TeSSLa: Temporal Stream-Based Specification Language

Lukas Convent$^{(\boxtimes)}$, Sebastian Hungerecker$^{(\boxtimes)}$, Martin Leucker$^{(\boxtimes)}$, Torben Scheffel$^{(\boxtimes)}$, Malte Schmitz$^{(\boxtimes)}$, and Daniel Thoma$^{(\boxtimes)}$

Institute for Software Engineering and Programming Languages, University of Lübeck, Lübeck, Germany
{convent,hungerecker,leucker,scheffel,schmitz, thoma}@isp.uni-luebeck.de

Abstract. Runtime verification is concerned with monitoring program traces. In particular, stream runtime verification (SRV) takes the program trace as input streams and incrementally derives output streams. SRV can check logical properties and compute temporal metrics and statistics from the trace. We present TeSSLa, a temporal stream-based specification language for SRV. TeSSLa supports timestamped events natively and is hence suitable for streams that are both sparse and fine-grained, which often occur in practice. We prove results on TeSSLa's expressiveness and compare different TeSSLa fragments to (timed) automata, thereby inheriting various decidability results. Finally, we present a monitor implementation and prove its correctness.

1 Introduction

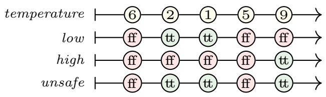
The essence of software verification is to check whether a program meets its specification. Runtime verification (RV) is an applied formal technique that has been established as a complement to traditional verification techniques such as model checking [19,22]. Compared to static verification, RV considers only a single run of a system and checks whether it satisfies a property. Thus, RV can be seen as a lightweight, but formal extension to testing and debugging. RV can be applied offline to previously recorded traces or online to evaluate correctness properties at the runtime of the system under scrutiny. Typically, a property to be checked is specified as a logical formula, e.g. in (past time) LTL, and then synthesized to a monitor which can evaluate a run [5,20]. Stream runtime verification (SRV) [7], as pioneered by the language LOLA [10,15], takes a different approach by incrementally relating a set of input streams to a set of output streams. This allows not only the monitoring of correctness properties but also of quantitative measures. In this paper we introduce the novel temporal stream-based specification

This work is supported in part by the European COST Action ARVI, the BMBF project ARAMiS II with funding ID 01 IS 16025, and the European Horizon 2020 project COEMS under number 732016.

© The Author(s) 2018
T. Massoni and M. R. Mousavi (Eds.): SBMF 2018, LNCS 11254, pp. 144–162, 2018.
https://doi.org/10.1007/978-3-030-03044-5_10

language TeSSLa which is tailored for SRV of cyber-physical systems, where timing is a critical issue. While traditional SRV approaches process event streams without considering timing information, TeSSLa supports timestamped events natively, which allows efficient processing of streams with sparse and fine-grained event sequences. Preliminary versions of TeSSLa have already been studied with regard to their usability to monitor trace data generated by embedded tracing units of processors [11]; how to implement stream-based monitors on hardware has been studied in theory [23] and practice [12]. These versions share the basic idea of transforming timed event streams but they did not allow for recursive equations and comprised only a set of ad-hoc operators. In this paper we define a minimal language with support for recursive definitions that allows us to obtain strong guarantees for evaluation algorithms, expressiveness results and meaningful fragments. While the practical applicability of such a language has been demonstrated by the previous papers, these papers lack a concise and clear theoretical basis and investigation. As an example for SRV, consider the following specification which checks whether a measured temperature stays within given boundaries. For every new event (measurement) on the temperature stream, new events on the derived streams *low*, *high* and *unsafe* are computed:

$$\begin{aligned} \mathit{low} &:= \mathit{temperature} < 3 \\ \mathit{high} &:= \mathit{temperature} > 8 \\ \mathit{unsafe} &:= \mathit{low} \lor \mathit{high} \end{aligned}$$

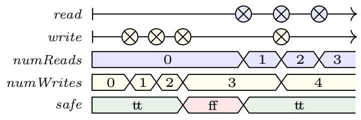


SRV is a combination of complex event processing (CEP) and traditional RV approaches: Streams are transformed into streams and there is not only one final verdict but the output is a stream of the property being evaluated at every temperature change. Furthermore, the user gets more detailed information about why an error occurred by being able to distinguish between the two separate causes *low* and *high*.

In the rest of this section we introduce the main features of TeSSLa and contrast them with related specification languages. The next section presents the language and its semantics formally, in Sect. 3 we present several results regarding the expressiveness of TeSSLa and in Sect. 4 we focus on comparing (fragments of) the language to variants of (timed) automata. Finally in Sect. 5 we discuss different approaches to implement TeSSLa monitors and present our TeSSLa tool suite. An extended preprint version of this paper is available as [9].

Asynchronous Streams. In the previous example of traditional SRV, every stream has an event for every step of the system. TeSSLa requires the events of all streams to be in a global order, but doesn't require all streams to have simultaneous events. As a consequence, both sparse and high-frequency streams can be modeled. As cyber-physical systems often give rise to streams at unstable frequencies or continuous signals, this asynchronous setting is especially suitable. Consider as an example a ring buffer where the number of write accesses should not exceed the number of read accesses too much:

$$\begin{aligned} \mathit{numReads} &:= \mathbf{count}(\mathit{read}) \\ \mathit{numWrites} &:= \mathbf{count}(\mathit{write}) \\ \mathit{safe} &:= \mathit{numWrites} - \mathit{numReads} \leq 2 \end{aligned}$$



Read and write events occur independently at different frequencies. The derived stream *numReads* (*numWrites*) counts the number of events of the input stream *read* (*write*). While the *read* and *write* streams contain only discrete events, the number of events can be seen as a piece-wise constant signal with the initial value of 0. The difference between the two signals is evaluated every time one of the two signals changes its value using the *last known value* of both signals. We call this concept *signal semantics*: TeSSLa handles internally only streams of discrete events, but one can express operators following signal semantics in TeSSla and hence these discrete events can be seen as those points in time where the signal changes its value. In these introductory examples operators are automatically lifted to signal semantics, which is formally introduced as the **slift** operator later.

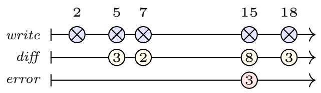
Recursive Equations. Like existing SRV approaches, TeSSLa relates a set of input streams to a set of output streams via mutually recursive equations, which allows self-references to the past, e.g. counting events of a stream x as in the previous example is expressed in TeSSLa as follows:

$$\mathit{count} := \mathbf{merge}(\mathbf{last}(\mathit{count}, x) + 1, 0)$$

The **last** operator outputs the last known value of the *count* stream, on every event of the stream x. The base of the recursion is provided by merging with 0, which is a stream with one initial event of value 0. Since **last** only refers to events strictly lying in the past, the unique solution of such recursive equations can be computed incrementally (see Sect. 2).

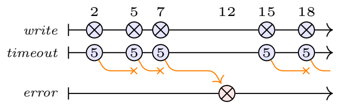
Time as First-Class Citizen. In TeSSLa, every event has a timestamp which can be accessed via the **time** operator. Since every event has a timestamp which is referring to a global clock and is unique for its stream, accessing the timestamps of events serves two purposes: Accessing the global order of events by comparing timestamps and performing calculations with the timestamps. Consider e.g. the following specification which checks whether the lapse of time between two write events exceeds 5 time units and outputs the overtime if it does:

$$\begin{aligned} \mathit{diff} &:= \mathbf{time}(\mathit{write}) - \mathbf{last}(\mathbf{time}(\mathit{write}), \mathit{write}) \\ \mathit{error} &:= \mathbf{filter}(\mathit{diff} > 5, \mathit{diff} - 5) \end{aligned}$$



In the example, the stream $\mathit{diff} - 5$ is filtered by the condition $\mathit{diff} > 5$. Note that the property violation is only reported when the delayed event happens. To report such errors as soon as possible, TeSSLa has the ability to create events at certain points in time via the **delay** operator. The following specification checks the same property but raises a unit event on the *error* stream as soon as we know that there was no *write* event in time:

$$\begin{aligned} timeout &:= \mathbf{const}(5)(write) \\ error &:= \mathbf{delay}(timeout, write) \end{aligned}$$



The **delay** function works as a timer, which is set to a timeout value with the first argument and reset with any event on the second argument. In the example, the function $\mathbf{const}(5)(write)$ maps the values of events to the constant value of 5, which is then used as timeout value. While in all the other examples the derived streams only contain events with timestamps taken from the input streams, in this example events with additional timestamps are generated. Like **last**, the **delay** operator can be used in recursive equations, for example the equation

$$period := \mathbf{merge}(\mathbf{const}(5)(\mathbf{delay}(period, \mathbf{unit})), 5)$$

produces an infinite stream with an event every 5 time units. The **merge** is used to provide a base case for the recursion and **const** is used to map the value of the generated events to 5 so that they can be used as the new timeout value.

Efficient Parallel Evaluation. TeSSLa's design follows two principles to allow efficient evaluation on parallel hardware: *Explicit memory usage* and *local operator composition*. If TeSSLa operates only on streams with bounded data-types of constant size, then the operators only need finite memory because every operator only needs to store at most one data value. This allows implementations on systems without random access memory, e.g. FPGAs or embedded systems. TeSSLa consists of a small set of primitive operators which can be flexibly combined. The TeSSLa semantics is defined in a way that allows a local composition of the individual operators, which can be realized via message passing without the need for global synchronization. Because of an explicit notion of progress for every stream describing how far the stream is known, local message passing is also sufficient to compute solutions for the recursive TeSSLa equations. Implementing an efficient evaluation on FPGAs is part of our EU research project COEMS[1].

Related Work and Comparison. LOLA [10,15] is a synchronous stream specification language in the following sense: Events arrive in discrete steps and for every step, all input streams provide an event and all output streams produce an event, which means that it is not suitable for handling events with arbitrary real-time timestamps arriving at variable frequencies. The not yet formally published RTLola [16] is an extension of LOLA which introduces asynchronous streams to perform aggregations over real-time intervals. A major difference between RTLola and TeSSLa is that RTLola focuses on splitting input streams and aggregating over them, whereas TeSSLa provides a more general framework that in particular allows the (recursive) definition of aggregation operators while giving strict memory guarantees at the same time. Focus [8] is a formalism for the specification of stream-based systems. Their timed streams progress by discrete

[1] https://www.coems.eu.

ticks that separate events inbetween, thereby allowing multiple events at the same timestamp. The synchronous stream programming languages Lustre [18], Esterel [6] and Signal [17], the stream specification language Copilot [25] as well as the class of functional reactive programming (FRP) languages [14] allow the description of the transformation in a linear style, i.e. an input stream is read chronologically and is thereby evaluated. TeSSLa also supports linear evaluation because there are no future-references and the number of past-references is limited by the specification size. The only complement to linear evaluation is the creation of additional events via the **delay** operator. Quantitative regular expressions (QREs) [2] and logics like Signal Temporal Logic (STL) [24] and Time-Frequency Logic (TFL) [13] allow the mapping from complete streams to one final verdict/quantity. They cannot generally be evaluated in a linear way. The idea used in TeSSLa of supporting signals and event streams has also been used for Timed Regular Expressions [4], but those have two explicitly different stream types, where TeSSLa internally represents signals as event streams. Recently, synthesis of hardware-based monitors from stream specifications has become an important field: For LOLA [10] constant memory bounds for an algorithm that evaluates well-formed specifications exist and for LOLA 2.0 [15] future references must be eliminated to gain constant memory bounds. There has been work on synthesis of STL to FPGAs in different ways as well [21,26].

2 Formal Definition of the TeSSLa Core Language

In this section we introduce syntax and semantics of the minimal core of TeSSLa. In examples we use parametrized definitions, e.g. $\mathbf{merge}(x, y) := \ldots$ on top, which are expanded to their definitions until only core operators remain.

Preliminaries. Given a partial order $(A, \leq)$, a set $D \subseteq A$ is called *directed* if $\forall a, b \in D : a \leq b \vee b \leq a$. $(A, \leq)$ is called *directed-complete partial order (dcpo)* if there exists a supremum $\bigvee D$ for every directed subset $D \subseteq A$. Let $f \in A \to B$ be a function and $(A, \leq)$, $(B, \leq')$ partial orders. f is called *monotonic* if it preserves the order, i.e. $\forall a_1, a_2 \in A : a_1 \leq a_2 \Rightarrow f(a_1) \leq' f(a_2)$. f is called *continuous* if it preserves the supremum, i.e. $\bigvee f(D) = f(\bigvee' D)$ for all directed subsets $D \subseteq A$. By the Kleene fixed-point theorem, every monotonic and continuous function $f : A \to A$ has a least fixed point μf if $(A, \leq)$ is a dcpo with a least element $\perp$. μf is the least upper bound of the chain iterating f starting with the bottom element: $\mu f = \bigvee \{f^n(\perp) \mid n \in \mathbb{N}\}$.

Syntax. A TeSSLa specification φ consists of a set of possibly mutually recursive stream definitions defined over a finite set of variables $\mathbb{V}$ where an equation has the form $x := e$ with $x \in \mathbb{V}$ and

$$e ::= \mathbf{nil} \mid \mathbf{unit} \mid x \mid \mathbf{lift}(f)(e, \ldots, e) \mid \mathbf{time}(e) \mid \mathbf{last}(e, e) \mid \mathbf{delay}(e, e).$$

All variables not occuring on the left-hand side of equations are *input variables*. All variables on the left-hand side are *output variables*. We call a TeSSLa specification *flat* if it does not contain any nested expressions. Every specification can be represented as a flat specification by using additional variables and equations.

Semantics. We define the semantics of TeSSLa in terms of an abstract time domain which only requires a total order and corresponding arithmetic operators:

Definition 1. *A* time domain *is a totally ordered semi-ring* $(\mathbb{T}, 0, 1, +, \cdot, \leq)$ *that is not negative, i.e.* $\forall_{t \in \mathbb{T}}\, 0 \leq t$.

We extend the order on time domains to the set $\mathbb{T}_\infty = \mathbb{T} \cup \{\infty\}$ with $\forall_{t\in\mathbb{T}}\, t < \infty$.

Conceptually, streams are timed words that are known inclusively or exclusively up to a certain timestamp, its progress, that might be infinite. A stream might contain an infinite number of events even if its progress is finite.

Definition 2. *An* event stream *over a time domain* $\mathbb{T}$ *and a data domain* $\mathbb{D}$ *is a finite or infinite sequence* $s = a_0 a_1 \cdots \in \mathcal{S}_\mathbb{D} = (\mathbb{T} \cdot \mathbb{D})^\omega \cup (\mathbb{T} \cdot \mathbb{D})^+ \cup (\mathbb{T} \cdot \mathbb{D})^* \cdot (\mathbb{T}_\infty \mathbb{T} \cdot \{\bot\})$ *where* $a_{2i} < a_{2(i+1)}$ *for all* i *with* $0 < 2(i+1) < |s|$ *(*$|s|$ *is* ∞ *for infinite streams). The* prefix relation *over* $\mathcal{S}_\mathbb{D}$ *is the least relation that satisfies* $s \sqsubseteq s$, $u \sqsubseteq s$ *if* $uv \sqsubseteq s$ *and* $ut'\bot \sqsubseteq s$ *if* $ut \sqsubseteq s$, $t' < t, t \in \mathbb{T}_\infty$ *and* $t' \in \mathbb{T}$.

We say a stream has an event with value d at time t if in its sequence d directly follows t. We say a stream is known at time t if it contains a strictly larger timestamp or a non-strictly larger timestamp followed by a data value or $\bot$. Where convenient, we also see streams as functions $s \in \mathbb{T} \to \mathbb{D} \cup \{\bot, ?\}$ such that $s(t) = d$ if the stream has value d at time t, $s(t) = \bot$ if it is known to have no value, and $s(t) = ?$ otherwise. We refer to the supremum of all known timestamps of a stream as inclusive or exclusive progress, depending on whether it is itself a known timestamp. The prefix relation realises the intuition of cutting a stream at a certain point in time while keeping or removing the cutting point.

In the following, we present the denotation of a specification φ as a function between input streams and output streams.

Definition 3 (TeSSLa semantics). *Given a specification* φ *of equations* $y_i := e_i$, *every* e_i *can be interpreted as a function of input streams* $s_1, \ldots, s_k$ *and output streams* $s'_1, \ldots, s'_n$, *that is composed of the primitive functions whose denotation is given in the rest of this section. Input variables are mapped to input streams,* $[\![x_i]\!]_{s_1,\ldots,s_k,s'_1,\ldots,s'_n} = s_i$ *and output variables to output streams,* $[\![y_i]\!]_{s_1,\ldots,s_k,s'_1,\ldots,s'_n} = s'_i$. *Thus for fixed input streams* $s_1, \ldots, s_k$ *and every* e_i, *we obtain a function* $[\![e_i]\!]_{s_1,\ldots,s_k} \in \mathcal{S}_{\mathbb{D}'_1} \times \ldots \times \mathcal{S}_{\mathbb{D}'_n} \to \mathcal{S}_{\mathbb{D}'_i}$ *and in combination a function* $[\![e_1, \ldots, e_n]\!]_{s_1,\ldots,s_k} \in \mathcal{S}_{\mathbb{D}'_1} \times \ldots \times \mathcal{S}_{\mathbb{D}'_n} \to \mathcal{S}_{\mathbb{D}'_1} \times \ldots \times \mathcal{S}_{\mathbb{D}'_n}$. *We now define the denotation of a specification* φ *as the least fixed-point of this function.*

$$[\![\varphi]\!] \in \mathcal{S}_{\mathbb{D}_1} \times \ldots \times \mathcal{S}_{\mathbb{D}_k} \to \mathcal{S}_{\mathbb{D}'_1} \times \ldots \times \mathcal{S}_{\mathbb{D}'_n}$$
$$[\![\varphi]\!](s_1, \ldots, s_k) = \mu\left([\![e_1, \ldots, e_n]\!]_{s_1,\ldots,s_k}\right)$$

The function $[\![e_1, \ldots, e_n]\!]_{s_1,\ldots,s_k}$ is monotonic and continuous because all primitive TeSSLa functions defined later in this section are monotonic and continuous and both properties are closed under function composition and cartesian products. $(\mathcal{S}_\mathbb{D}, \sqsubseteq)$ and by extension $(\mathcal{S}_{\mathbb{D}_1} \times \ldots \times \mathcal{S}_{\mathbb{D}_n}, \sqsubseteq \times \ldots \sqsubseteq)$ are dcpos.

By the Kleene fixed-point theorem $[\![e_1, \dots, e_n]\!]_{s_1,\dots,s_k}$ has a least fixed point, which is the least upper bound of its Kleene chain.

Next we give the semantics of the primitive TeSSLa functions. The dependency of the input and output streams $s_1, \dots, s_k, s'_1, ..., s'_n$ is assumed implicitly.

Definition 4. Nil *is a constant for the completely known stream without any events:* $[\![\textbf{unit}]\!] = \infty \in \mathcal{S}_{\mathbb{D}}$.

We use the unit type $\mathbb{U} = \{\square\}$ for streams that can carry only the single value $\square$.

Definition 5. Unit *is a constant for the completely known stream with a single unit event at timestamp zero:* $[\![\textbf{unit}]\!] = 0\,\square\,\infty \in \mathcal{S}_{\mathbb{U}}$

The following functions are given by specifying two conditions: the first for positions where an output event occurs, and the second where no output event occurs. Thereby the progress of the stream is defined indirectly as the position where the output can no longer be inferred from these conditions.

Definition 6. *The* time *operator returns the stream of the timestamps of another stream* $[\![\textbf{time}(e)]\!] = \mathsf{time}([\![e]\!])$ *where* $\mathsf{time} \in \mathcal{S}_{\mathbb{D}} \to \mathcal{S}_{\mathbb{T}}$ *is defined as* $\mathsf{time}(s) = s'$ *such that*

$$\forall_t s'(t) = t \Leftrightarrow s(t) \in \mathbb{D} \qquad \forall_t s'(t) = \bot \Leftrightarrow s(t) = \bot.$$

The lift operator lifts an n-ary function f from values to streams. The notation $A_1 \times \ldots \times A_n \rightarrowtail B$ denotes the set of functions where all A_i and B have been extended by the value $\bot$.

Definition 7. *Unary* lift *is defined as* $[\![\textbf{lift}(f)(e)]\!] = \mathsf{lift}_1(f)([\![e]\!])$ *where* $\mathsf{lift}_1 \in (\mathbb{D} \rightarrowtail \mathbb{D}') \to (\mathcal{S}_{\mathbb{D}} \to \mathcal{S}_{\mathbb{D}'})$ *is given by* $\mathsf{lift}_1(f)(s) = s'$ *such that*

$$\forall_{t,d\in\mathbb{D}'} s'(t) = d \Leftrightarrow s(t) \in \mathbb{D} \wedge f(s(t)) = d$$
$$\forall_t s'(t) = \bot \Leftrightarrow s(t) = \bot \vee s(t) \in f(s(t)) = \bot.$$

Definition 8. *Binary* lift *is given as* $[\![\textbf{lift}(f)(e_1, e_2)]\!] = \mathsf{lift}_2(f)([\![e_1]\!], [\![e_2]\!])$ *where* $\mathsf{lift}_2 \in (\mathbb{D}_1 \times \mathbb{D}_2 \rightarrowtail \mathbb{D}') \to (\mathcal{S}_{\mathbb{D}_1} \times \mathcal{S}_{\mathbb{D}_2} \to \mathcal{S}_{\mathbb{D}'})$ *is given by* $\mathsf{lift}_2(f)(s, s') = s''$ *s.t.*

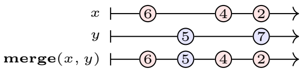
$$\forall_{t,d\in\mathbb{D}'} s''(t) = d \Leftrightarrow (s(t) \in \mathbb{D}_1 \vee s'(t) \in \mathbb{D}_2) \wedge \mathsf{known}(t) \wedge f(s(t), s'(t)) = d$$
$$\forall_t s''(t) = \bot \Leftrightarrow (s(t) = \bot \wedge s'(t) = \bot) \vee \mathsf{known}(t) \wedge f(s(t), s'(t)) = \bot$$

where $\mathsf{known}(t) := s(t) \neq ? \wedge s'(t) \neq ?$.

The binary lift can naturally be extended to an n-ary lift by recursively combining two streams into a stream of tuples or partially applied functions until the final result is obtained. Alternatively, the scheme of the binary lift can be easily extended to higher arities.

Example 1. Merge combines events of two streams, prioritising the first one.

$$\mathbf{merge}(x,y) := \mathbf{lift}(\mathsf{mergeaux})(x,y)$$
$$\mathsf{mergeaux}(a \neq \bot, b) := a$$
$$\mathsf{mergeaux}(\bot, b) := b$$



Example 2. Const maps the values of all events of the input stream to a constant value: $\mathbf{const}(c)(a) := \mathbf{lift}(\mathsf{constaux}(c))(a)$ with $\mathsf{constaux}(c)(a) := c$. Using **const** we can lift constants into streams representing a constant signal with this value, e.g. $\mathbf{true} := \mathbf{const}(\text{true})(\mathbf{unit})$ or $\mathbf{zero} := \mathbf{const}(0)(\mathbf{unit})$.

Definition 9. *The* last *operator takes two streams and returns the previous value of the first stream at the timestamps of the second. It is defined as* $[\![\mathbf{last}(e_1,e_2)]\!] = \mathsf{last}([\![e_1]\!],[\![e_2]\!])$ *where* $\mathsf{last}_{\mathbb{D},\mathbb{D}'} \in \mathcal{S}_{\mathbb{D}} \times \mathcal{S}_{\mathbb{D}'} \to \mathcal{S}_{\mathbb{D}}$ *is given as* $\mathsf{last}(s,s') = s''$ *such that*

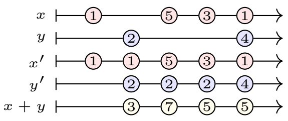
$$\forall_{t,d\in\mathbb{D}} s''(t) = d \;\Leftrightarrow s'(t) \in \mathbb{D}' \wedge \exists_{t'<t} s(t') = d \wedge \mathsf{noData}(t',t)$$
$$\forall_t s''(t) = \bot \Leftrightarrow s'(t) = \bot \wedge \mathsf{defined}(t) \vee \forall_{t'<t} s(t') = \bot$$

where $\mathsf{noData}(t,t') := \forall_{t''|t<t''<t'} s(t'') = \bot$ *and* $\mathsf{defined}(t) := \forall_{t'<t} s''(t') \neq ?$.

Note that while TeSSLa is defined on event streams, **last** realizes some essential aspects of the signal semantics: With this operator one can query the last known value of an event stream at a specific time and hence interpret the events on this stream as points where a piece-wise constant signal changes its value.

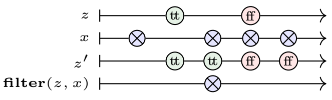
Example 3. By combining the **last** and the **lift** operators, we can now realize the *signal lift* semantics implicitly used in the introduction:
$\mathbf{slift}(f)(x,y) := \mathbf{lift}(\mathsf{sliftaux}(f))(x',y')$ with

$$x' := \mathbf{merge}(x, \mathbf{last}(x,y)) \text{ and}$$
$$y' := \mathbf{merge}(y, \mathbf{last}(y,x)).$$
$$\mathsf{sliftaux}(f)(a \neq \bot, b \neq \bot) := f(a,b)$$
$$\mathsf{sliftaux}(f)(\bot, b) := \bot$$
$$\mathsf{sliftaux}(f)(a, \bot) := \bot$$



Example 4. In order to *filter* an event stream with a dynamic condition, we apply the last known filter condition to the current event:
$\mathbf{filter}(z,x) := \mathbf{lift}(\mathsf{filteraux})(\mathbf{merge}(z, \mathbf{last}(z,x)), x)$

$$\mathsf{filteraux} : \mathbb{B} \times A \rightarrowtail A$$
$$\mathsf{filteraux}(c \neq \text{true}, a) = \bot$$
$$\mathsf{filteraux}(\text{true}, a) = a$$



Definition 10. *The* delay *operator takes delays as its first argument. After a delay has passed, a unit event is emitted. A delay can only be set if a reset event is received via the second argument, or if an event is emitted on the output.*

Formally, $[\![\mathbf{delay}(e_1, e_2)]\!] = \mathsf{delay}([\![e_1]\!], [\![e_2]\!])$ *where* $\mathsf{delay}_{\mathbb{D}} \in \mathcal{S}_{\mathbb{T}\setminus\{0\}} \times \mathcal{S}_{\mathbb{D}} \to \mathcal{S}_{\mathbb{U}}$ *is given as* $\mathsf{delay}(s, s') = s''$ *such that*

$$\begin{aligned}
&\forall_t s''(t) = \Box \Leftrightarrow \exists_{t'<t} s(t') = t - t' \wedge \mathsf{setable}(t') \wedge \mathsf{noreset}(t', t)\\
&\forall_t s''(t) = \bot \Leftrightarrow\\
&\qquad \mathsf{defined}(t) \wedge \forall_{t'<t} s(t') \neq ? \wedge s(t') \neq t - t' \vee \mathsf{unsetable}(t') \vee \mathsf{reset}(t', t)
\end{aligned}$$

where $\mathsf{setable}(t) := s''(t) = \Box \vee s'(t) \in \mathbb{D}$*,* $\mathsf{unsetable}(t) := s''(t) = \bot \wedge s'(t) = \bot$*,* $\mathsf{noreset}(t, t') := \forall_{t''|t<t''<t'} s'(t'') = \bot$ *and* $\mathsf{reset}(t, t') := \exists_{t''|t<t''<t'} s'(t'') \in \mathbb{D}$*.*

In many applications the delay operator is used in simplified versions: In the first example of the introduction that uses the delay operator, the delay and the reset argument can be the same because the delay is used only in non-recursive equations and every new delay is a reset, too. If a periodical event pattern is generated independently from input events then the second argument can be set to unit because only an initial reset event is needed. The full complexity of the delay operator is only needed if the delay is used in recursive equations with input dependencies and ensures that the fixed-point is unique.

We can observe that all basic functions are monotonic and continuous. From the fact, that these properties are closed under composition and the smallest fixed-point is determined by the Kleene chain, we can therefore conclude:

Proposition 1. *The semantics of a TeSSLa specification is monotonic and continuous in the input streams.*

In other words, the semantics will provide an extended result for an extended input and is therefore suited for online monitoring.

We can further observe that the pre-fixed-points on the Kleene chain have the following property: the progress only increases a finite number of times until a further event has to be appended. This is due to the basic functions that do handle progress in this way. We therefore obtain:

Theorem 1. *For a specification* φ *every finite prefix of* $[\![\varphi]\!](s_1, \ldots, s_k)$ *can be computed assuming all lifted functions are computable. Assuming they are computable in* $O(1)$ *steps, the prefix can be computed in* $O(k \cdot |\varphi|)$ *steps where* k *is the number of events over all involved streams.*

Note that in case the specification contains no **delay** output streams cannot contain any such timestamps that did not occur already in the inputs. Further note, that fixed-points might contain infinitely many positions with data values (in case of **delay**) and we can thus only compute prefixes. A respective monitor would exhibit infinite outputs even for finite inputs.

Due to Proposition 1 we can reuse a previously computed fixed-point if new input events occur and hence also compute the outputs incrementally.

Well-Formedness. While the least fixed-point is unique it does not have to be the only fixed-point. In that case, the least fixed-point is often the stream with progress 0 or some other stream with too little progress and one would be interested in (one of) the maximal fixed-points. Since the largest fixed-points would be more difficult to compute, especially in the setting of online monitoring, we define a fragment for which a unique fixed-point exists.

Definition 11. *We call a TeSSLa specification* φ well-formed *if every cycle of the dependency graph (of the flattened specification) contains at least one* delayed-*labelled edge. The* dependency graph *of a flat TeSSLa specification* φ *of equations* $y_i := e_i$ *is the directed multi-graph* $G = (V, E)$ *of nodes* $V = \{y_1, \ldots, y_n\}$*. For every* $y_i := e_i$ *the graph contains the edge* (y_i, y_j) *iff* y_j *is used in* e_i*. We label edges corresponding to the first argument of* **last** *or* **delay** *with* delayed*.*

Theorem 2. *Given a well-formed specification* φ *of equations* $y_i := e_i$ *and input streams* $s_1, \ldots, s_k$ *then* $\mu(\llbracket e_1, \ldots, e_n \rrbracket_{s_1,\ldots,s_k})$ *is the only fixed-point.*

Proof. From the Kleene fixed-point theorem we know $\mu\left(\llbracket e_1, \ldots, e_n \rrbracket_{s_1,\ldots,s_k}\right) = \bigsqcup\{\llbracket e_1, \ldots, e_n \rrbracket^n_{s_1,\ldots,s_k}(\bot) \mid n \in \mathbb{N}\}$. Because φ is well-formed, every $\llbracket e_i \rrbracket_{s_1,\ldots,s_k}$ is either constant or contains at least one **last** or **delay**. The input streams $s_1, \ldots, s_k$ limit progress, i.e. the maximal timestamp produced, of $\llbracket e_i \rrbracket_{s_1,\ldots,s_k}$. The progress strictly increases with every step of the iteration of $\llbracket e_1, \ldots, e_n \rrbracket_{s_1,\ldots,s_k}$ in the Kleene chain until the limit given by the input streams is reached. Every other fixed-point of $\llbracket e_1, \ldots, e_n \rrbracket_{s_1,\ldots,s_k}$ must be an extension of the least fixed-point, but the least fixed-point has already the maximal progress permitted by the input streams. $\square$

3 Expressiveness of TeSSLa

We discuss the expressiveness of four different TeSSLa fragments: TeSSLa specifications without the delay operator can only produce events with timestamps which are already included in the input streams and TeSSLa specifications with the delay operator can produce arbitrary event patterns even without any input event. On the other hand we distinguish between TeSSLa specifications which use only bounded data structures, which can only consider finitely many past events, and those with unbounded data structures which can consider infinitely many past events in the computation of new events. For an overview of the different TeSSLa fragments see Fig. 1 at the end of the next section.

To characterize functions which can be expressed in TeSSLa we define *timestamp conservatism* and *future independence* in addition to monotonicity and continuity. For a stream $a \in \mathcal{S}_{\mathbb{D}}$ we denote with $T(a)$ the set of timestamps present in the stream a and for multiple streams $T(a_1, \ldots, a_n) := \bigcup_{1 \le i \le n} T(a_i)$.

Definition 12 (Timestamp Conservatism). *We call a function* $f \in \mathcal{S}_{\mathbb{D}_1} \times \ldots \times \mathcal{S}_{\mathbb{D}_k} \to \mathcal{S}_{\mathbb{D}'_1} \times \ldots \times \mathcal{S}_{\mathbb{D}'_n}$ *on streams* timestamp conservative *iff it does not introduce new timestamps, i.e. for input streams* $a \in \mathcal{S}_{\mathbb{D}_1} \times \ldots \times \mathcal{S}_{\mathbb{D}_k}$ *and output streams* $b \in \mathcal{S}_{\mathbb{D}'_1} \times \ldots \times \mathcal{S}_{\mathbb{D}'_n}$ *we have* $f(a) = b$ *implies* $T(a) \supseteq T(b)$*.*

Note that TeSSLa specifications without delay are timestamp conservative because only delay can introduce new timestamps.

For a stream $a \in \mathcal{S}_{\mathbb{D}}$ we denote with $a|_t$ the prefix of a with progress t.

Definition 13 (Future Independence). *We call a function $f \in \mathcal{S}_{\mathbb{D}_1} \times \ldots \times \mathcal{S}_{\mathbb{D}_k} \rightarrow \mathcal{S}_{\mathbb{D}'_1} \times \ldots \times \mathcal{S}_{\mathbb{D}'_n}$ on streams* future independent *iff output events only depend on current or previous events, i.e. for input streams $a \in \mathcal{S}_{\mathbb{D}_1} \times \ldots \times \mathcal{S}_{\mathbb{D}_k}$ and output streams $b \in \mathcal{S}_{\mathbb{D}'_1} \times \ldots \times \mathcal{S}_{\mathbb{D}'_n}$ we have $f(a) = b$ implies $\forall_{t \in \mathbb{T}}\ f(a_1|_t, \ldots, a_k|_t) = (b_1|_t, \ldots, b_n|_t)$.*

Note that every TeSSLa specification is future independent because the operators **last** and **delay** are the only operators referring to events with different timestamps and they refer only to previous events.

Theorem 3 (Expressiveness of TeSSLa Without Delay). *Every function $f \in \mathcal{S}_{\mathbb{D}_1} \times \ldots \times \mathcal{S}_{\mathbb{D}_k} \rightarrow \mathcal{S}_{\mathbb{D}'_1} \times \ldots \times \mathcal{S}_{\mathbb{D}'_n}$ on streams can be represented as a TeSSLa specification without delay iff it is (a) monotonic and continuous, (b) timestamp conservative and (c) future independent.*

Proof Sketch. Represent the function f as the iterative function $\tilde{f}(m, d, t) = m'$ taking a memory state m, the current input values d, and the corresponding current timestamp t and returning the new memory state m'. Output events for all output streams can be derived from m'. Because f is monotonic it is sufficient to compute the output events step by step; because f is future independent it is sufficient to allow $\tilde{f}$ to store arbitrary information about the past events; and because f is timestamp conservative it is sufficient to execute $\tilde{f}$ for every timestamp in the input events. Translate $f(x_1, \ldots, x_k) = y_1, \ldots, y_n$ into an equivalent TeSSLa specification: $t := \textbf{time}(\textbf{merge}(x_1, \ldots, x_k))$, $m := \textbf{lift}(\tilde{f})(\textbf{last}(m, t), x_1, \ldots, x_k, t)$ and $\forall_{i \leq n}\ y_i := \textbf{lift}(\tilde{o}_i)(m)$.

If all data types in the TeSSLa specification φ are bounded, $\tilde{f}$ uses a finite memory cell m, which can only store a constant number of current and previous events. Monotonicity guarantees that we can compute output events incrementally and by future independence we know that knowledge about the previous events is sufficient to derive new events. From the combination of both properties we know that it is not necessary to queue (arbitrarily large) event sequences to compute the output events. Instead one memory cell (capable of storing one element of the data domain) per delay and per last operator in the specification is sufficient. Restricting TeSSLa to bounded data types allows TeSSLa implementations on embedded systems without addressable memory because then finite memory is sufficient. Such a restricted TeSSLa specification can compute new events only based on a finite number of current and previous events.

Theorem 4 (Expressiveness of TeSSLa With Delay). *Every function $f \in \mathcal{S}_{\mathbb{D}_1} \times \ldots \times \mathcal{S}_{\mathbb{D}_k} \rightarrow \mathcal{S}_{\mathbb{D}'_1} \times \ldots \times \mathcal{S}_{\mathbb{D}'_n}$ can be represented as a TeSSLa specification with delay iff it is (a) monotonic and continuous and (b) future independent.*

The proof accompanies the step-function $\tilde{f}$ with a timeout function $\tilde{u}$ which is evaluated on every new memory state. $\tilde{u}$ returns the timestamp of the next evaluation of $\tilde{f}$, which allows arbitrary event generation. The effect of $\tilde{u}$ can be realized using the **delay** operator.

We call a stream *Zeno* if it contains two timestamps t_1 and t_2 with infinitely many events between t_1 and t_2. With the delay operator it is possible to construct such Zeno streams because the timeout function is not restricted in any way. By Rice's theorem it is impossible to check for an arbitrary timeout function whether it only generates non-Zeno timestamp sequences. Hence, one would need to restrict allowed timeout functions more drastically, which would restrict the possible event sequences generated by a TeSSLa specification further than necessary. For that reason we decided to include the capability to generate Zeno streams with TeSSLa.

As a consequence of Theorem 4 we obtain:

Corollary 1. *A TeSSLa specification with multiple delays can be translated into an equivalent specification with only one delay.*

TeSSLa with and without delay are closely related because TeSSLa without delay can verify the relation of given input/output streams with respect to a TeSSLa specification that uses delay. The delay is only needed to actively generate the events at specified times. In the following we denote with $[\![\varphi|_y]\!](x_1, \ldots, x_k) \in \mathbb{B}$ the boolean function indicating whether the boolean output stream $y \in \mathcal{S}_{\mathbb{B}}$ of the TeSSLa specification φ contains only events with value true for the input streams $x_1, \ldots, x_k \in \mathcal{S}_{\mathbb{D}_1} \times \ldots \times \mathcal{S}_{\mathbb{D}_k}$.

Theorem 5 (Delay Elimination). *For every TeSSLa specification φ with $[\![\varphi]\!] \in \mathcal{S}_{\mathbb{D}_1} \times \ldots \times \mathcal{S}_{\mathbb{D}_k} \to \mathcal{S}_{\mathbb{D}'_1} \times \ldots \times \mathcal{S}_{\mathbb{D}'_n}$ with delay operators there exists a TeSSLa specification φ' without delay operators, which derives a boolean stream $z \in \mathcal{S}_{\mathbb{B}}$, s.t. for any input streams $x_1, \ldots, x_k$ and output streams $y_1, \ldots, y_n$ we have $[\![\varphi]\!](x_1, \ldots, x_k) = y_1, \ldots, y_n$ iff $[\![\varphi'|_z]\!](x_1, \ldots, x_k, y_1, \ldots, y_n)$.*

The above theorem follows from Theorem 3 and the fact that $[\![\varphi'|_z]\!]$ is timestamp conservative, because the output stream z only contain events when any input stream contains an event.

4 TeSSLa Fragments and Transducers

In this section we investigate two TeSSLa fragments related to deterministic Büchi automata and timed automata, resp. We translate TeSSLa specifications to transducers, which can be seen as automata taking the in- and output of the corresponding transducer as input word. Thus by relating TeSSLa fragments to certain transducer classes, we inherit complexity and expressiveness results from the well-known automata models.

Boolean Fragment. The fragment TeSSLa$_{\text{bool}}$ restricts TeSSLa to boolean streams and the operators **last**, **lift** and **slift** with $\geq$ on timestamps. In the syntax expressions are restricted as follows, where f is a function $f : \mathbb{B}^n \rightarrowtail \mathbb{B}$:

$$e := \mathbf{nil} \mid \mathbf{unit} \mid x \mid \mathbf{lift}(f)(e, \ldots, e) \mid \mathbf{slift}(\geq)(\mathbf{time}(e), \mathbf{time}(e)) \mid \mathbf{last}(e, e)$$

Note that since one can only compare timestamps, for a TeSSLa$_{\text{bool}}$-formula φ and two tuples of input streams $S, S' \in \mathcal{S}_{\mathbb{D}_1} \times \ldots \mathcal{S}_{\mathbb{D}_n}$ we have $[\![\varphi]\!](S) = [\![\varphi]\!](S')$ iff all events in S' carry the same values in the same order as those in S, independent from the exact timestamps of the events.

A *deterministic finite state transducer (DFST)* is a 5-tuple $R = (\Sigma, \Gamma, Q, q_0, \delta)$ with input alphabet Σ, output alphabet Γ, state set Q, initial state $q_0 \in Q$ and transition function $\delta : Q \times \Sigma \to Q \times \Gamma$. For an input word $w = w_0 w_1 w_2 \ldots$ we call a sequence $s_0 \xrightarrow{w_0/o_0} s_1 \xrightarrow{w_1/o_1} s_2 \xrightarrow{w_2/o_2} \cdots$ a run of a DFST R with output $[\![R]\!](w) = o_0 o_1 o_2 \cdots \in \Gamma^\infty$ iff $s_0 = q_0$ and $\delta(s_i, w_i) = (s_{i+1}, o_i)$ for all $i \geq 0$. To show that TeSSLa$_{\text{bool}}$ and DFSTs have the same expressiveness, we encode DFST words as TeSSLa$_{\text{bool}}$ streams and vice versa. The function $\alpha_\Sigma(w) = S$ encodes a DFST word $w = w_0 w_1 \cdots \in \Sigma^\infty$ as a corresponding set of TeSSLa$_{\text{bool}}$ streams: For every $p \in \Sigma$ a stream $s_p \in S$ exists with $s_p = 0 d_0 1 d_1 \ldots \infty \Leftrightarrow \forall i : (d_i \Leftrightarrow w_i = p)$. The function $\beta_\Sigma(s_1, \ldots, s_k) = w = w_0 w_1 \cdots \in \Sigma^\infty$ encodes TeSSLa$_{\text{bool}}$ streams as a synchronized DFST word w over the alphabet $\Sigma = \{z_1, \ldots, z_k\} \to \text{Val}$ with $\text{Val} = \{\bot, d, <', \bot', d' \mid d \in \{\text{tt}, \text{ff}\}\}$ which maps stream names to their current values: Let $T = \{t_0 = 0, t_1, t_2, \ldots\}$ be the set of all timestamps present in the streams including 0 with $t_i < t_{i+1}$. Then $w_i(s) = <'$ if s has exclusive progress of t_i, $w_i(s) = s(t_i)'$ if s has inclusive progress of t_i or $w_i(s) = s(t_i)$ otherwise.

Theorem 6. *For a DFST $R = (\Sigma, \Gamma, Q, q_0, \delta)$ there is a TeSSLa$_{bool}$ formula φ_R and for a TeSSLa$_{bool}$ formula φ there is a DFST $R_\varphi = (\Sigma, \Gamma, Q, q_0, \delta)$ s.t.*

$$\alpha_\Gamma \circ [\![R]\!] = [\![\varphi_R]\!] \circ \alpha_\Sigma \quad \textit{and} \quad \beta_\Gamma \circ [\![\varphi]\!] = [\![R_\varphi]\!] \circ \beta_\Sigma.$$

Note that since the boolean transducers produce one output symbol per input symbol one could reattach the timestamps of the input streams to the output streams to preserve the exact timestamps, too.

Translating DFST to TeSSLa$_{bool}$. We represent the states $q \in Q \setminus \{q_0\}$ as stream which is true iff the transducer is in it: $a_q := \mathbf{merge}(x_q, \mathbf{false})$ and the initial state $a_{q_0} := \mathbf{merge}(x_{q_0}, \mathbf{true})$, where $x_{q'} := \bigvee_{(a_q, \sigma, a_{q'}, \gamma) \in \delta} d_{a_q, \sigma}$. For every transition $\eta_i = (q, \sigma, q', \gamma)$ we add $d_{q,\sigma} := \mathbf{last}(a_q, \mathbf{merge}\{s_p \mid p \in \Sigma\}) \wedge s_\sigma$ and $o_i := \mathbf{filter}(d_{q,\sigma}, \mathbf{const}(\gamma)(d_{q,\sigma}))$. The merge of all the output streams is the output: *output* $:= \mathbf{merge}\{o_i \mid \eta_i \in \delta\}$.

Translating TeSSLa$_{\text{bool}}$ *to DFST.* We translate every equation of the flattened specification φ into individual DFSTs, which are then composed into one DFST R_φ. For every DFST the input symbols are functions from the names of the input streams to Val and the output symbols are functions from the name of the equation to Val. As discussed in the previous section, for this finite data domain we only need to consider finitely many different internal states for every equation. The transition function realizes the state changes the current output based on the current state.

For the composition of the individual DFSTs every two $R = (I \to \text{Val}, O \to \text{Val}, Q, q_0, \delta)$ and $R' = (I' \to \text{Val}, O' \to \text{Val}, Q', q_0', \delta')$ are then composed parallel into $R'' = (I \cup I' \to \text{Val}, O \cup O' \to \text{Val}, Q \times Q', (q_0, q_0'), \delta'')$ with $\delta''((s_1, s_2), g'') = ((s_1', s_2'), h'') \iff \delta(s_1, g) = (s_1', h) \wedge \delta'(s_2, g') = (s_2', h') \wedge g'' = g \cup g' \wedge \forall \sigma \in I \cap I' : g(\sigma) = g'(\sigma) \wedge h'' = h \cup h'$ until one transducer $R_A = (I_A \to \text{Val}, O_A \to \text{Val}, Q_A, {q_0}_A, \delta_A)$ represents all equations. R_A contains transitions with the same in- and output values for certain propositions which represents dependencies between the original equations. We now build the closure of this transducer which roughly resembles substituting the variables and computing the fixed-point of the equations: $R_\varphi = (I_A \backslash O_A \to \text{Val}, O_A \to \text{Val}, Q_A, {q_0}_A, \delta_\varphi)$, where $\delta_\varphi(s, g) = (s', h) \iff \delta_A(s, g') = (s', h) \wedge g = g'|_{I_A \backslash O_A} \wedge (\forall a \in I_A \cap O_A : g'(a) = h(a))$ for $g|_I := g \cap (I \times \text{Val})$.

Equivalence of deterministic Büchi automata is in P and because the constructed DFSTs can be represented as those we can conclude:

Theorem 7. *Equivalence of TeSSLa$_{bool}$-formulas is in P.*

Timed Fragment. TeSSLa$_{\text{bool+c}}$ extends TeSSLa$_{\text{bool}}$ with the comparison of a timestamp with another, previous timestamp and a constant. In the syntax, expressions are restricted as follows, where $f \in \mathbb{B}^n \rightarrowtail \mathbb{B}$:

$$\begin{aligned} e := \mathbf{nil} \mid \mathbf{unit} \mid x \mid \mathbf{lift}(f)(e, \ldots, e) \mid \\ \mathbf{lift}(g_v)(\mathbf{time}(e), \mathbf{last}(\mathbf{time}(e), e)) \mid \mathbf{last}(e, e) \end{aligned}$$

Time comparison is restricted to expressions $\mathbf{lift}(g_v)(\mathbf{time}(a), \mathbf{last}(\mathbf{time}(b), a))$ for streams $a, b \in \mathcal{S}_\mathbb{B}$ and a constant $v \in \mathbb{T}$, where g_v is a function $g_v : \mathbb{T} \times \mathbb{T} \to \mathbb{B}$ of the form $g_v(t_1, t_2) = t_1 \lesseqgtr t_2 + v$ with $\lesseqgtr \in \{<, >\}$, which allows checking the temporal distance of the current events of two streams. This is directly related to how clock constraints in timed automata [1,3] work.

A *timed finite state transducer (TFST)* is a DFSTs with an additional set of clocks C and $\delta : Q \times \Sigma \times \Theta(C) \to Q \times 2^C \times \Gamma$ where $\Theta(C)$ is the set of clock constraints. A clock constraint $\vartheta \in \Theta(C)$ is defined over the grammar $\vartheta ::= \mathit{true} \mid T \leq x + c \mid T \geq x + c \mid \neg\vartheta \mid \vartheta \wedge \vartheta$, where $x \in C$, and $c \in \mathbb{T}$ is a constant and T refers to the current time. δ now also takes a clock constraint and provides a set of clocks that have to be reset to T when taking this transition. A run of a TFST extends a run of a DFST with timestamps in the input and output word. An additional clock constraint has to be fulfilled to take a transitions and when taking a transitions, some clocks are set to the current time T.

TFSTs resemble timed automata using the notion of clock constraints from [3]. A TFST is called *deterministic*, or DTFST, iff for any two different transitions $\eta_1, \eta_2 \in \delta$ their conjuncted clock constraints $\vartheta_{\eta_1} \wedge \vartheta_{\eta_2}$ are unsatisfiable.

To show that TeSSLa$_{\text{bool+c}}$ and DTFSTs have the same expressiveness, we again encode words as streams and vice versa, but this time α_Σ and β_Σ preserve the timestamps. Hence both representations are now isomorphic and we can use the inverse encoding functions for decoding:

Theorem 8. *For a DTFST $R = (\Sigma, \Gamma, Q, q_0, C, \delta)$ a TeSSLa$_{bool+c}$ formula φ_R exists and for a TeSSLa$_{bool+c}$ formula φ a DTFST $R_\varphi = (\Sigma, \Gamma, Q, q_0, C, \delta)$ exists:*

$$[\![R]\!] = \alpha_\Gamma^{-1} \circ [\![\varphi_R]\!] \circ \alpha_\Sigma \quad \textit{and} \quad [\![\varphi]\!] = \beta_\Gamma^{-1} \circ R_\varphi \circ \beta_\Sigma.$$

Translating DTFST to TeSSLa$_{\text{bool+c}}$. We reuse the translation for DFSTs with the following adjustments: We extend the stream $d_{q,\sigma}$ to $d_{q,\sigma,\vartheta}$ by adding the timing constraint ϑ, which is translated by lifting the boolean combination to signal semantics and translating the constraint $T \lesseqgtr x + c$ to $\textbf{time}(\textbf{merge}\{s_p \mid p \in \Sigma\}) \lesseqgtr \textbf{last}(\textbf{time}(\textbf{merge}(b_x, \textbf{unit})), \textbf{merge}\{s_p \mid p \in \Sigma\}) + c$. Also for every clock $x \in C$ we add $b_x := \textbf{merge}\big\{\textbf{filter}(d_{q,\sigma,\vartheta}, d_{q,\sigma,\vartheta}) \mid (q, \sigma, \vartheta, q', r, \gamma) \in \delta \wedge x \in r\big\}$.

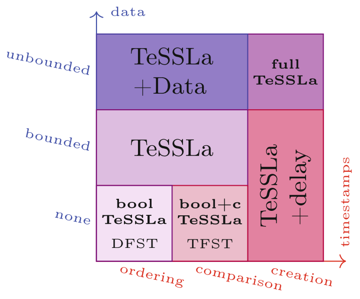


Fig. 1. TeSSLa fragments are restricted regarding (a) event values and available *data* structures and (b) event *timestamps* and how events sequences are recognized and generated: TeSSLa$_{\text{bool}}$ only checks event *ordering* like deterministic Büchi automata and BSRV [7] (LOLA restricted to boolean streams). TeSSLa$_{\text{bool+c}}$ additionally has timestamp *comparison* with constants like deterministic timed automata. TeSSLa has arbitrary *bounded* data structures and arbitrary computations on the timestamps. Full TeSSLa allows *unbounded* data structures and the creation of new timestamps via **delay**.

Translating TeSSLa$_{\text{bool+c}}$ *to DTFST.* The transducers from the equations in φ are build as before, but instead of translating equations that compare timestamps, we now translate equations of the form $\textbf{lift}(g_v)(\textbf{time}(a), \textbf{last}(\textbf{time}(b), a))$. Besides the **lift** and **last** operators, it also contains a comparison on timestamps, which is translated using the clocks and clock constraints of the DTFSTs to remember and compare timestamps. The parallel composition algorithm for DFSTs is extended by conjuncting the timing constraints of the composed transducers. Afterwards the same closure algorithm is applied. Equivalence of deterministic timed automata is PSPACE-complete [1] and because the constructed DTFSTs can be represented as those we can conclude:

Theorem 9. *Equivalence of TeSSLa$_{bool+c}$-formulas is PSPACE-complete.*

Figure 1 shows the modularity of the different TeSSLa fragments.

5 TeSSLa Implementations and Tool Support

The TeSSLa semantics presented in this paper allows multiple implementation styles: Centralized implementations using global memory which take one synchronized input word, as well as distributed implementations using message passing which take individual asynchronous input streams.

Centralized implementations are based on the same idea as the transducers: A global step function triggers the reevaluation of all TeSSLa operators involved in the specification for one timestamp, i.e. until a delayed-labelled edge in the dependency graph is reached. This step function is either triggered by new input events or a timeout of a delay if that has a smaller timestamp. Therefore every delay can register its timeouts globally s.t. the programs main loop can check with every incoming new events if the step function must be triggered for earlier delays before handling the external input. This implementation form is well-suited for software implementations running on traditional CPUs because it minimizes the internal communication overhead. Because software implementations can use dynamic memory management, the integration of unbounded data structures is straightforward.

As motivated in the introduction, one goal of TeSSLa's design is to allow distributed, parallel implementations with finite memory, e.g. on embedded systems or FPGAs. In this scenario we neither have dynamic memory management nor can we implement a global step function. Instead, every operator in the dependency graph is translated into a computation node with a fixed-size memory cell and finite input queues storing incoming events for every dependency. This setup has already been discussed for a preliminary non-recursive version of TeSSLa in [23]. The streams used in the TeSSLa semantics presented in this paper have an explicit notion of progress, which allows the local composition of TeSSLa operators without a global synchronization. Hence every computation node can produce a new output value if at least one input queue contains a new event and all other input queues contain at least progress until the timestamp of this event. The output value is sent to the input queues of all nodes depending on this node. While recursive equations in the transducers are solved by building the closure of the transducer created by applying the parallel composition to all computation nodes, in this message passing scenario we actually implement the Kleene chain of the fixed-point defined in the TeSSLa semantics in Definition 3: Progress and values are circulated in the cyclic graph of computation nodes until the progress increases no longer, which is exactly when the fixed point is reached. Since every computation node only produces new output events if there is enough progress on every input queue, we can guarantee that the fixed point is computed before new external events are processed.

For practical evaluations of TeSSLa we implemented a TeSSLa compiler in Scala which parses the TeSSLa specification, performs static type checking and converts the specification to flat TeSSLa. Additionally we added a macro system to be able to specify more complex functions based on the basic TeSSLa operators. The macro system allows to build application-domain-specific standard libraries, which makes TeSSLa a very flexible and powerful but still convenient and easy-to-learn specification language.

Furthermore, the types of the input streams are declared explicitly and the user can specify which streams should be contained in the output. Using the macro system, implicit application of **slift** to functions and implicit conversion from constants to constant signals, we can write the event counting example from the introduction as follows:

```
def count[A](a: Events[A]) := {                          in x: Events[Unit]
  def c: Events[Int] := merge(last(c, a) + 1, 0)         def y := count(x)
  c }                                                     out y
```

We combined the compiler with an interpreter written in Scala, which allows the usage of Java data structures. In order to apply TeSSLa for runtime verification we instrument the LLVM byte code of C programs and analyse this trace online with TeSSLa. This tool chain is available as a Docker container and a web IDE[2].

6 Conclusion

In this paper we presented the real-time specification language TeSSLa which operates on independent, timed streams and proved that it is suitable for online monitoring. We characterized the expressiveness of TeSSLa in terms of certain classes of stream-transforming functions. We also proved the equivalence of a boolean and a timed fragment of TeSSLa to respective classes of transducers and thereby obtained that equivalence for those fragments is in P and PSPACE, resp. These results facilitate advanced optimizations and static analyses of specifications, e.g. whether such a specification can generate certain outputs. We presented an implementation based on infinite-state transducers and sketched how TeSSLa is also suitable for parallelized implementations.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. TCS **126**(2), 183–235 (1994)
2. Alur, R., Fisman, D., Raghothaman, M.: Regular programming for quantitative properties of data streams. In: Thiemann, P. (ed.) ESOP 2016. LNCS, vol. 9632, pp. 15–40. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49498-1_2
3. Alur, R., Henzinger, T.A.: Back to the future: towards a theory of timed regular languages. In: IEEE FOCS, pp. 177–186 (1992)
4. Asarin, E., Caspi, P., Maler, O.: Timed regular expressions. J. ACM **49**(2), 172–206 (2002)

[2] http://www.tessla.io.

5. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM TOSEM **20**(4), 14 (2011)
6. Berry, G.: The foundations of Esterel. In: Plotkin, G., Stirling, C., Tofte, M. (eds.) Proof, Language, and Interaction: Essays in Honour of Robin Milner, pp. 425–454. MIT Press, Cambridge (2000)
7. Bozzelli, L., Sánchez, C.: Foundations of boolean stream runtime verification. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 64–79. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_6
8. Broy, M., Stølen, K.: Specification and Development of Interactive Systems - Focus on Streams, Interfaces, and Refinement. Springer, New York (2001). https://doi.org/10.1007/978-1-4613-0091-5
9. Convent, L., Hungerecker, S., Leucker, M., Scheffel, T., Schmitz, M., Thoma, D.: TeSSLa: temporal stream-based specification language. arXiv:1808.10717, August 2018
10. D'Angelo, B., et al.: LOLA: runtime monitoring of synchronous systems. In: TIME, pp. 166–174. IEEE (2005)
11. Decker, N., et al.: Online analysis of debug trace data for embedded systems. In: DATE. IEEE (2018)
12. Decker, N., et al.: Rapidly adjustable non-intrusive online monitoring for multi-core systems. In: Cavalheiro, S., Fiadeiro, J. (eds.) SBMF 2017. LNCS, vol. 10623, pp. 179–196. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70848-5_12
13. Donzé, A., Maler, O., Bartocci, E., Nickovic, D., Grosu, R., Smolka, S.A.: On temporal logic and signal processing. ATVA **7561**, 92–106 (2012)
14. Eliot, C., Hudak, P.: Functional reactive animation. In: ICFP, pp. 163–173 (1997)
15. Faymonville, P., Finkbeiner, B., Schirmer, S., Torfah, H.: A stream-based specification language for network monitoring. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 152–168. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_10
16. Faymonville, P., Finkbeiner, B., Schwenger, M., Torfah, H.: Real-time stream-based monitoring. arXiv:1711.03829, November 2017
17. Gautier, T., Le Guernic, P., Besnard, L.: SIGNAL: a declarative language for synchronous programming of real-time systems. In: Kahn, G. (ed.) FPCA 1987. LNCS, vol. 274, pp. 257–277. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-18317-5_15
18. Halbwachs, N., Caspi, P., Pilaud, D., Plaice, J.: LUSTRE: a declarative language for programming synchronous systems. In: POPL, pp. 178–188. ACM Press (1987)
19. Havelund, K., Goldberg, A.: Verify your runs. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 374–383. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69149-5_40
20. Havelund, K., Roşu, G.: Synthesizing monitors for safety properties. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 342–356. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46002-0_24
21. Jaksic, S., Bartocci, E., Grosu, R., Kloibhofer, R., Nguyen, T., Nickovic, D.: From signal temporal logic to FPGA monitors. In: MEMOCODE, pp. 218–227 (2015)
22. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Logic Algebr. Progr. **78**(5), 293–303 (2009)
23. Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M., Schramm, A.: TeSSLa: runtime verification of non-synchronized real-time streams. In: SAC. ACM (2018)

24. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS/FTRTFT -2004. LNCS, vol. 3253, pp. 152–166. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30206-3_12
25. Pike, L., Goodloe, A., Morisset, R., Niller, S.: Copilot: a hard real-time runtime monitor. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 345–359. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16612-9_26
26. Selyunin, K., et al.: Runtime monitoring with recovery of the SENT communication protocol. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 336–355. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_17

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.