing functions `delay`, `shift` and `within`. The function `delay` is overloaded for signals

\[
\text{delay} : S_D \times T \times D \rightarrow S_D
\]

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

and for event streams

\[
\text{delay} : E_D \times T \rightarrow E_D
\]

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

The function `delay` delays a signal or an event stream by a given amount of time. Since signals must always carry a value, a value `v` is provided as default in case the signal being delayed is fetched outside its domain. For event streams, the occurrence of each event is delayed by the indicated amount of time (and an undefined value is used if the original event is fetched outside the boundaries).

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 \( s_\eta \) be an arbitrary event stream:

\[
s_\eta = (t_0, \eta(t_0))(t_1, \eta(t_1))(t_2, \eta(t_2)) \ldots,
\]

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:

\[
\text{shift} : E_D \rightarrow E_D
\]

\[
\text{shift}(s_\eta) = \begin{cases} 
\epsilon & \text{if } s_\eta = (t_0, \eta(t_0)) \text{ or } s_\eta = \epsilon \\
s_\eta^\rightarrow & \text{otherwise}
\end{cases}
\]
The last timing function is \textit{within}, which already appeared in Example 2. The function \textit{within} produces a Boolean valued signal that captures whether there is an event within the timing window provided:

\[
\text{within} : \mathbb{T} \times \mathbb{T} \times \mathcal{E}_D \to \mathcal{S}_B
\]

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

2.4 Semantics of TeSSLa\(_a\)

We define the semantics of TeSSLa\(_a\) in terms of evaluation models, as commonly done in SRV \cite{10}. The intended meaning of TeSSLa\(_a\) specifications is to define output signals and event streams from input signals and event streams. In case of Boolean valued outputs, these outputs are verdicts that can represent errors, but richer domains can be used to capture richer information like statistics of the execution.

Consider a TeSSLa\(_a\) specification over input variables \(I\) and defined variables \(O\). A \textit{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 \(\sigma_I\) and \(\sigma_O\) 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 \(\sigma_I\) on \(I\) and \(\sigma_O\) on \(O\).

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 \(\sigma\) for each of the variables \(I \cup O\) of a specification \(\varphi(I, O)\), we can give a meaning to each expression \(E\) over variables \(I\) and \(O\), written \(\llbracket E, \sigma \rrbracket\), recursively as follows:

- \text{variable name } (E = \text{name}): 
  \[
  \llbracket \text{name}, \sigma \rrbracket = \sigma(\text{name});
  \]
- \text{literal } (E = l): 
  \[
  \llbracket l, \sigma \rrbracket = [l];
  \]
- \text{function application } (E = f(e_1, \ldots, e_n)): 
  \[
  \llbracket E, \sigma \rrbracket = [f][\llbracket e_1, \sigma \rrbracket, \ldots, \llbracket e_n, \sigma \rrbracket]
  \]

An \textit{evaluation model} of a specification \(\varphi(I, O)\) is a valuation \(\sigma\) for variables \(I\) and \(O\), such that the valuation of every output \(x\) variable coincides with the valuation of its defining equation \(E_x\):

\[
\llbracket x, \sigma \rrbracket = \llbracket E_x, \sigma \rrbracket.
\]

Informally, a valuation \(\sigma\) 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 $\sigma_I$ of input variables $I$ there is a unique valuation $\sigma_O$ of output variables $O$ such that $\sigma_I \cup \sigma_O$ is an evaluation model of $\varphi$. Note that a candidate $\sigma$ to be an evaluation model assigns a signal (or event stream) to each input and output variable. In other words, the semantics we just introduced allow to check whether a candidate output signal assignment is an evaluation model for the given input signals. We will give in Section 3 an iterative algorithm to compute, for a given specification, the (unique) output for a given input.

Non-recursive specifications In order to guarantee that every specification is well-defined, we restrict legal TeSSLa specifications such that no variable $x$ can depend circularly 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 $x \rightarrow y$. We say that $x$ depends on $y$ if $x \rightarrow^+ y$ (where $\rightarrow^+$ is the transitive closure of $\rightarrow$). The dependency relation $x \rightarrow^+ y$ 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 $x$, $x \not\rightarrow^+ x$), 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.

---

**Fig. 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.
be a possible future extension. delays extracted from values of input signals, which are only known for well-formedness is not required. A specification in TeSSLa restricted to be constants, because the analysis of dependency cycles performed the delay dependencies between one variable and another need not be restricted to be constants, because the analysis of dependency cycles performed well-formedness is not required. A specification in TeSSLa can allow, for example, delays extracted from values of input signals, which are only known dynamically. This is currently not reflected in the semantics above, but could be a possible future extension.

In Section 3 we will use the dependency graph to define operational semantics of an evaluation engine for TeSSLa specifications. 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 \( \text{eventCount} \) could be merged. Such a node is called computation node or node for short. A node either corresponds to a single function or to multiple composed functions in the TeSSLa specification. LOFA [10] allows recursive specifications at the price of definedness (meaning not all specification have a unique evaluation model). The main result in [10] concerning well-definedness of synchronous specifications is that well-formed specifications are well-defined. A well-formed stream specification is such that all recursive dependencies of a variable \( x \) must be either all strictly forward or all strictly backwards. In TeSSLa, however, all specifications are well-defined because dependencies are guaranteed to be non-recursive. This apparent limitation however, allows the following expressive power to TeSSLa. In TeSSLa, the delay dependencies between one variable and another need not be restricted to be constants, because the analysis of dependency cycles performed for well-formedness is not required. A specification in TeSSLa can allow, for example, delays extracted from values of input signals, which are only known dynamically. This is currently not reflected in the semantics above, but could be a possible future extension.

| \( \text{add} : \mathcal{S}_D \times \mathcal{S}_D \rightarrow \mathcal{S}_D, \quad D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\} \) | \( \text{sub} : \mathcal{S}_D \times \mathcal{S}_D \rightarrow \mathcal{S}_D, \quad D \in \{\mathbb{Z}, \mathbb{R}\} \) |
| \( \text{mul} : \mathcal{S}_D \times \mathcal{S}_D \rightarrow \mathcal{S}_D, \quad D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\} \) | \( \text{div} : \mathcal{S}_D \times \mathcal{S}_{D'} \rightarrow \mathcal{S}_D, \quad D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\} \) |
| \( \text{gt} : \mathcal{S}_D \times \mathcal{S}_D \rightarrow \mathcal{S}_B, \quad D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\} \) | \( \text{geq} : \mathcal{S}_D \times \mathcal{S}_D \rightarrow \mathcal{S}_B, \quad D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\} \) |
| \( \text{lt} : \mathcal{S}_D \times \mathcal{S}_D \rightarrow \mathcal{S}_B, \quad D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\} \) | \( \text{leq} : \mathcal{S}_D \times \mathcal{S}_D \rightarrow \mathcal{S}_B, \quad D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\} \) |
| \( \text{max} : \mathcal{S}_D \times \mathcal{S}_D \rightarrow \mathcal{S}_B, \quad D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\} \) | \( \text{min} : \mathcal{S}_D \times \mathcal{S}_D \rightarrow \mathcal{S}_D, \quad D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\} \) |
| \( \text{abs} : \mathcal{S}_D \rightarrow \mathcal{S}_D, \quad D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\} \) | \( \text{abs} : \mathcal{S}_D \rightarrow \mathcal{S}_D, \quad D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\} \) |

Fig. 2 Arithmetic operations with their semantics.
and : \mathcal{S}_D \times \mathcal{S}_D \rightarrow \mathcal{S}_D

or : \mathcal{S}_D \times \mathcal{S}_D \rightarrow \mathcal{S}_D

not : \mathcal{S}_D \rightarrow \mathcal{S}_D

neg (\eta)(t) := \begin{cases} \neg \eta(t) & \text{if } t \in E(\eta) \\
\bot & \text{otherwise} \end{cases}

sum : \mathcal{E}_D \rightarrow \mathcal{S}_D

\quad \sum (\eta)(t) := \begin{cases} \sum_{t' \in E(\eta), t' \leq t} \eta(t') & \text{if } t \in E(\eta) \\
\bot & \text{otherwise} \end{cases}

mrv : \mathcal{E}_D \times D \rightarrow \mathcal{S}_D

\quad \text{mrv} (\eta,d)(t) := \begin{cases} \eta(\text{max} E(\eta) \cap [0,t]) & \text{if } E(\eta) \cap [0,t] \neq \emptyset \\
\bot & \text{otherwise} \end{cases}

eventCount : \mathcal{E}_D \times \mathcal{E}_D \rightarrow \mathcal{S}_D

\quad \text{eventCount} (\eta_1, \eta_2)(t) := | \{ t' \in E(\eta_1) \mid t' \leq t \land \forall t'' \leq t \in E(\eta_2) : t' > t'' \}|
Sampling functions convert a signal into an event stream. The function \( \text{changeOf} : \mathcal{S}_D \to \mathcal{E}(\top) \) returns an event stream with an event at the point in time at which the signal changes. The function \( \text{sample} : \mathcal{S}_D \times \mathcal{E}_D \to \mathcal{E}_D \) 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 \( \text{filter} \) operator which allows to delete events and \( \text{merge} \) which fuses two event streams.

**Example 4** Consider the following specification:

```plaintext
in open1: Events<Unit>
in open2: Events<Unit>
in close: Events<Unit>
define numberClose := eventCount(close)
define numberOpen := eventCount(merge(open1, open2))
define error := gt(numberClose, numberOpen)
out error
```

The specification is about the opening and closing of files. We assume that there are two functions that open a file and one which closes a file in the system under analysis. The specification establishes that the method which closes files should not be called more often than the two which open files together. In the first three lines, the functions to observe are declared. In the fourth line, the number of closing events is counted and in line five, the two open event streams are merged before the number of opens is counted. In line six it is checked if the number of close events is larger than the number of open
events and in line seven, the signal which is \textit{true} as long as too many close events occurred (hence, which is \textit{true} if an error happened), is the output.

![Stream Diagram]

The stream picture above shows a possible run.

3 Online Evaluation of Efficiently Monitorable Specifications

In runtime verification \cite{10,28} there is a distinction between online and offline evaluation. In online RV the monitor must respond after each stimulus while in offline evaluation the monitor has all the trace at its disposal, like in post-mortem analysis.

The semantics provided in the previous section is denotational in the sense that these semantics allows to check whether an input valuation and an output candidate valuation satisfy the specification. Even though the well-definedness of specifications guarantees that there is a unique output for every input, these semantics does not give a method to compute the only output for a given input. Moreover, these denotational semantics require the whole input to be available to the monitor. This restriction implies that a naive monitor would have to wait for the whole execution to be available. Even worse, in this section we consider unbounded executions as infinite traces.

The main activity in runtime verification is the study of how to generate monitors from formal specifications. In online monitoring, these monitors inspect the input as it is received producing the verdict incrementally. We develop now an iterative operational semantics for online monitoring of TeSSLa\textsubscript{a} specifications. To ease the presentation in this section we restrict TeSSLa\textsubscript{a} specifications to refer to present and past values only (even though the results can be easily extended to arbitrary TeSSLa\textsubscript{a} specifications). These specifications are known as efficiently monitorable \cite{10}, and satisfy that the values of an output stream variable \(x\) at a position \(t\) can be immediately resolved to their unique possible values (by evaluating \(E_x\) on the variables lower in the dependency graph) when inputs are known up to some time \(t'\) (typically \(t' = t\)).

We show in this section how TeSSLa\textsubscript{a} specifications can be compiled into a single monitor that receives multiple inputs from the system under observation.
Each input is received at an input source, which is associated with an input stream variable from the TeSSLa specification. Recall that at runtime the monitor can receive events at each input source at different speeds and with different delays (even though all events or signal changes are stamped with a time value from a global clock). However, at each source, the input events received will be in order of increasing time-stamp and the output and there are only a finite number of events between any two time instants.

3.1 Evaluation Engines

We introduce now a model of computation, called evaluation engines, to define the operational semantics of TeSSLa specifications and to reason about the correctness of different implementations of the online monitors.

Given a specification $\varphi(I, O)$ every defined stream variable is translated into a building block called an evaluation engine node or simply a node. Nodes communicate using event queues, including the input queues associated with input sources. We describe now evaluation engines and later in Section 3.3 describe the specific nodes that correspond to each TeSSLa construct.

Let $\varphi(I, O)$ be a TeSSLa specification (which, without loss of generality we assume to be flat). Let $G$ be the dependency graph of $\varphi$, and $N$ the collection of vertices of $G$. The evaluation engine of $\varphi$ contains one execution node per vertex in $N$. These nodes can execute concurrently and asynchronously. Nodes that read input sources are called input nodes. Nodes communicate using timed letters $(a, t)$, which we call events. Evaluation engines equip each node $n$ with one queue for each of the node’s inputs, that is, there is one queue per edge in the graph $G$. The node $n$ will only inspect and extract events from the heads of its input queues, while nodes that $n$ directly depends on generate events and insert these events through the tail of the corresponding queue. That is, at runtime, events are sent along the reversed edges of $G$. For example, in the specification in Fig. 1, there are two incoming edges to node $e$, one from node $eventCount$ and another edge from node $filter$. Every event received at input source $e$ will be copied into the input queue of node $eventCount$ and into the input queue of node $filter$. Then, the $eventCount$ node and the $filter$ node will process their copy of the event independently.

Queues support the standard operations for extracting the head, and appending to the tail.

**Definition 4 (Queue)** A queue $Q$ can be accessed with the following functions:

- $enqueue(Q, a)$: adds $a$ to the tail of $Q$,
- $dequeue(Q)$: removes the head 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 $Q$.

The only non-standard operation is $last(Q)$ which simply allows to remember the last element extracted from the queue. Our queues are typed in the
sense that each queue stores events \((a, t)\) where \(a\) has sort \(D\) (the sort of the corresponding stream variable or expression).

Formally, we model an evaluation engine as 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 input queue of every node. 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 at the input sources. During execution the evaluation engine can produce output events emitted at the queues that correspond to output stream variables. A transition \(\tau \in T\) of an evaluation engine involves the execution of exactly one node. A transition is called enabled when there is at least one event present in every input queue of the node, or if the corresponding node is an input source and there are input events received. Firing a transition corresponds to executing one step of the (small step) operational semantics of the TeSSLa operation associated with the execution node. Firing a transition 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. In particular, if \(t\) is the oldest timestamp among the events in the heads of the input queues, firing a node will consume all heads of all queues that have timestamp \(t\). The events produced are pushed to the corresponding queues of the nodes directly depending on the executing node. For convenience, we add the special transition \(\lambda \in T\) for the empty transition where no event is consumed, which is always enabled. We use \(\text{apply} : S \times T \rightarrow S\) for the application of a transition to a state of the evaluation engine. It is important to remark that firing a transition only removes events from the input queues, only places events in the output queues and preserves the internal states of all nodes except possibly the firing node. The function \(\text{node} : T \setminus \{\lambda\} \rightarrow N\) provides the node corresponding to a transition. A run is obtained by the repeated application of transitions.

**Definition 5 (Run)** A run of an evaluation engine \(E\) is a sequence \(r = (\lambda, s_0)(\tau_1, s_1)(\tau_2, s_2) \ldots \in (T \times S)^\omega\) of transitions and states such that for every \(i > 1\), \(\text{node}(\tau_i)\) is enabled at state \(s_{i-1}\) and \(\text{apply} (s_{i-1}, \tau_i) = s_i\).

We consider here finite and infinite inputs and outputs. Note that every finite run prefix can be extended to an infinite run by adding \(\lambda\) transitions at the end if necessary. A finite stream of events on source \(x\) will always contain a final event \((x, \text{prog}, \infty)\) (see below) to indicate that the stream corresponding to \(x\) contains no further event on any future time-stamp. We call these terminating streams and they are modeled by finite strings of events. However, we also model infinite words in which case the stream is modeled by an \(\omega\)-word whose time-stamps grow beyond any bound.

We will reason about the output of a run for a given engine and input. Let \(V = I \cup O\) be the finite collection of stream variables in a specification \(\varphi(I, O)\), and let \(D_v\) be the sort of variable \(v\). A timed letter is an element \((v, d, t)\) where \(v \in V\) is a stream variable, \(d \in D_v\) is a value, and \(t\) is a time-stamp. We use \(\Sigma_v\) for the alphabet of timed letters \((v, d, t)\) for stream variable \(v\), \(\Sigma_{in} = \bigcup_{v \in I} \Sigma_v\) for the input alphabet and \(\Sigma_{in} = \bigcup_{v \in O} \Sigma_v\) for the output alphabet. We use
\( \Sigma_{\text{in}}^\ast \) to denote finite strings from \( \Sigma_{\text{in}} \) (in increasing order of time-stamps), \( \Sigma_{\text{in}}^\omega \) for \( \omega \) strings of events from \( \Sigma_{\text{in}} \) (in increasing order and with a finite number of events between any two bounds) and \( \Sigma_{\text{in}}^\infty \) for the union of \( \Sigma_{\text{in}}^\ast \) and \( \Sigma_{\text{in}}^\omega \). The definitions of \( \Sigma_{\text{out}}^\ast \), \( \Sigma_{\text{out}}^\omega \) and \( \Sigma_{\text{out}}^\infty \) are analogous.

Given a timed letter \( a \) we use \( t(a) \) to denote its time component, we use \( \text{source}(a) \) to represent the source of \( a \) and \( \text{value}(a) \) for the value. Given a timed word \( w \) we use \( L(w) \) for the timed letters occurring in \( w \) and \( \text{pos}(w,a) \) for the position of a letter \( a \) in \( w \). The time-stamps of letters model the system time (the time in terms of the global clock of the system at which the event was stamped), while the position of a letter in the word models the monitoring time (the moment at which the monitor produced or received the event, relative to the processing instant of other events). Given a run \( r \) we use \( \text{output}(r) \) for the concatenation of the outputs produced in \( r \), and \( \text{output}(r,x) \) for the output produced at the queue corresponding to stream variable \( x \in O \). The notion of output a run can also be applied to a run prefix. We will later show that every possible execution of an evaluation engine generates an equivalent output if each input stream is the same even if the streams are non-synchronized, as long as these streams are in-order.

**Definition 6 (In-order & Synchronized Inputs)** A word \( w \) is

- **in-order** whenever for every \( a, b \in L(w) \) if \( \text{pos}(w,a) < \text{pos}(w,b) \) and \( \text{source}(a) = \text{source}(b) \) then \( t(a) \leq t(b) \).
- **synchronized** whenever for every \( a, b \in L(w) \) if \( \text{pos}(w,a) < \text{pos}(w,b) \) then \( t(a) \leq t(b) \).

**Example 5** Consider two input sources \( x \) and \( y \), and let \( x \) receive the input \((x, T, 0)(x, T, 3)\) and \( y \) receive \((y, F, 1)(y, F, 6)\). The following two inputs

- \( w_1 : (x, T, 0)(y, F, 1)(x, T, 3)(y, F, 6) \) and
- \( w_2 : (x, T, 0)(y, F, 1)(y, F, 6)(x, T, 3) \)

are in-order. However, \( w_1 \) is synchronized but \( w_2 \) is not, because in \( w_2 (x, T, 3) \) is received after \((y, F, 6)\) but \( 3 \) is an earlier time-stamp than \( 6 \). The input \( w_2 \) is still in-order because the sources are different for the letters received in reverse order of time-stamps.

Note that 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. We assume that all input streams are received in increasing time order and that all nodes produce events in increasing time order if their inputs are received in increasing time order. Even though output events produced by any given node are produced in increasing time according to a time-stamp, outputs produced by different nodes may not be ordered among each other, so concatenating output streams does not necessarily lead to a timed order sequence (that is, our streams are in-ordered but not necessarily synchronized).
Output Completeness and Progress

An actual event carries information about its occurrence, but we also need to be able to convey information about the absence of events up to a given time. Consider for example a filter node that never generates an event because its input always gets filtered out, followed by an eventCount node. The eventCount node would never generate an output because it will never receive an event from the filter node. Consequently, the verdict that the output of the eventCount node models would not be known, because the eventCount block does not know whether there are no events or there are events but these have not arrived. We introduce now extra events, called progress events, whose only purpose is to inform nodes downstream about the absence of events up to the time-stamp of the progress event. In particular, an event \((a, \text{prog}, \infty)\) corresponds to the information that the stream that \(a\) models has no events in the future (modeling a terminating stream).

**Definition 7 (Output complete)** 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, which can either be a real event or a progress event.

In our example, if the node implementing the filter is output complete it will inform the eventCount node about the absence of actual events (due to the filtering) by sending a progress event. In turn, the eventCount node would not increase the counter but emit the verdict with the same value, or a progress event to indicate that there is no change in the total number of events counted. As we will see, in order to guarantee progress of all nodes beyond any time bound, we require the operational semantics of all TeSSLa operators to be output complete, while still implement the intended functionality.

Nodes realizing state-less functions on signals with multiple inputs (e.g., addition) always wait until they know the values of all their input signals in order to produce an output. If such a node \(n\) receives a progress event, instead of an event carrying the change of a value, \(n\) knows that the signal has not changed up to the time-stamp attached to the progress event. With that knowledge, \(n\) can produce the output for the time instant of the change in the other inputs (or generate itself a progress event if no input has changed its value). Recall that nodes are enabled whenever all of their input queues contain at least one event (real or progress). For state-full functions the change required to process progress events depends on the particular function. It is easy to see that with output complete building blocks all nodes, when fired, consume at least one input and generate exactly one output.

**Definition 8 (Progressing node)** We say that a node \(n\) is **progressing** whenever the output of \(n\) progresses beyond any time \(t\), provided that all its inputs are eventually available beyond any time \(t'\) (and the node is fired enough times).

Note that if a node \(n\) is enabled (all its inputs have at least an event), then it will be continuously enabled until fired. This is because all other transitions
can only add events to the input queues of \( n \). Hence, the only requirement to guarantee that every node eventually generates output beyond any time bound is that the scheduler (eventually) fires all enabled nodes, a property that is usually known as fairness of the scheduler. In particular, a run is fair if every enabled transition is eventually taken. Using output complete progressing nodes, in fair runs all events in all queues are eventually processed, and all queues eventually progress beyond any bound.

**Theorem 1** Let \( E \) be an evaluation engine and let all its nodes be progressing and output complete. Then, in every fair run of \( E \) all outputs for all queues eventually progress beyond any bound.

**Proof** By contradiction, assume that there is a run of \( E \) for which some node does not progress beyond some bound \( t \). Let \( n \) be one such node that is minimal in a reverse topological order of \( G \) (that is, all nodes upstream between input source nodes and \( n \) progress beyond any bound in the run). It follows, by the progress of all nodes of \( E \), that all queues up-stream from \( n \) progress beyond any bound, in particular the nodes directly connected to \( n \). Since all nodes are complete, this means that the input queues of \( n \) contain events beyond any bound, and therefore \( n \) is continuously enabled. If \( n \) is not taken then the run is not fair, which is a contradiction. But since \( n \) is progressing, if \( n \) is taken enough times it then must generate output beyond any bound, and in particular beyond \( t \), which contradicts our assumption. Therefore, in every fair execution all nodes generate output beyond any bound, as desired. \( \square \)

We say that a word \( v \) is complete up-to \( t \) if the last event in \( v \) has a time-stamp \( t' \) with \( t' \geq t \). Given a word \( v \) and a time-stamp \( t \), we use \( v|_t \) for the word that results by eliminating all progress events and all events with a time-stamp higher than \( t \).

Consider two finite streams \( v \) and \( w \) for the same source \( x \). We say that \( v \) is complete up-to \( t \) if the last event in \( v \) has a time-stamp \( t' \) with \( t' \geq t \). We say that \( v \) and \( w \) coincide up-to \( t \) if \( v|_t = w|_t \).

### 3.2 Asynchronous Correctness

We introduce now a theory of time transducers to prove that evaluation engines always compute an equivalent answer, in spite of the (fair) scheduler used and the relative arrival times of the input events.

A “classical” synchronous transducer is simply an element of \((\Sigma_{in} \times \Sigma_{out})^*\). However, we model asynchronous transducers to decouple the rate of arrival at input sources from the internal execution of the evaluation engine. A timed transducer \( F \) is \( F \subset \Sigma_{in}^\infty \times \Sigma_{out}^\infty \). Our timed transducers will relate every input to some output (possibly \( \varepsilon \)). A timed transducer \( F \) is complete if for all \( w \in \Sigma_{in}^\infty \) there is some \( v \in \Sigma_{out}^\infty \) such that \((w, v) \in F \). Our intention is to use \( F \) as the set of outputs that executions of an engine, such that by reading \( w \), the engine that \( F \) models can produce \( v \).
Definition 9 A timed transducer $F$ is strictly deterministic if for all $w \in \Sigma^\infty_{\text{in}}$, and for all $v, v' \in \Sigma^\infty_{\text{out}}$, if $(w, v) \in F$ and $(w, v') \in F$, then $v = v'$.

In the theory of transducers, a strictly deterministic transducer is sometimes called a functional transducer. For example, consider a transducer that delays every input letter by one time instant. This transducer is complete and strictly deterministic and would translate $(a, F, 0)(b, F, 1)(a, F, 2)$ into $(a, F, 1)(b, F, 2)(a, F, 3)$. However, strict determinism is too fine grained for our purposes because we want to allow output letters to be produced out of order, that is, we want to allow the monitor to produce earlier a verdict with a later timestamp (for a different output stream). We use the timed reordering function $\text{timed} : \Sigma^\infty_{\text{out}} \to \Sigma^\infty_{\text{out}}$ which removes progress events and reorders a word according to the time-stamps of its time letters (and break ties according to some lexicographic order in the source). The following notion of asynchronous determinism captures more precisely the deterministic nature of asynchronous evaluation engines.

Definition 10 (Asynchronous determinism) A timed transducer $F$ is called asynchronous deterministic if for all $w \in \Sigma^\infty_{\text{in}}$ and for all $v, v' \in \Sigma^\infty_{\text{out}}$ with $(w, v) \in F$ and $(w, v') \in F$, $\text{timed}(v) = \text{timed}(v')$.

Asynchronous determinism allows non-deterministic transducers to produce different outputs for the same input prefix as long as the outputs are identical up-to reordering.

Finally, we introduce observational equivalence between transducers. We will show that the transducers corresponding to different (fair) schedulers of the same evaluation engine are observationally equivalent, which allows to reason about runs using deterministic schedulers but use highly concurrent schedulers at runtime.

Definition 11 (Observational Equivalence) Let $F$ and $G$ be two timed transducers over the same input and output alphabets, and let $w \in \Sigma^\infty_{\text{in}}$. We say that $F$ and $G$ are observational equivalent, and we write $F \equiv_O G$ whenever for all $v, u \in \Sigma^\infty_{\text{out}}$ with $(w, v) \in F$ and $(w, u) \in G$, $\text{timed}(v) = \text{timed}(u)$.

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 $F \equiv_O F$.

Before we prove the main result of evaluation engines as asynchronous transducers, we show two auxiliary lemmas.

Lemma 1 Let $r$ and $r'$ be two arbitrary runs of $E$ on the same input. The output generated at every node $x$ is the same in $r$ and in $r'$, that is output$(r, x) = \text{output}(r', x)$.

Proof By contradiction, assume that the outputs are different and let $n$ be the lowest node in a reverse topological order in $G$ whose output differ. Then, in $r$ and $r'$, even though the input queues to $n$ are identical, $n$ generates different outputs, which is a contradiction because the individual transitions, including $n$ of engines are deterministic. $\square$
Lemma 2 Let \( r \) and \( r' \) be two runs of \( E \) such that \( \text{output}(r, x) = \text{output}(r, x') \) for every \( x \). Then \( \text{timed}(\text{output}(r)) = \text{timed}(\text{output}(r')) \).

Proof By contradiction, assume \( \text{timed}(\text{output}(r)) \neq \text{timed}(\text{output}(r')) \) and let \((x, a, t)\) be the first event after the common prefix of \( r \) and \( r' \). That means that the projection on \( x \) of the common prefix of \( r \) and \( r' \) before \((x, a, t)\) is equal, and \((x, a, t)\) is the next event in \( \text{output}(r, x) \), but this event is not in \( \text{output}(r', x) \), which contradicts Lemma 1. \( \square \)

Lemma 2 implies directly the following result.

Theorem 2 Let \( E \) be an evaluation engine. Then
- the transducer for any set of fair runs of \( E \) is asynchronous deterministic
  and
- the transducers of \( E \) for two fair schedulers are observationally equivalent.

3.3 TeSSLa Library of Builtin Functions. Operational Semantics

We now present the operational semantics of the TeSSLa_{sa} functions listed in Section 2. All the nodes presented—when fired—only modify their internal state, always process the oldest input event, and always generate at least one output event (that is, all nodes are output complete). The fact that the oldest event is always processed implies that all input events are processed beyond any bound in a finite number of firings, because the inputs are non-zeno. Moreover, we will show that output events generated by the nodes grow beyond any bound in a finite number of firings (that is, all nodes are progressing). Hence, all TeSSLa_{sa} specifications satisfy the conditions of Theorem 1 and all runs under a fair scheduler progress beyond any bound.

The operational semantics defined here correspond with the denotational semantics from Section 2.4. In order to see this, consider a node \( n \) that has processed all inputs up to \( t \) and generated the output up-to \( t' \). It is sufficient to show that for every valuation of the input streams that extends the input processed, the only output valuation corresponds with the output generated up to \( t' \). If this holds, after a finite number of firings all output streams correspond with the only possible outputs according to the denotational semantics of TeSSLa_{sa}. Different schedulers can produce different runs for the same inputs (which can also arrive at different times), but after enough time every output stream will be the same.

The computational nodes that we describe now are enabled if for all input queues \( Q \) we have \( \text{peek}(Q) \neq \text{nil} \). We use the statement \( \text{emit}(a) \) to indicate that \( a \) is send to the output, that is \( a \) is enqueued to the corresponding input queues of all nodes directly connected to the output of the running node in the dependency graph.
3.3.1 Binary Arithmetic Functions

We start with \(\text{add} : S_D \times S_D \to S_D\) that has two input queues \(A\) and \(B\) and the following code:

```plaintext
prog ← false
if peek(A).time = peek(B).time then
    a ← peek(A).value, b ← peek(B).value,
    t ← peek(A).time
    if a = progress and b = progress then
        prog ← true
    else if a = progress then a ← last(A).value
    else if b = progress then b ← last(B).value
dequeue(A),
dequeue(B)
else if peek(A).time < peek(B).time then
    a ← peek(A).value, b ← last(B).value,
    t ← peek(A).time
    if a = progress then prog = true
dequeue(A)
else
    a ← last(A).value, b ← peek(B).value,
    t ← peek(B).time
    if b = progress then prog ← true
dequeue(B)
if prog then
    emit(progress, time = t)
else
    emit(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 \(\text{peek}(Q).value = \text{progress}\) we have the following implicit behavior for a call of \(\text{dequeue}(Q)\):

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

Furthermore, for signals the command \(\text{emit}\) 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, unless the domain is \text{unit}, \(\text{emit}\) 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: \(\text{sub}, \text{mul}, \text{div}, \text{gt}, \text{geq}, \text{eq}, \text{max}, \text{min}, \text{and}\) and \(\text{or}\). This is simply achieved by replacing line 24 as follows.
– For sub:
  24 emit(value = a − b, time = t)

– For mul:
  24 emit(value = a · b, time = t)

– For div:
  24 emit(value = a/b, time = t)

– For gt:
  24 emit(value = a > b, time = t)

– For geq:
  24 emit(value = a ≥ b, time = t)

– For leq:
  24 emit(value = a ≤ b, time = t)

– For eq:
  24 emit(value = a = b, time = t)

– For max:
  24 emit(value = max(a, b), time = t)

– For min:
  24 emit(value = min(a, b), time = t)

3.3.2 Unary Arithmetic Functions

For abs : S_D → S_D we have one input queue A and the following code:

```java
if peek(A).value = progress then
    emit(progress, time = peek(A).time)
else
    emit(value = |peek(A).value|, time = peek(A).time)
dequeue(A)
```
By changing the applied arithmetics, the code above can be applied to the following unary operators as well: \( \text{abs} : \mathcal{E}_D \rightarrow \mathcal{E}_D, \text{not} : \mathcal{S}_B \rightarrow \mathcal{S}_B, \text{neg} : \mathcal{E}_B \rightarrow \mathcal{E}_B, \text{timestamps} : \mathcal{E}_D \rightarrow \mathcal{E}_T, \text{mrv} : \mathcal{E}_D \times \mathcal{D} \rightarrow \mathcal{S}_D \) and \( \text{changeOf} : \mathcal{S}_D \rightarrow \mathcal{E}_{(\top)} \). This is accomplished by changing line 4 in the code of \( \text{abs} \) with the corresponding operation.

### 3.3.3 Aggregation Functions

For \( \text{sum} : \mathcal{E}_D \rightarrow \mathcal{S}_D \) we have one input queue \( A \), the internal state \( \in \mathcal{D} \) initialized with 0 and the following code:

```plaintext
1  if peek(A).value = progress then
2     emit(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: \( \text{maximum} : \mathcal{E}_D \times \mathcal{D} \rightarrow \mathcal{S}_D, \text{maximum} : \mathcal{S}_D \rightarrow \mathcal{S}_D, \text{minimum} : \mathcal{E}_D \times \mathcal{D} \rightarrow \mathcal{S}_D, \text{minimum} : \mathcal{S}_D \rightarrow \mathcal{S}_D \) and \( \text{sma} : \mathcal{E}_D \times \mathcal{D} \). This is accomplished by replacing line 4 with the right operation.

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

```plaintext
1  prog ← false
2  if peek(A).time = peek(B).time then
3     t ← peek(B).time
4     if peek(A).value = progress and peek(B).value = progress then
5        prog ← true
6     else if peek(B).value = progress then
7        state ← state + 1
8  else
9     state ← 0
10    dequeue(A)
11    dequeue(B)
12  else if peek(A).time < peek(B).time then
13     t ← peek(A).time
14     if peek(A).value = progress then
15        prog ← true
16     else
17        state ← state + 1
18        dequeue(A)
19  else
20     t ← peek(B).time
21     if peek(B).value = progress then
22        prog ← true
```
3.3.4 Filtering Functions

For \( \text{ifThen} : \mathcal{E}_{D_1} \times \mathcal{S}_{D_2} \rightarrow \mathcal{E}_{D_2} \) we have two input queues \( A \) and \( B \) and the following code:

```plaintext
1. prog ← false
2. if peek(A).time = peek(B).time then
3.    t ← peek(A).time
4.    if peek(A).value = progress then
5.      prog ← true
6.    else if peek(B).value = progress then
7.      v ← last(B).value
8.    else
9.      v ← peek(B).value
10.   dequeue(A), dequeue(B)
11. else if peek(A).time < peek(B).time then
12.    t ← peek(A).time
13.    if peek(A).value = progress then
14.      prog ← true
15.    else
16.      v ← last(B).value
17.    dequeue(A)
18. else
19.    t ← peek(B).time
20.   prog ← true
21.   dequeue(B)
22. if prog then
23.   emit(value = progress, t)
24. else
25.   emit(value = v, t)
```

For \( \text{filter} : \mathcal{E}_D \times \mathcal{S}_B \rightarrow \mathcal{E}_D \) we have two input queues \( A \) and \( B \) and the following code:

```plaintext
1. prog ← true
2. if peek(A).time = peek(B).time then
3.    t ← peek(A).time
4.    if (peek(B).value = progress and last(B).value) or peek(B).value then
5.      prog ← true
6.    else
7.      v ← last(B).value
8.    dequeue(B)
9. if prog then
10.   emit(value = progress, t)
11. else
12.   emit(value = v, t)
```
5     \texttt{v} \leftarrow \texttt{peek(A).value}
6     \texttt{else}
7         \texttt{prog} \leftarrow \texttt{true}
8         \texttt{dequeue(A), dequeue(B)}
9     \texttt{else if} \texttt{peek(A).time < peek(B).time then}
10        \texttt{t} \leftarrow \texttt{peek(A).time}
11        \texttt{if last(B).value} \texttt{then}
12           \texttt{v} \leftarrow \texttt{peek(A).value}
13        \texttt{else}
14           \texttt{prog} \leftarrow \texttt{true}
15           \texttt{dequeue(A)}
16     \texttt{else}
17         \texttt{prog} \leftarrow \texttt{true}
18         \texttt{if prog then}
19            \texttt{emit(value = progress, time = t)}
20        \texttt{else}
21            \texttt{emit(value = v, time = t)}

For \texttt{ifThenElse} : \mathcal{S}_A \times \mathcal{S}_D \times \mathcal{S}_D \rightarrow \mathcal{S}_D we have three input queues A, B and C, but apart from more cases the operative semantics is very similar to \texttt{ifThen} and \texttt{filter} above. For \texttt{merge} : \mathcal{E}_D \times \mathcal{E}_D \rightarrow \mathcal{E}_D we have two input queues A and B and the following code:

1     \texttt{prog} \leftarrow \texttt{false}
2     \texttt{if peek(A).time = peek(B).time then}
3        \texttt{t} \leftarrow \texttt{peek(A).time}
4        \texttt{if peek(A).value = progress} \texttt{then}
5           \texttt{v} \leftarrow \texttt{peek(A).value}
6     \texttt{else}
7        \texttt{v} \leftarrow \texttt{peek(B).value}
8        \texttt{dequeue(A), dequeue(B)}
9     \texttt{else if} \texttt{peek(A).time < peek(B).time then}
10        \texttt{t} \leftarrow \texttt{peek(A).time}
11        \texttt{v} \leftarrow \texttt{peek(A).value}
12        \texttt{dequeue(A)}
13     \texttt{else}
14        \texttt{t} \leftarrow \texttt{peek(B).time}
15        \texttt{v} \leftarrow \texttt{peek(B).value}
16        \texttt{dequeue(B)}
17     \texttt{emit(value = v, time = t)}

With slight modifications the code above can be applied to \texttt{occursAny} : \mathcal{E}_D_1 \times \mathcal{E}_D_2 \rightarrow \mathcal{E}_\{\top\} and \texttt{occursAll} : \mathcal{E}_D_1 \times \mathcal{E}_D_2 \rightarrow \mathcal{E}_\{\top\} as well.
### 3.3.5 Timing Functions

For delay: \( S_D \times T \times D \to S_D \) we have one input queue \( A \) and a constant \( d \in T \) and the following code:

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

Taking into account the additional default value this can be extended to delay: \( E_D \times T \to E_D \) on signals, too. For shift: \( E_D \to E_D \) we only have one input queue \( A \) and the following code:

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

For within: \( T \times T \times E_D \to S_B \) we have one input queue \( X \) belonging to a stream \( x \in E_D \) and the two constants \( a, b \in T \), such that the node iteratively computes within\((a, b, x)\). For the operative semantics we assume \( a < b \leq 0 \). We then have the following code:

```plaintext
1 if last(X).time − a < peek(X).time − b then
2   emit(value = false, time = last(X).time − a
3   emit(value = true, time = peek(X).time − b)
4 else
5   emit(value = progress, time = peek(X).time − b)
6   dequeue(X)
```

All the constructs are deterministic and have no loops; all constructs consume at least one event from at least one input queue (all the oldest events), and produce one event. Moreover, the inputs are consumed in increasing time-stamps and the outputs are also generated in increasing time-stamps. The operational semantics for all functions except the timing functions only use time-stamps that already occur in the input for their outputs.

### 4 Implementation and Evaluation

We report here an empirical evaluation of an implementation of the TeSSLa\(_a\) evaluation engine\(^3\). Our implementation consists of two parts. First, a compiler translates a TeSSLa\(_a\) 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 is built on top of Erlang, the Erlang virtual machine BEAM and the Erlang runtime library, called The Open Telecom Platform (OTP). One of the key concepts of Erlang/OTP is the usage of the actor model\(^23\) to deploy

\(^3\) Tools and benchmarks available at [https://www.tessla.io/acyclic](https://www.tessla.io/acyclic)
code over multiple cores or even distributed machines in a network. An actor is basically a self contained entity, that holds a state and can receive and send messages to other actors. Since an actor manages its own state and is the only one that can manipulate it, an actor can be scheduled on any core as long as the runtime guarantees transparent message delivery. Our implementation realizes the computation nodes as actors and relies on Erlang/OTP for the scheduling of those. Theorem 2 guarantees that all runs of an engine are observationally equivalent, independently of the scheduler. Hence this implementation is correct independently of the concrete realization of the Erlang/OTP scheduler.

4.1 Evaluation

With the empirical evaluation we pursue to answer the following questions:

(A) *How well does the implementation exploit parallelism?* The evaluation engine discussed in this paper uses the minimal amount of synchronization which guarantees a correct output. Hence, our implementation should be able to automatically utilize the parallel computational power of multiple cores.

(B) *How is the runtime influenced by the length of the input trace?* The evaluation engine exchanges messages along the reversed dependency graph, which is acyclic. If the runtime of the computation nodes itself is constant per processed message, every additional input event should add a constant delta to the overall runtime.

(C) *What is the relation between the specification size and the runtime?* Adding one extra computation node to the specification should add a constant delta to the runtime, because every event is now processed by one extra computational node.

To investigate these questions and 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. This evaluation only considers computation nodes, which store at most the previous input. Hence the assumption of constant computation time per node is justified. Furthermore, we do not filter any events, and input values in the input events does not influence how the events are processed. Otherwise adding more input events would not always have the same effect and the measurements would be biased.

Under these premises we created two scenarios: the chain scenario and the tree scenario. The chain scenario consists of a chain of abs functions. Every computation node directly depends on the output of the previous node. This is a worst case scenario in the sense that there are no two independent paths in the dependency graph, and each block sequentially depends on the previous node. The tree scenario goes one step further by first reusing the same input event multiple times repeatedly and then recombining all those by joining two inputs each in a logarithmic binary tree. This scenario has a
maximal synchronization requirement, since the output node—the root of the tree—depends on all the intermediate computation nodes. Topologically, real TeSSLa specifications are typically a variation and combination of these two scenarios.

All benchmarks were performed on the same machine with up to 12 cores and 32GB of RAM. The results displayed in Figure 6 and Tables 1–3 summarize the average of 20 runs. In addition to the overall runtime, the tables show the number of outliers ignored in the average, the error—i.e. the maximal deviation of the individual executions from the average—and the relative runtime per event or node, respectively.

(A) Number of cores. For this benchmark we used the chain specification with 16 nodes and the tree specification with 48 nodes. The input trace contains exactly 10,000 events. The results in Table 1 and the first row of Figure 6 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 in the case of the completely linear dependency graph. The answer to question (A) is that our asynchronous implementation is able to exploit parallelism automatically in a pipeline fashion.

(B) Number of events in the input. To study the dependency of the execution time with the input length we reused the same specifications and modified the input sizes. The results are shown in Table 2 and the second row of Figure 6. The plot on the right shows the average time per event. With an increase on the input length, the static overhead is quickly amortized and the length of the trace becomes less relevant for the average event time. Consequently, in both cases the relative time shows a decay towards a constant as more events are added.

(C) Number of nodes in the specification. The execution time also grows more or less linearly in the size of the specification for both specifications. For this benchmark we used input traces of 1,000 events and increased the number of computation nodes in both specifications. The results are shown in Table 3 and the bottom row of Figure 6. Again, the relative time per event shows a decay as more nodes are added, because the static overhead becomes less relevant. Even with the number of computation nodes now being exactly the same in the chain and the tree specification the runtime for the tree specification is slightly higher. The tree specification has many more edges between the computation nodes than the chain specification and hence the message passing overhead is higher in this benchmark.

Even though these measurements are not fully cover all arbitrary TeSSLa specifications, they have been chosen to represent extreme cases of the dependencies between blocks. Our empirical evaluation supports the feasibility of our distributed asynchronous monitoring approach in terms of efficiency.
Table 1 Benchmarking the chain specification with 16 nodes (left) and the tree specification with 48 nodes (right), both with 10,000 input events and a varying number of cores. Average runtime of 20 executions without the outliers, error being the maximal deviation of the individual executions from the average.

<table>
<thead>
<tr>
<th>#cores</th>
<th>time [s]</th>
<th>outliers</th>
<th>error [ms]</th>
<th>time/##</th>
<th>#cores</th>
<th>time [s]</th>
<th>outliers</th>
<th>error [ms]</th>
<th>time/##</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.09650</td>
<td>0</td>
<td>3.6</td>
<td>1</td>
<td>3.44400</td>
<td>0</td>
<td>13.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.84310</td>
<td>1</td>
<td>5.8</td>
<td>2</td>
<td>1.96590</td>
<td>0</td>
<td>7.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.58390</td>
<td>1</td>
<td>2.5</td>
<td>4</td>
<td>1.31000</td>
<td>1</td>
<td>11.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.62420</td>
<td>1</td>
<td>2.2</td>
<td>8</td>
<td>1.20700</td>
<td>0</td>
<td>12.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>0.63550</td>
<td>0</td>
<td>3.3</td>
<td>12</td>
<td>1.38080</td>
<td>3</td>
<td>5.6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2 Benchmarking the chain specification with 16 nodes (left) and the tree specification with 48 nodes (right), both with 12 cores and a varying number of input events. Average runtime of 20 executions without the outliers, error being the maximal deviation of the individual executions from the average.

<table>
<thead>
<tr>
<th>#events</th>
<th>time [s]</th>
<th>outliers</th>
<th>error [ms]</th>
<th>time/##</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>0.19599</td>
<td>0</td>
<td>0.39198</td>
<td>500</td>
</tr>
<tr>
<td>1000</td>
<td>0.21397</td>
<td>1</td>
<td>0.39198</td>
<td>1000</td>
</tr>
<tr>
<td>1500</td>
<td>0.23301</td>
<td>0</td>
<td>0.39198</td>
<td>1500</td>
</tr>
<tr>
<td>2000</td>
<td>0.25167</td>
<td>0</td>
<td>0.39198</td>
<td>2000</td>
</tr>
<tr>
<td>2500</td>
<td>0.27487</td>
<td>0</td>
<td>0.39198</td>
<td>2500</td>
</tr>
<tr>
<td>3000</td>
<td>0.30048</td>
<td>0</td>
<td>0.39198</td>
<td>3000</td>
</tr>
<tr>
<td>3500</td>
<td>0.32120</td>
<td>0</td>
<td>0.39198</td>
<td>3500</td>
</tr>
<tr>
<td>4000</td>
<td>0.34940</td>
<td>1</td>
<td>0.39198</td>
<td>4000</td>
</tr>
<tr>
<td>4500</td>
<td>0.37226</td>
<td>1</td>
<td>0.39198</td>
<td>4500</td>
</tr>
<tr>
<td>5000</td>
<td>0.39520</td>
<td>0</td>
<td>0.39198</td>
<td>5000</td>
</tr>
<tr>
<td>5500</td>
<td>0.41887</td>
<td>0</td>
<td>0.39198</td>
<td>5500</td>
</tr>
<tr>
<td>6000</td>
<td>0.44130</td>
<td>2</td>
<td>0.39198</td>
<td>6000</td>
</tr>
<tr>
<td>6500</td>
<td>0.46300</td>
<td>1</td>
<td>0.39198</td>
<td>6500</td>
</tr>
<tr>
<td>7000</td>
<td>0.48505</td>
<td>1</td>
<td>0.39198</td>
<td>7000</td>
</tr>
<tr>
<td>7500</td>
<td>0.50200</td>
<td>0</td>
<td>0.39198</td>
<td>7500</td>
</tr>
<tr>
<td>8000</td>
<td>0.52940</td>
<td>0</td>
<td>0.39198</td>
<td>8000</td>
</tr>
<tr>
<td>8500</td>
<td>0.55640</td>
<td>0</td>
<td>0.39198</td>
<td>8500</td>
</tr>
<tr>
<td>9000</td>
<td>0.58330</td>
<td>0</td>
<td>0.39198</td>
<td>9000</td>
</tr>
<tr>
<td>9500</td>
<td>0.61440</td>
<td>2</td>
<td>0.39198</td>
<td>9500</td>
</tr>
<tr>
<td>10000</td>
<td>0.64220</td>
<td>0</td>
<td>0.39198</td>
<td>10000</td>
</tr>
</tbody>
</table>

Table 3 Benchmarking the chain specification (left) and the tree specification (right), both with 10,000 input events, 12 cores and a varying number of nodes in the specification. Average runtime of 20 executions without the outliers, error being the maximal deviation of the individual executions from the average.

<table>
<thead>
<tr>
<th>#nodes</th>
<th>time [s]</th>
<th>outliers</th>
<th>error [ms]</th>
<th>time/##</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.42120</td>
<td>0</td>
<td>52.65000</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>0.69330</td>
<td>0</td>
<td>38.08125</td>
<td>16</td>
</tr>
<tr>
<td>24</td>
<td>0.81260</td>
<td>0</td>
<td>53.86667</td>
<td>24</td>
</tr>
<tr>
<td>32</td>
<td>1.02110</td>
<td>2</td>
<td>31.99038</td>
<td>32</td>
</tr>
<tr>
<td>48</td>
<td>1.17420</td>
<td>0</td>
<td>24.46250</td>
<td>48</td>
</tr>
<tr>
<td>64</td>
<td>1.26470</td>
<td>1</td>
<td>19.76094</td>
<td>64</td>
</tr>
<tr>
<td>96</td>
<td>1.50900</td>
<td>0</td>
<td>15.71875</td>
<td>96</td>
</tr>
<tr>
<td>128</td>
<td>1.82700</td>
<td>27.0</td>
<td>14.27344</td>
<td>128</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#nodes</th>
<th>time [s]</th>
<th>outliers</th>
<th>error [ms]</th>
<th>time/##</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.42120</td>
<td>0</td>
<td>52.65000</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>0.69330</td>
<td>0</td>
<td>38.08125</td>
<td>16</td>
</tr>
<tr>
<td>24</td>
<td>0.81260</td>
<td>0</td>
<td>53.86667</td>
<td>24</td>
</tr>
<tr>
<td>32</td>
<td>1.02110</td>
<td>2</td>
<td>31.99038</td>
<td>32</td>
</tr>
<tr>
<td>48</td>
<td>1.17420</td>
<td>0</td>
<td>24.46250</td>
<td>48</td>
</tr>
<tr>
<td>64</td>
<td>1.26470</td>
<td>1</td>
<td>19.76094</td>
<td>64</td>
</tr>
<tr>
<td>96</td>
<td>1.50900</td>
<td>0</td>
<td>15.71875</td>
<td>96</td>
</tr>
<tr>
<td>128</td>
<td>1.82700</td>
<td>27.0</td>
<td>14.27344</td>
<td>128</td>
</tr>
</tbody>
</table>
Fig. 6 Benchmarks. The plots on the left show the total time and the plots on the right show the relative time per event or node, resp. The blue line with cross marks indicates the runtime of the chain specification and the red line with asterisk marks indicates the runtime of the tree specification.

4.2 Instrumentation & Tool Chain

As mentioned in the introduction, TeSSLa$_a$ is designed to simplify FPGA based implementations of the evaluation engine. Additionally, we also implemented auxiliary tools to use TeSSLa$_a$ 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$_a$ 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$_a$ functions are pre-processed 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\(a\) specification \(\varphi\):

1. Process \(\varphi\) 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\(a\) evaluation engine with \(G\) and \(r\) as inputs.

This tool chain using the TeSSLa\(a\) evaluation engine analyzed in this paper is available as an Atom plugin integrating all the steps described above. The plugin for the Atom editor is shown in Figure 7. Other TeSSLa implementations not restricted to the acyclic TeSSLa\(a\) fragment are discussed in [9] and are available on the TeSSLa website\(^4\).

5 Conclusion

We studied in this paper the problem of efficiently monitoring stream runtime verification specifications of real-time events and signals. In particular, our goal was to support the arrival of events at different speeds of input event streams and with different and varying latencies. The language we support

\(^4\) https://www.tessla.io
in our solution is TeSSL\small{a}, an acyclic fragment of the stream-based runtime verification language TeSSL\small{a} for non-synchronized streams of piece-wise constant real-time signals. We introduced the notion of evaluation engines as an abstraction of online monitors that can execute asynchronously, based on independent building blocks that communicate using message passing. The main result is that, assuming a few local properties in each of the concurrent nodes (output completeness and progress), all executions for a given input compute an equivalent output. The possible differences between the outputs are in terms of the relative order of the output produced, but not in terms of the streams they compute. A novel notion of asynchronous transducers allows to formalize these equivalences, both for terminating and unbounded executions.

We then defined the operational semantics of TeSSL\small{a} in terms of evaluation engines. To achieve output completeness we introduce the notion of progress events, that allows the explicit communication of the absence of events in a given stream. Our results enable 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. Finally, we report in this paper an implementation of an evaluation engine written in Elixir/Erlang.

Future work includes the extension of the approach to support richer specification languages, as well as supporting input streams that contain missing or approximate information or errors in their values or precise timings.

Acknowledgments

We thank Jannis Harder and Sebastian Hungerecker for their work on TeSSL\small{a} and its compiler.
References