Noname manuscript No.
(will be inserted by the editor)

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 instrumentation 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 [5,14,22]. Similar formalisms proposed are based on regular expressions [36], 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 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, 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$_a$ 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[1]. TREs have been combined with STL [34] 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$_a$ specifications. These mon-

[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.

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$_a$ specifications.

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

(4) The precise syntax and semantics of TeSSLa$_a$, an acyclic fragment of TeSSLa, 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 TeSSLa$_a$ 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 ω-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 TeSSLa$_a$

We describe in this section the real-time specification language TeSSLa$_a$[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 $\mathbb{T}$ 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 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 $\eta : \mathbb{T} \rightharpoonup D$ such that $E(\eta) := \{t \in \mathbb{T} \mid \eta(t) \text{ is defined}\}$ does not contain bounded infinite subsets.

[2] TeSSLa$_a$ is available at `https://www.tessla.io/acyclic`

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 only value in a 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.

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$_a$ and delivers a more convenient model to the user.

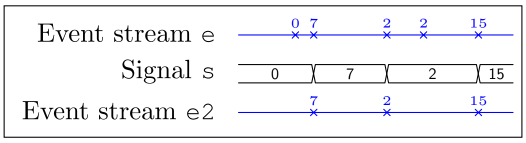
Definition 2 (Signal) A *signal* is a total function $\sigma : \mathbb{T} \to D$ such that the set of *change points*

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

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$_a$ event streams represent piece-wise constant signals with perfect accuracy.

Example 1 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 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.

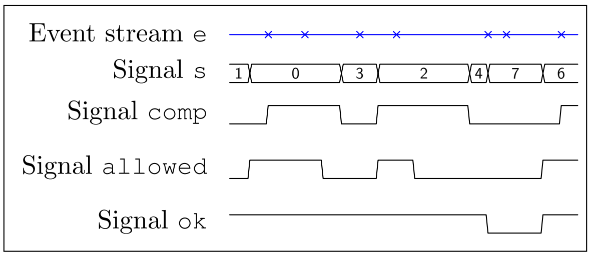
2.2 Syntax of TeSSLa$_a$

We begin with an example to illustrate a simple TeSSLa$_a$ specification. Specifications are declarative, 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 their values will be reported to the user.

Example 2 Consider the following TeSSLa$_a$ 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 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$_a$ specification *spec* is

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

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). 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$_a$ by adding builtin functions, user defined macros and timing functions.

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

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. Macros can be defined with the following production, which is added to *spec*:

$$macro := \texttt{\textbf{fun}}\ \ name\texttt{\textbf{(}}name(\texttt{\textbf{,}}\ \ name)^*\texttt{\textbf{)}}\ \ \texttt{\textbf{:=}}\ \ texpr \mid macro\ \ macro$$

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:

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

■

The expressivity of TeSSLa$_a$ is obtained by the use a set of *builtin functions*. We first define in Section 2.3 the semantics of the temporal core of TeSSLa$_a$, which is enough to define the semantics of TeSSLa$_a$ 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

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

and for event streams

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

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 ouside 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_η 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:

$$\begin{aligned} \mathit{shift} &: \mathcal{E}_D \rightarrow \mathcal{E}_D \\ \mathit{shift}(s_\eta) &= \begin{cases} \varepsilon & \text{if } s_\eta = (t_0, \eta(t_0)) \text{ or } s_\eta = \varepsilon \\ s_\eta^{\rightarrow} & \text{otherwise} \end{cases} \end{aligned}$$

The last timing function is *within*, which already appeared in Example 2. The function *within* produces a Boolean valued signal that captures whether there is an event within the timing window provided:

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

2.4 Semantics of TeSSLa$_a$

We define the semantics of TeSSLa$_a$ in terms of evaluation models, as commonly done in SRV [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 *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.

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 give a meaning to each expression E over variables I and O, written $[\![E, \sigma]\!]$, recursively as follows:

– *variable name* ($E = \mathit{name}$):

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

– *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, such that the valuation of every output x variable coincides with the valuation of its 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

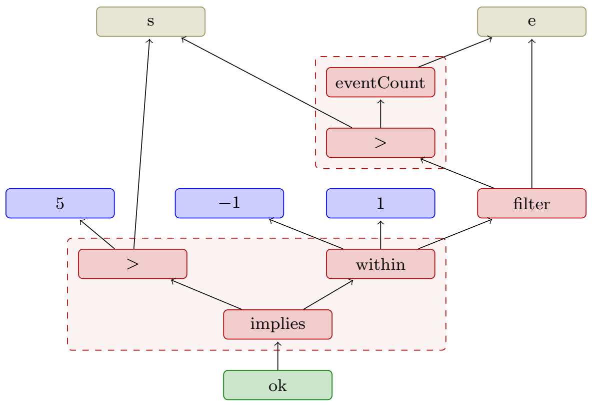


Fig. 1 The dependency graph for the spec in Example 2. Inputs are shown in brown, constants in blue, outputs in green, computation nodes in red and some possible merges of computation nodes in dashed red.

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. Note that a candidate σ 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$_a$ 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 $x \to y$. 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$_a$ specifications must be non-recursive (i.e. for every x, $x \not\to^+ x$), 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$_a$ specifications are well-defined. Figure 1 shows the dependency graph of the specification from Example 2.

$\mathit{add}: \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_D, \quad D \in \{\mathbb{N},\mathbb{Z},\mathbb{R}\}$ $\mathit{add}\ (\sigma_1,\sigma_2)(t) := \quad \sigma_1(t) + \sigma_2(t)$	$\mathit{sub}: \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_D, \quad D \in \{\mathbb{Z},\mathbb{R}\}$ $\mathit{sub}\ (\sigma_1,\sigma_2)(t) := \quad \sigma_1(t) - \sigma_2(t)$
$\mathit{mul}: \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_D, \quad D \in \{\mathbb{N},\mathbb{Z},\mathbb{R}\}$ $\mathit{mul}\ (\sigma_1,\sigma_2)(t) := \quad \sigma_1(t) \cdot \sigma_2(t)$	$\mathit{div}: \mathcal{S}_D \times \mathcal{S}_{D'} \to \mathcal{S}_D, \quad D \in \{\mathbb{N},\mathbb{Z},\mathbb{R}\}$ $\mathit{div}\ (\sigma_1,\sigma_2)(t) := \frac{\sigma_1(t)}{\sigma_2(t)} \quad D' = D \setminus \{0\}$
$\mathit{gt}: \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_\mathbb{B}, \quad D \in \{\mathbb{N},\mathbb{Z},\mathbb{R}\}$ $\mathit{gt}\ (\sigma_1,\sigma_2)(t) := \quad \sigma_1(t) > \sigma_2(t)$	$\mathit{geq}: \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_\mathbb{B}, \quad D \in \{\mathbb{N},\mathbb{Z},\mathbb{R}\}$ $\mathit{geq}\ (\sigma_1,\sigma_2)(t) := \quad \sigma_1(t) \geq \sigma_2(t)$
$\mathit{leq}: \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_\mathbb{B}, \quad D \in \{\mathbb{N},\mathbb{Z},\mathbb{R}\}$ $\mathit{leq}\ (\sigma_1,\sigma_2)(t) := \quad \sigma_1(t) \leq \sigma_2(t)$	$\mathit{eq}: \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_\mathbb{B},$ any D with equality $\mathit{eq}\ (\sigma_1,\sigma_2)(t) := \quad \sigma_1(t) = \sigma_2(t)$
$\mathit{max}: \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_D, \quad D \in \{\mathbb{N},\mathbb{Z},\mathbb{R}\}$ $\mathit{max}\ (\sigma_1,\sigma_2)(t) := \quad \max\{\sigma_1(t),\sigma_2(t)\}$	$\mathit{min}: \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_D, \quad D \in \{\mathbb{N},\mathbb{Z},\mathbb{R}\}$ $\mathit{min}\ (\sigma_1,\sigma_2)(t) := \quad \min\{\sigma_1(t),\sigma_2(t)\}$
$\mathit{abs} \ : \mathcal{E}_D \to \mathcal{E}_D, \quad D \in \{\mathbb{N},\mathbb{Z},\mathbb{R}\}$ $\mathit{abs}\ (\eta)(t) := \begin{cases} \lvert\eta(t)\rvert & \text{if } t \in E(\eta) \\ \perp & \text{otherwise} \end{cases}$	$\mathit{abs} \ : \mathcal{S}_D \to \mathcal{S}_D, \quad D \in \{\mathbb{N},\mathbb{Z},\mathbb{R}\}$ $\mathit{abs}\ (\sigma)(t) := \quad \lvert\sigma(t)\rvert$

Fig. 2 Arithmetic operations with their semantics.

In Section 3 we will use the dependency graph to define operational semantics of an evaluation engine for TeSSLa$_a$ 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 *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$_a$ specification. LOLA [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$_a$, 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$_a$. In TeSSLa$_a$ 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$_a$ 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.

$\mathit{and} \ : \mathcal{S}_{\mathbb{B}} \times \mathcal{S}_{\mathbb{B}} \to \mathcal{S}_{\mathbb{B}}$ $\mathit{and}\,(\sigma_1, \sigma_2)(t) := \sigma_1(t) \wedge \sigma_2(t)$	$\mathit{or} \ : \mathcal{S}_{\mathbb{B}} \times \mathcal{S}_{\mathbb{B}} \to \mathcal{S}_{\mathbb{B}}$ $\mathit{or}\,(\sigma_1, \sigma_2)(t) := \sigma_1(t) \vee \sigma_2(t)$
$\mathit{not} \ : \mathcal{S}_{\mathbb{B}} \to \mathcal{S}_{\mathbb{B}}$ $\mathit{not}\,(\sigma)(t) := \neg\sigma(t)$	$\mathit{neg} \ : \mathcal{E}_{\mathbb{B}} \to \mathcal{E}_{\mathbb{B}}$ $\mathit{neg}\,(\eta)(t) := \begin{cases} \neg\eta(t) & \text{if } t \in E(\eta) \\ \bot & \text{otherwise} \end{cases}$

Fig. 3 Logical operations with their semantics.

$\mathit{maximum} \ : \mathcal{S}_D \to \mathcal{S}_D$ $\mathit{maximum}\,(\sigma)(t) := \max\{\eta(t') \mid t' \leq t\}$	$\mathit{minimum} \ : \mathcal{S}_D \to \mathcal{S}_D$ $\mathit{minimum}\,(\sigma)(t) := \min\{\eta(t') \mid t' \leq t\}$
$\mathit{maximum} : \mathcal{E}_D \times D \to \mathcal{S}_D$ $\mathit{maximum} \ (\eta, d)(t) :=$ $\max(\{d\} \cup \{\eta(t') \mid t' \in E(\eta), t' \leq t\})$	$\mathit{minimum} : \mathcal{E}_D \times D \to \mathcal{S}_D$ $\mathit{minimum} \ (\eta, d)(t) :=$ $\min(\{d\} \cup \{\eta(t') \mid t' \in E(\eta), t' \leq t\})$
$\mathit{timestamps} \ : \mathcal{E}_D \to \mathcal{E}_{\mathbb{T}}$ $\mathit{timestamps}\,(\eta)(t) := \begin{cases} t & \text{if } t \in E(\eta) \\ \bot & \text{otherwise} \end{cases}$	$\mathit{sum} \ : \mathcal{E}_D \to \mathcal{S}_D$ $\mathit{sum}\,(\eta)(t) := \Sigma_{\{t' \in E(\eta) \mid t' \leq t\}} \eta(t')$
$\mathit{sma} : \mathcal{E}_D \times \mathbb{N} \to \mathcal{E}_D, \quad D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\}$ $\mathit{sma} \ (\eta, n)(t) :=$ $\begin{cases} \frac{\sum_{t'' \in \max_n \{t' \in E(\eta) \mid t' \leq t\}} \eta(t'')}{\lvert \max_n \{t' \in E(\eta) \mid t' \leq t\} \rvert} & \text{if } t \in E(\eta) \\ \bot & \text{otherwise} \end{cases}$	$\mathit{mrv} : \mathcal{E}_D \times D \to \mathcal{S}_D$ $\mathit{mrv} \ (\eta, d)(t) :=$ $\begin{cases} \eta(\max E(\eta) \cap [0, t]) & \text{if } E(\eta) \cap [0, t] \neq \emptyset \\ d & \text{otherwise} \end{cases}$
$\mathit{eventCount} : \mathcal{E}_{D_1} \times \mathcal{E}_{D_2} \to \mathcal{S}_{\mathbb{N}}$ $\mathit{eventCount} \quad (\eta_1, \eta_2)(t) := \lvert\{t' \in E(\eta_1) \mid t' \leq t \wedge \forall t'' \leq t \in E(\eta_2) : t' > t''\}\rvert$	

Fig. 4 Aggregation operators with their semantics.

2.5 A Library Builtin Functions and their Semantics

There are five types of functions in TeSSLa$_a$, apart from logical functions: arithmetic functions, aggregations, stream manipulators, timing functions (explained above) and temporal property functions. Figures 2–5 show the set of functions provided by TeSSLa$_a$ as well as 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 $\mathit{mrv} : \mathcal{E}_D \times D \to \mathcal{S}_D$ 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). This function is important for transforming event streams into signals.

$\mathit{changeOf} \;: \mathcal{S}_D \to \mathcal{E}_{\{\top\}}$ $\mathit{changeOf}\,(\sigma)(t) := \begin{cases} \top & \text{if } t \in \Delta(\sigma) \\ \bot & \text{otherwise} \end{cases}$	$\mathit{ifThen} : \mathcal{E}_{D_1} \times \mathcal{S}_{D_2} \to \mathcal{E}_{D_2}$ $\mathit{ifThen} \quad (\eta, \sigma)(t) := \begin{cases} \sigma(t) & \text{if } t \in E(\eta) \\ \bot & \text{otherwise} \end{cases}$
$\mathit{sample} : \mathcal{S}_{D_1} \times \mathcal{E}_{D_2} \to \mathcal{E}_{D_1}$ $\mathit{sample} \quad (\sigma, \eta) := \mathit{ifThen}(\eta, \sigma)$	$\mathit{filter} : \mathcal{E}_D \times \mathcal{S}_{\mathbb{B}} \to \mathcal{E}_D$ $\mathit{filter} \quad (\eta, \sigma)(t) := \begin{cases} \eta(t) & \text{if } t \in E(\eta) \\ & \text{and } \sigma(t) = \mathit{true} \\ \bot & \text{otherwise} \end{cases}$
$\mathit{ifThenElse} : \mathcal{S}_{\mathbb{B}} \times \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_D$ $\mathit{ifThenElse} \quad (\sigma_1, \sigma_2, \sigma_3)(t) :=$ $\begin{cases} \sigma_2(t) & \text{if } \sigma_1(t) = \mathit{true} \\ \sigma_3(t) & \text{otherwise} \end{cases}$	$\mathit{merge} : \mathcal{E}_D \times \mathcal{E}_D \to \mathcal{E}_D$ $\mathit{merge}\ (\eta_1, \eta_2)(t) :=$ $\begin{cases} \eta_1(t) & \text{if } t \in E(\eta_1) \\ \eta_2(t) & \text{if } t \in E(\eta_2) \setminus E(\eta_1) \\ \bot & \text{otherwise} \end{cases}$
$\mathit{occursAny} : \mathcal{E}_{D_1} \times \mathcal{E}_{D_2} \to \mathcal{E}_{\{\top\}}$ $\mathit{occursAny} \quad (\eta_1, \eta_2)(t) :=$ $\begin{cases} \top & \text{if } t \in E(\eta_1) \cup E(\eta_2) \\ \bot & \text{otherwise} \end{cases}$	$\mathit{occursAll} : \mathcal{E}_{D_1} \times \mathcal{E}_{D_2} \to \mathcal{E}_{\{\top\}}$ $\mathit{occursAll} \quad (\eta_1, \eta_2)(t) :=$ $\begin{cases} \top & \text{if } t \in E(\eta_1) \cap E(\eta_2) \\ \bot & \text{otherwise} \end{cases}$

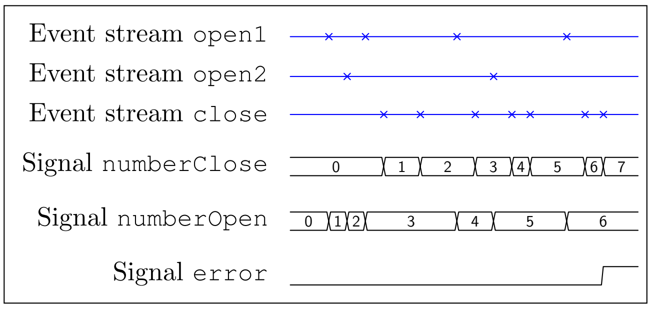
Fig. 5 Stream manipulation operators with their semantics.

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.

Example 4 Consider the following specification:

```
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 *true* as long as too many close events occurred (hence, which is *true* if an error happened), is the output.



The stream picture above shows a possible run. ■

3 Online Evaluation of Efficiently Monitorable Specifications

In runtime verification [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$_a$ specifications. To ease the presentation in this section we restrict TeSSLa$_a$ specifications to refer to present and past values only (even though the results can be easily extended to arbitrary TeSSLa$_a$ specifications). These specifications are known as efficiently monitorable [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$_a$ 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$_a$ 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$_a$ 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$_a$ construct.

Let $\varphi(I, O)$ be a TeSSLa$_a$ specification (which, without loss of generality we assume to be flat). Let G be the dependency graph of φ, and N the collection of vertices of G. The evaluation engine of φ 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$_a$ 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 time-stamp 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 $\mathit{apply} : S \times T \to S$ 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 $\mathit{node} : T \setminus \{\lambda\} \to N$ 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 $i > 1$, $\mathit{node}(\tau_i)$ is enabled at state s_{i-1} and $\mathit{apply}(s_{i-1}, \tau_i) = s_i$.

We consider here finite and infinte inputs and outputs. Note that every finite run prefix can be extended to an infinite run by adding λ transitions at the end if necessary. A finite stream of events on source x will always contain a final event $(x, \mathit{prog}, \infty)$ (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 ω-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 Σ_v for the alphabet of timed letters (v, d, t) for stream variable v, $\Sigma_{in} = \cup_{v \in I} \Sigma_v$ for the input alphabet and $\Sigma_{in} = \cup_{v \in O} \Sigma_v$ for the output alphabet. We use

Σ_{in}^* to denote finite strings from Σ_{in} (in increasing order of time-stamps), Σ_{in}^ω for ω strings of events from Σ_{in} (in increasing orther and with a finite number of events between any two bounds) and Σ_{in}^∞ for the union of Σ_{in}^* and Σ_{in}^ω. The definitions of Σ_{out}^*, Σ_{out}^ω and Σ_{out}^∞ are analogous.

Given a timed letter a we use $t(a)$ to denote its time component, we use $\mathit{source}(a)$ to represent the source of a and $\mathit{value}(a)$ for the value. Given a timed word w we use $L(w)$ for the timed letters occurring in w and $\mathit{pos}(w, a)$ 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 $\mathit{output}(r)$ for the concatenation of the outputs produced in r, and $\mathit{output}(r, x)$ 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-sychronized, 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 $\mathit{pos}(w, a) < \mathit{pos}(w, b)$ and $\mathit{source}(a) = \mathit{source}(b)$ then $t(a) \leq t(b)$.
- *synchronized* whenever for every $a, b \in L(w)$ if $\mathit{pos}(w, a) < \mathit{pos}(w, b)$ then $t(a) \leq t(b)$.

Example 5 Consider two input sources x and y, and let x receive the input $(x, \mathrm{T}, 0)(x, \mathrm{T}, 3)$ and y receive $(y, \mathrm{F}, 1)(y, \mathrm{F}, 6)$. The following two inputs
- $w_1 : (x, \mathrm{T}, 0)(y, \mathrm{F}, 1)(x, \mathrm{T}, 3)(y, \mathrm{F}, 6)$ and
- $w_2 : (x, \mathrm{T}, 0)(y, \mathrm{F}, 1)(y, \mathrm{F}, 6)(x, T, 3)$

are in-order. However, w_1 is synchronized but w_2 is not, because in w_2 $(x, \mathrm{T}, 3)$ is received after $(y, \mathrm{F}, 6)$ 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 *evenCount* 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 $(a, prog, \infty)$ 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$_a$ 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 ε). 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 $w \in \Sigma_{in}^{\infty}$, and for all $v, v' \in \Sigma_{out}^{\infty}$, 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 $\mathit{timed} : \Sigma_{out}^{\infty} \to \Sigma_{out}^{\infty}$ 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 $w \in \Sigma_{in}^{\infty}$ and for all $v, v' \in \Sigma_{out}^{\infty}$ with $(w, v) \in F$ and $(w, v') \in F$, $\mathit{timed}(v) = \mathit{timed}(v')$.

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 $w \in \Sigma_{in}^{\infty}$. We say that F and G are observational equivalent, and we write $F \equiv_O G$ whenever for all $v, u \in \Sigma_{out}^{\infty}$ with $(w, v) \in F$ and $(w, u) \in G$, $\mathit{timed}(v) = \mathit{timed}(u)$.

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 $\mathit{output}(r, x) = \mathit{output}(r', x)$.*

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. □

Lemma 2 *Let r and r' be two runs of E such that $\mathit{output}(r,x) = \mathit{output}(r,x')$ for every x. Then $\mathit{timed}(\mathit{output}(r)) = \mathit{timed}(\mathit{output}(r'))$.*

Proof By contradiction, assume $\mathit{timed}(\mathit{output}(r)) \neq \mathit{timed}(\mathit{output}(r'))$ 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 $\mathit{output}(r,x)$, but this event is not in $\mathit{output}(r',x)$, which contradicts Lemma 1. □

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$_a$ 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$_a$ 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$_a$. 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 $\mathit{peek}(Q) \neq \mathit{nil}$. We use the statement $\mathit{emit}(a)$ 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 $\mathit{add} : \mathcal{S}_D \times \mathcal{S}_D \to \mathcal{S}_D$ that has two input queues A and B and the following code:

```
1  $prog \leftarrow false$
2  **if** $peek(A).time = peek(B).time$ **then**
3      $a \leftarrow peek(A).value, b =\leftarrow peek(B).value,$
4      $t \leftarrow peek(A).time$
5      **if** $a = progress$ **and** $b = progress$ **then**
6          $prog \leftarrow true$
7      **else if** $a = progress$ **then** $a \leftarrow last(A).value$
8      **else if** $b = progress$ **then** $b \leftarrow last(B).value$
9      $dequeue(A),$
10     $dequeue(B)$
11 **else if** $peek(A).time < peek(B).time$ **then**
12     $a \leftarrow peek(A).value,\ b \leftarrow last(B).value,$
13     $t \leftarrow peek(A).time$
14     **if** $a = progress$ **then** $prog = true$
15     $dequeue(A)$
16 **else**
17     $a \leftarrow last(A).value,\ b \leftarrow peek(B).value,$
18     $t \leftarrow peek(B).time$
19     **if** $b = progress$ **then** $prog \leftarrow true$
20     $dequeue(B)$
21 **if** $prog$ **then**
22     $emit(progress, time = t)$
23 **else**
24     $emit(a + b, time = t)$
```

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 $peek(Q).value = progress$ 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, unless the domain is `unit`, *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 24 as follows.

– For *sub*:

```
$emit(value = a - b, time = t)$
```

– For *mul*:

```
$emit(value = a \cdot b, time = t)$
```

– For *div*:

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

– For *gt*:

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

– For *geq*:

```
$emit(value = a \geq b, time = t)$
```

– For *leq*:

```
$emit(value = a \leq b, time = t)$
```

– For *eq*:

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

– For *max*:

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

– For *min*:

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

– For *and*:

```
$emit(value = a \wedge b, time = t)$
```

– For *or*:

```
$emit(value = a \vee b, time = t)$
```

3.3.2 Unary Arithmetic Functions

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

```
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: $abs : \mathcal{E}_D \rightarrow \mathcal{E}_D$, $not : \mathcal{S}_\mathbb{B} \rightarrow \mathcal{S}_\mathbb{B}$, $neg : \mathcal{E}_\mathbb{B} \rightarrow \mathcal{E}_\mathbb{B}$, $timestamps : \mathcal{E}_D \rightarrow \mathcal{E}_\mathbb{T}$, $mrv : \mathcal{E}_D \times D \rightarrow \mathcal{S}_D$ and $changeOf : \mathcal{S}_D \rightarrow \mathcal{E}_{\{\top\}}$. This is accomplished by changing line 4 in the code of *abs* with the corresponding operation.

3.3.3 Aggregation Functions

For $sum : \mathcal{E}_D \rightarrow \mathcal{S}_D$ 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     $emit(progress, time = peek(A).time)$
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: $maximum : \mathcal{E}_D \times D \rightarrow \mathcal{S}_D$, $maximum : \mathcal{S}_D \rightarrow \mathcal{S}_D$, $minimum : \mathcal{E}_D \times D \rightarrow \mathcal{S}_D$, $minimum : \mathcal{S}_D \rightarrow \mathcal{S}_D$ and $sma : \mathcal{E}_D \times D$. This is accomplished by replacing line 4 with the right operation.

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

```
$prog \leftarrow false$
if $peek(A).time = peek(B).time$ then
    $t \leftarrow peek(B).time$
    if $peek(A).value = progress$ and $peek(B).value = progress$ then
        $prog \leftarrow true$
    else if $peek(B).value = progress$ then
        $state \leftarrow state + 1$
    else
        $state \leftarrow 0$
    $dequeue(A)$
    $dequeue(B)$
else if $peek(A).time < peek(B).time$ then
    $t \leftarrow peek(A).time$
    if $peek(A).value = progress$ then
        $prog \leftarrow true$
    else
        $state = state + 1$
    $dequeue(A)$
else
    $t \leftarrow peek(B).time$
    if $peek(B).value = progress$ then
        $prog \leftarrow true$
    else
        $state = 0$
    $dequeue(B)$
if $prog$ then
        $emit(value = progress, t)$
else
        $emit(value = state, t)$
```

3.3.4 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:

```
$prog \leftarrow false$
if $peek(A).time = peek(B).time$ then
    $t \leftarrow peek(A).time$
    if $peek(A).value = progress$ then
        $prog \leftarrow true$
    else if $peek(B).value = progress$ then
        $v \leftarrow last(B).value$
    else
        $v \leftarrow peek(B).value$
    $dequeue(A)$, $dequeue(B)$
else if $peek(A).time < peek(B).time$ then
    $t \leftarrow peek(A).time$
    if $peek(A).value = progress$ then
        $prog \leftarrow true$
    else
        $v \leftarrow last(B).value$
    $dequeue(A)$
else
    $t \leftarrow peek(B).time$
    $prog \leftarrow true$
    $dequeue(B)$
if $prog$ then
    $emit(value = progress, t)$
else
    $emit(value = v, t)$
```

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:

```
$prog \leftarrow true$
if $peek(A).time = peek(B).time$ then
    $t \leftarrow peek(A).time$
    if ($peek(B).value = progress$ and $last(B).value$) or $peek(B).value$ then
```

```
        $v \leftarrow peek(A).value$
    else
        $prog \leftarrow true$
    $dequeue(A)$, $dequeue(B)$
else if $peek(A).time < peek(B).time$ then
    $t \leftarrow peek(A).time$
    if $last(B).value$ then
        $v \leftarrow peek(A).value$
    else
        $prog \leftarrow true$
    $dequeue(A)$
else
    $prog \leftarrow true$
if $prog$ then
    $emit(value = progress, time = t)$
else
    $emit(value = v, time = t)$
```

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:

```
$prog \leftarrow false$
if $peek(A).time = peek(B).time$ then
    $t \leftarrow peek(A).time$
    if $peek(A).value = progress$ then
        $v \leftarrow peek(B).value$
    else
        $v \leftarrow peek(A).value$
    $dequeue(A)$, $dequeue(B)$
else if $peek(A).time < peek(B).time$ then
    $t \leftarrow peek(A).time$
    $v \leftarrow peek(A).value$
    $dequeue(A)$
else
    $t \leftarrow peek(B).time$
    $v \leftarrow peek(B).value$
    $dequeue(B)$
$emit(value = v, time = t)$
```

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.

3.3.5 Timing Functions

For *delay* : $\mathcal{S}_D \times \mathbb{T} \times D \to \mathcal{S}_D$ we have one input queue 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 the node iteratively computes $within(a, b, x)$. For the operative semantics we assume $a < b \leq 0$. We then have the following code:

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

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`

code over multiple cores or even dsitributed 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$_a$ 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$_a$ 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.

#cores	time [s]	out-liers	error [ms]
1	1.69650	0	3.6
2	0.84310	1	5.8
4	0.58390	1	2.5
8	0.62420	1	2.2
12	0.63550	0	3.3

#cores	time [s]	out-liers	error [ms]
1	3.44400	0	13.0
2	1.96590	0	7.4
4	1.31000	1	11.0
8	1.20700	0	12.0
12	1.38080	3	5.6

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.

#events	time [s]	out-liers	error [ms]	time/# [ms]
500	0.19599	0	0.5	0.39198
1000	0.21397	1	0.5	0.21397
1500	0.23301	0	0.6	0.15534
2000	0.25167	0	0.8	0.12584
2500	0.27487	0	0.6	0.10995
3000	0.30048	0	0.8	0.10016
3500	0.32120	0	1.4	0.09177
4000	0.34940	1	1.3	0.08735
4500	0.37226	1	1.0	0.08272
5000	0.39520	0	1.1	0.07904
5500	0.41585	1	1.0	0.07561
6000	0.44130	0	2.0	0.07355
6500	0.46030	0	1.5	0.07082
7000	0.48505	1	0.7	0.06929
7500	0.52000	0	3.8	0.06933
8000	0.54940	0	2.8	0.06868
8500	0.56480	0	2.7	0.06645
9000	0.58730	0	3.1	0.06526
9500	0.61440	0	2.7	0.06467
10000	0.64220	0	4.7	0.06422

#events	time [s]	out-liers	error [ms]	time/# [ms]
500	0.21882	1	0.7	0.43764
1000	0.26770	0	1.6	0.26770
1500	0.32750	2	1.4	0.21833
2000	0.39510	2	1.6	0.19755
2500	0.45360	2	3.3	0.18144
3000	0.50420	0	4.0	0.16807
3500	0.59120	0	7.4	0.16891
4000	0.64950	1	4.8	0.16238
4500	0.69230	0	4.3	0.15384
5000	0.73100	0	4.4	0.14620
5500	0.78590	2	5.1	0.14289
6000	0.83300	0	7.8	0.13883
6500	0.88930	0	9.0	0.13682
7000	0.96680	1	4.7	0.13811
7500	0.98250	0	5.8	0.13100
8000	1.06570	0	9.3	0.13321
8500	1.11460	0	7.6	0.13113
9000	1.17280	2	5.9	0.13031
9500	1.22670	0	5.6	0.12913
10000	1.28500	0	12.0	0.12850

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.

#nodes	time [s]	out-liers	error [ms]	time/# [ms]
8	0.42120	0	1.6	52.65000
16	0.60930	0	4.3	38.08125
24	0.81280	0	4.1	33.86667
32	1.02110	2	4.2	31.90938
48	1.17420	0	5.9	24.46250
64	1.26470	1	4.4	19.76094
96	1.50900	0	12.0	15.71875
128	1.82700	0	27.0	14.27344

#nodes	time [s]	out-liers	error [ms]	time/# [ms]
8	0.46220	0	3.3	57.77500
16	0.71590	0	6.1	44.74375
24	0.94260	0	5.7	39.27500
32	1.18360	1	6.6	36.98750
48	1.33200	0	12.0	27.75000
64	1.47520	1	8.1	23.05000
96	1.75950	0	8.6	18.32813
128	2.28000	1	10.0	17.81250

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.

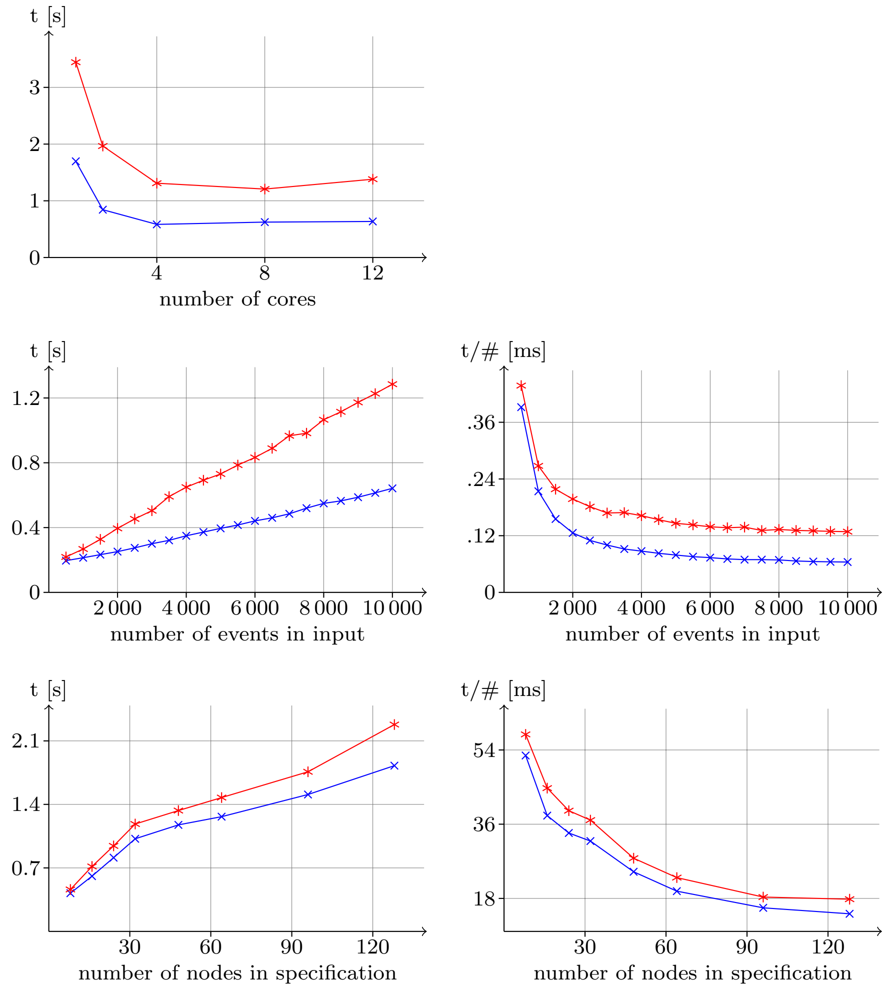
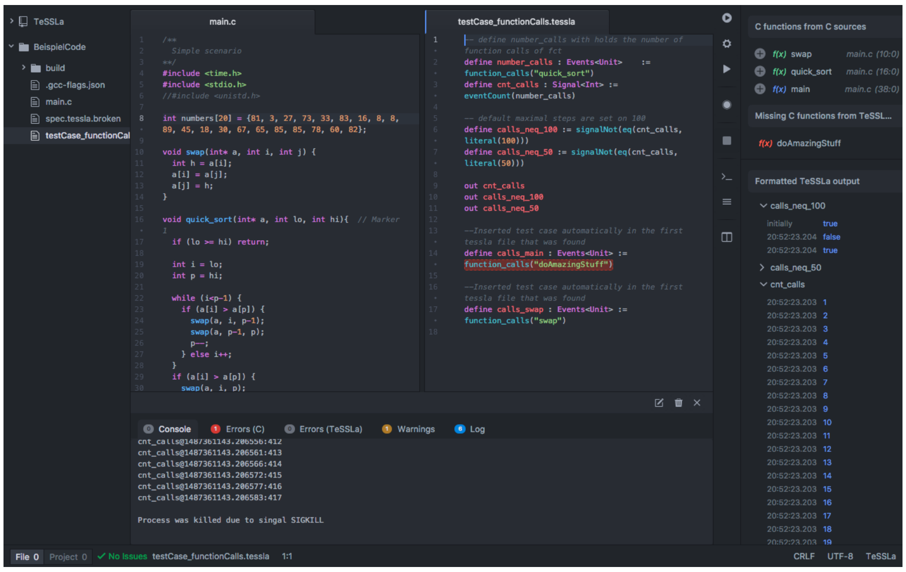


Fig. 6 Benchmarks. The plots on the left show the total time and the plots on the right show the relative time per event or node, resp. The blue line with cross marks indicates the runtime of the chain specification and the red line with asterisk marks indicates the runtime of the tree specification.

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



```
/**
  Simple scenario
**/
#include <time.h>
#include <stdio.h>
//#include <unistd.h>

int numbers[20] = {81, 3, 27, 73, 33, 83, 16, 8, 8,
89, 45, 18, 30, 67, 65, 85, 85, 78, 60, 82};

void swap(int* a, int i, int j) {
  int h = a[i];
  a[i] = a[j];
  a[j] = h;
}

void quick_sort(int* a, int lo, int hi){  // Marker
1
  if (lo >= hi) return;

  int i = lo;
  int p = hi;

  while (i<p-1) {
    if (a[i] > a[p]) {
      swap(a, i, p-1);
      swap(a, p-1, p);
      p--;
    } else i++;
  }
  if (a[i] > a[p]) {
    swap(a, i, p);
```

```
-- define number_calls with holds the number of
function calls of fct
define number_calls : Events<Unit>    :=
function_calls("quick_sort")
define cnt_calls : Signal<Int> :=
eventCount(number_calls)

-- default maximal steps are set on 100
define calls_neq_100 := signalNot(eq(cnt_calls,
literal(100)))
define calls_neq_50 := signalNot(eq(cnt_calls,
literal(50)))

out cnt_calls
out calls_neq_100
out calls_neq_50

--Inserted test case automatically in the first
tessla file that was found
define calls_main : Events<Unit> :=
function_calls("doAmazingStuff")

--Inserted test case automatically in the first
tessla file that was found
define calls_swap : Events<Unit> :=
function_calls("swap")
```

```
cnt_calls@1487361143.206556:412
cnt_calls@1487361143.206561:413
cnt_calls@1487361143.206566:414
cnt_calls@1487361143.206572:415
cnt_calls@1487361143.206577:416
cnt_calls@1487361143.206583:417

Process was killed due to singal SIGKILL
```

C functions from C sources

f(x)	swap	main.c (10:0)
f(x)	quick_sort	main.c (16:0)
f(x)	main	main.c (38:0)

Missing C functions from TeSSL...

f(x) doAmazingStuff

Formatted TeSSLa output

```
calls_neq_100
  initially     true
  20:52:23.204  false
  20:52:23.204  true
calls_neq_50
cnt_calls
  20:52:23.203  1
  20:52:23.203  2
  20:52:23.203  3
  20:52:23.203  4
  20:52:23.203  5
  20:52:23.203  6
  20:52:23.203  7
  20:52:23.203  8
  20:52:23.203  9
  20:52:23.203  10
  20:52:23.203  11
  20:52:23.203  12
  20:52:23.203  13
  20:52:23.203  14
  20:52:23.203  15
  20:52:23.203  16
  20:52:23.203  17
  20:52:23.203  18
  20:52:23.203  19
```

Fig. 7 Atom plugin for TeSSLa$_a$ based runtime verification of C programs.

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 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$_a$ 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 TeSSLa$_a$, an acyclic fragment of the stream-based runtime verification language TeSSLa 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 TeSSLa$_a$ 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 TeSSLa and its compiler.

References

1. Alur, R., Feder, T., Henzinger, T.A.: The benefits of relaxing punctuality. J. ACM (1996)
2. Asarin, E., Caspi, P., Maler, O.: Timed regular expressions. J. ACM **49**(2), 172–206 (2002)
3. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: Proc. of VMCAI'04, LNCS 2937, pp. 44–57. Springer (2004)
4. Bartocci, E., Falcone, Y. (eds.): Lectures on Runtime Verification - Introductory and Advanced Topics, *LNCS*, vol. 10457. Springer (2018)
5. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM T. Softw. Eng. Meth. **20**(4), 14 (2011)
6. Berry, G.: Proof, language, and interaction: essays in honour of Robin Milner, chap. The foundations of Esterel, pp. 425–454. MIT Press (2000)
7. Bozelli, L., Sánchez, C.: Foundations of Boolean stream runtime verification. In: In Proc. RV'14, *LNCS*, vol. 8734, pp. 64–79. Springer (2014)
8. Caspi, P., Pouzet, M.: Synchronous Kahn Networks. In: Proc. of ICFP'96, pp. 226–238. ACM Press (1996)
9. Convent, L., Hungerecker, S., Leucker, M., Scheffel, T., Schmitz, M., Thoma, D.: Tessla: Temporal stream-based specification language. In: Formal Methods: Foundations and Applications - 21th Brazilian Symposium, SBMF 2018, Recife, Brazil, November 26 - November 30, 2018, Proceedings, Lecture Notes in Computer Science. Springer (2018)
10. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: runtime monitoring of synchronous systems. In: Proc. of TIME'05, pp. 166–174. IEEE (2005)
11. Decker, N., Dreyer, B., Gottschling, P., Hochberger, C., Lange, A., Leucker, M., Scheffel, T., Wegener, S., Weiss, A.: Online Analysis of Debug Trace Data for Embedded Systems. In: DATE. IEEE (2018)
12. Decker, N., Gottschling, P., Hochberger, C., Leucker, M., Scheffel, T., Schmitz, M., Weiss, A.: Rapidly Adjustable Non-Intrusive Online Monitoring for Multi-core Systems. In: 20th Brazilian Symposium on Formal Methods (SBMF 2017). Springer (2017)
13. Donzé, A., Maler, O., Bartocci, E., Nickovic, D., Grosu, R., Smolka, S.A.: On temporal logic and signal processing. In: In Proc. of ATVA'12, *LNCS*, vol. 7561, pp. 92–106. Springer (2012)
14. Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., Campenhout, D.V.: Reasoning with temporal logic on truncated paths. In: Proc. of CAV'03, *LNCS 2725*, vol. 2725, pp. 27–39. Springer (2003)
15. Eliot, C., Hudak, P.: Functional reactive animation. In: Proc. of ICFP'07, pp. 163–173. ACM (1997)
16. Faymonville, P., Finkbeiner, B., Schwenger, M., Torfah, H.: Real-time stream-based monitoring. CoRR **abs/1711.03829** (2017). URL `http://arxiv.org/abs/1711.03829`
17. Gautier, T., Le Guernic, P., Besnard, L.: SIGNAL: A declarative language for synchronous programming of real-time systems. In: Proc. of FPCA'87, LNCS 274, pp. 257–277. Springer (1987)
18. Goodloe, A.E., Pike, L.: Monitoring distributed real-time systems: A survey and future directions. Tech. rep., NASA Langley Research Center (2010)
19. Gorostiaga, F., Sánchez, C.: Striver: Stream runtime verification for real-time event-streams. In: Proc. of the 18th Int'l Conf on Runtime Verification (RV'18), *LNCS*, vol. 11237, pp. 282–298. Springer (2018)
20. Halbwachs, N., Caspi, P., Pilaud, D., Plaice, J.: Lustre: a declarative language for programming synchronous systems. In: Proc. of POPL'87, pp. 178–188. ACM Press (1987)
21. Havelund, K., Goldberg, A.: Verify your runs. In: Proc. of VSTTE'05, LNCS 4171, pp. 374–383. Springer (2005)
22. Havelund, K., Roşu, G.: Synthesizing monitors for safety properties. In: Proc. of TACAS'02, LNCS 2280, pp. 342–356. Springer (2002)
23. Hewitt, C., Bishop, P., Steiger, R.: A Universal Modular ACTOR Formalism for Artificial Intelligence. IJCAI pp. 235–245 (1973)

24. Jaksic, S., Bartocci, E., Grosu, R., Kloibhofer, R., Nguyen, T., Nickovic, D.: From signal temporal logic to FPGA monitors. In: Proc. of MEMOCODE 2015, pp. 218–227 (2015)
25. Jaksic, S., Bartocci, E., Grosu, R., Nickovic, D.: Quantitative monitoring of STL with edit distance. In: Prov. of RV'16, *LNCS*, vol. 10012, pp. 201–218 (2016)
26. Leucker, M.: Teaching runtime verification. In: Proc. of RV'11, no. 7186 in LNCS, pp. 34–48. Springer (2011)
27. Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M., Schramm, A.: TeSSLa: Runtime verification of non-synchronized real-time streams. In: Proc. of the 33rd Symposium on Applied Computing (SAC'18). ACM (2018)
28. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Logic Algebr. Progr. **78**(5), 293–303 (2009)
29. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: FTRTFT, pp. 152–166 (2004)
30. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Springer, New York (1995)
31. Pike, L., Goodloe, A., Morisset, R., Niller, S.: Copilot: A hard real-time runtime monitor. In: Proc. of RV'10, LNCS 6418. Springer (2010)
32. Pnueli, A., Zaks, A.: PSL model checking and run-time verification via testers. In: Proc. of FM'06, LNCS 4085, pp. 573–586. Springer (2006)
33. Roşu, G., Havelund, K.: Rewriting-based techniques for runtime verification. Automated Software Engineering **12**(2), 151–197 (2005)
34. Selyunin, K., Jaksic, S., Nguyen, T., Reidl, C., Hafner, U., Bartocci, E., Nickovic, D., Grosu, R.: Runtime monitoring with recovery of the sent communication protocol. In: Proc. of CAV'17, *LNCS*, vol. 10426, pp. 336–355. Springer (2017)
35. Selyunin, K., Nguyen, T., Bartocci, E., Grosu, R.: Applying runtime monitoring for automotive electronic development. In: Proc. of RV'16, *LNCS*, vol. 10012, pp. 462–469 (2016)
36. Sen, K., Roşu, G.: Generating optimal monitors for extended regular expressions. ENTCS **89**(2), 226–245 (2003)