Contents

1 Related Work

Just a collection of thoughts for now, needs to be polished a lot

As Runtime monitoring and verification is a widely researched field, multiple approaches to attain it's goals were developed.

As stated in [3] most approaches are geared towards software written in Java, while many critical systems are written in C and there are countless other systems that could benefit from monitoring and verification written in all kinds of programming languages. With TeSSLa as a specification Language over Streams, which has no assumptions on the environment of the system that produces the Streams, as the base for our monitoring approach, we recognized the possibility to abstract the monitoring plattform from the monitored program. This means that the developed runtime for TeSSLa specifications is not restricted to monitor programs written in a specific language but can monitor anything that can produce Streams.

To show that the runtime is valuable in the context of existing approaches, we will show ways to generate traces from systems that were used to evaluate other monitoring techniques. Afterwards we will compare the expressiveness of TeSSLa and the runtime with other approaches, based on the generated traces, to show what kinds of specifications can be monitored with TeSSLa and where the Language or the runtime can be extended.

The following Chapter will highlight the systems against which TeSSLa and the runtime is evaluated, furthermore it will also give insights into other work that TeSSLa and this thesis is based on.

1.1 TeSSLa

The implemented Runtime and the theoretic work of this thesis is built upon the TeSSLa project in [2]. For the project the syntax and a formal semantic of a specification language was defined.

Specifications in TeSSLa are based on streams of data. Streams are the representation of things that change over time, e.g. a variable in a program. To model streams, TeSSLa defines a timing model. That model is based on timestamps that are isomorphic to real numbers $\mathbb{R}$. Figure 1.1 shows how streams behave over time.

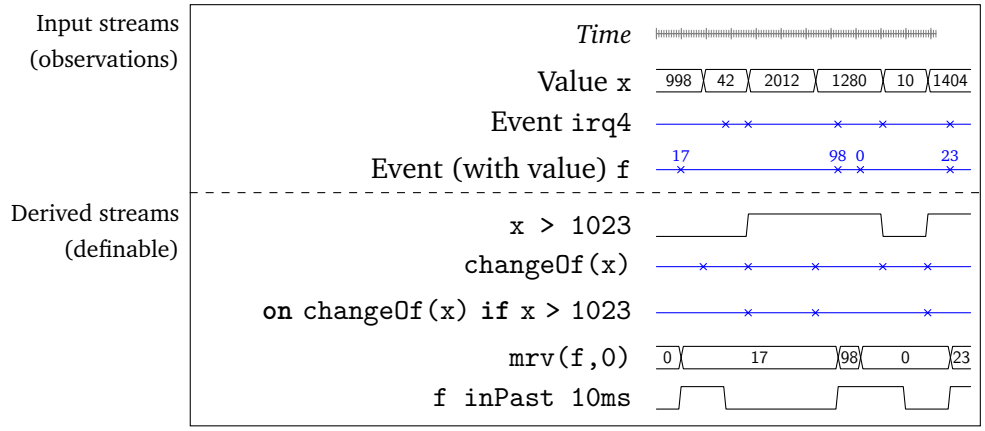

Fig. 1.1: Visualization of TeSSLas Stream model, taken from [2]

The syntax of TeSSLa specifications is pretty small, but can be used to define complex functions and specifications:

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

1.2 LOLA

The concepts of LOLA [1] are very similar to the ones of TeSSLa. Both approaches built upon streams of events. The biggest difference in the modelation is, that while streams in Lola are based on a discrete model of time TeSSLa uses a continuous timing model.

The specification language of Lola is very small (expressions are built upon three operators) but the expressiveness surpasses Temporal Logics and many other Formalisms [1]. Expressions in Lola are built by manipulating existing streams to form new ones. Therefore streams depend on other streams, so they can be arranged in weighted dependency graph, where the weight describes the amount of steps a generated Stream is delayed compared to the parent.

Based on this graph a notion of efficiently monitorable properties is given and an algorithm to monitor them is presented.

TeSSLa takes concepts of LOLA and applies them to a continuous model of time and introduces a language and a rich set of functions that can be applied to streams. The dependency graph is a core concept of TeSSLa and is used to check if specifications are valid (e.g. cycle free) and is also the core concept to evaluate specifications over traces in this thesis.

1.3 Distributed Verification Techniques

While most implementations of RV systems don't consider or use modern ways of parallelism and distribution and focus on programs running locally, in [4] a way to monitor distributed programs is given. To do this distributed monitors, which have to communicate with one another, are implemented.

As stated earlier, the TeSSLa runtime doesn't care about the environment of the monitored program, so it doesn't distinguish between traces from distributed and non distributed programs. But the runtime itself is highly concurrent and can be distributed easily to many processors or event different computers. Therefore many of the definitions for distributed monitors can be used to reason about the behaviour of the Runtime.

1.4 Copilot

The realtime runtime monitor system Copilot was introduced in [6]. Copilot is designed to overcome the shortcomings of existing RV tools in regards to hard-realtime software written in C.

To do so they first define characteristics a monitoring approach has to fullfill to be considered valuable for this domain. The four principles are:

Functionality Monitors cannot change the functionality of the observed program unless a failure is observed.

Schedulability Monitors cannot alter the schedule of the observed program.

Certifiability Monitors must minimize the difficulty in re-validating the observed program; in particular, we make it our goal to avoid modifying the observed programs source code.

SWaP overhead Monitors must minimize the additional overhead required including size, weight, and power (SWaP).

The monitors follow a sampling based approach, where at specified steps the values of global variables are observed and the monitors are evaluated on that values. While sampling based approaches are widely disregarded in RV, because they can lead to both false positives and false negatives, they argue:

> In a hard real-time context, sampling is a suitable strategy. Under the assumption that the monitor and the observed program share a global clock and a static periodic schedule, while false positives are possible, false negatives are not. [6]

A special detail of Copilot is that monitors aren't inlined into the program but can be scheduled as independet processes. The implementation of the TeSSLa runtime in this thesis follows a similar approach: It is a totally independent program, and therefore also has some of the gains in regard to the specified four characteristics. Because the runtime works with all kinds of traces, it is insignificant how they are produced: It can work with traces based on sampling, working in a similar fashion as Copilot, or by actually instrumenting code to generate traces, which alters the semantics of the program.

1.5 RMoR

RMoR is another approach on monitoring C programs. It does so by transforming C code into an *armored* version, which includes monitors to check conformance to a specification.

Specifications are given as a textual representation of state machines. The specifications are then interweaved into the programm using CIL [5]. Specifications work on the level of function calls and state properties like *write may never be called before open was called*. Because Software Developers are often working at the same abstraction level (in contrast to e.g. assembler or machine instructions), they can define specifications without having to learn new concepts. For the TeSSLa runtime support for traces at the same abstraction level (function calls, variable reads and writes) is present and used in most of the tests in Section **??**.

Because RMoR specifications are interweaved into the program, their observations can not only be reported but also used to recover the program or even to prevent errors by calling specified functions when some condition is encountered. The TeSSLa runtime doesn't support this out of the box, as it's primary purpose is testing and offline monitoring, but in Section **??** we will look at ways to support this.

1.6 MaC

2 System

Besides the theoretical basics presented in Section 1 the TeSSLa runtime of this thesis is built upon a number of technologies. To better understand decisions made during the implementation this Chapter will give an overview of them and show why they were choosen.

As already mentioned, the implemented runtime itself is independent of the way traces are generated. Therefore we will not only look at building blocks for the Runtime but also examine related projects which can be used to obtain traces. Because the format of the traces can differ heavily, depending on how and why they were collected, they are not only used to test the runtime but also to determine how it can consume them.

2.1 TeSSLa Runtime

The Runtime to evaluate specifications is implemented in the programming language Elixir, which itself is built on top of Erlang. To understand why this plattform was choosen we will look at the Erlang ecosystem in the next section.

2.1.1 Erlang and Elixir

BEAM, Actors/Thread, multi-plat-tform (nerves project)

2.1.2 Implementation

Timing model: reason

2.2 Trace Generation

2.2.1 Debie

2.2.2 TraceBench

2.2.3 Aspect oriented programming

2.2.4 CIL

2.2.5 Google XRay

2.2.6 GCC instrument functions

2.2.7 Sampling

2.2.8 LLVM/clang AST matchers

3 Concepts

In this Chapter different ways to evaluate TeSSLa specifications are given and their equivalence is shown. To do so in Section 3.1 building blocks for evaluation approaches are defined, which are then used in later Sections to define behaviour of them and show their equivalence.

3.1 Definitions

While the TeSSLa specification itself defines a set of semantics, for this Thesis we will slightly alter some of it and add some new definitions based on them. This is necessary to reason about the specifics how the evaluation Engine is built (Note that TeSSLa doesn't define an operational semantic, therefore we will define our own) and how it behaves.

3.1.1 Time

TeSSLa has a model of continuous Time, where timestamps $t \in \mathbb{T}$ are used to represent a certain point in Time and $\mathbb{T}$ has to be isomorphic to $\mathbb{R}$.

3.1.2 Events

Events are the atomic unit of information that all computations are based on. There are three Types of Events: external, output and internal Events.

The Set of all Events is denoted as E. Each event carries a value, which can be *nothing* or a value of a Type (types are formally defined in the TeSSLa speicification but aren't important for this thesis), a timestamp and the stream it's perceived on (e.g. a function call of a specific function or the name of an output stream).

The value of an event can be queried with the function υ, its timestamp with π and its Stream with ς.

$E_e \subset E$ is the Set of all external events, their Stream corresponds to a specific trace. $E_o \subset E$ is the Set of all output events, their Stream is specified by an output

name of the TeSSLa specification. $E_n \subset E$ is the Set of all internal events. Internal events are mostly an implementation detail, which denote steps of computation inside the runtime. The Stream of internal events is implicitly given by the node that produces the stream of the Event. Note that E_e, E_o, E_n are pairwise disjoint and $E_e \cup E_o \cup E_n = E$.

3.1.3 Streams

Streams are a collection of Events with specific characteristics. While Events are the atomic unit of information, Streams represent the sequence of related Events over time.

There are two kind of Streams: Signals, which carry values at all times, and Eventstreams, which only hold values at specific times. EventStreams can be described by a sequence of Events. Signals can be described by a sequence of changes, where a change denotes that the value of a Signal changed at a specific timestamp. The only difference between a Signal and an EventStreams is that Signals always have a value while an EventStream may return $\perp$ when queried for its value at a specific time, which denotes, that no event happened at that time. Based on the similarity of Signals and Eventstreams in the following we will mainly reason about Eventstreams, but most things can also be applied to Signals.

Formally a Stream σ can be represented as a Sequence of Events $\langle e_1, \dots e_n \rangle$ where $\pi(e_i) < \pi(e_{i+1}),\ \forall i < n \in \mathbb{N}$. The Set of all Streams Σ is defined as all possible finite sequences of Events $\Sigma = \{\sigma | \sigma \in E^*\}$. An external Stream σ_e is a stream consisting only of external Events, the set of all external Streams is $\Sigma_e = \{\sigma_e | \sigma_e \in E_e^*\}$. Output and internal Streams are defined analogous. To get the Event of a Stream σ at a timestamp t it can be queried like a function: $\sigma(t) = e$ with $\pi(e) = t$. When working with Signals, the function will return the latest Event that happened at or before t while an EventStream may return $\perp$.

Furthermore Streams hold the timestamp to which they have progressed, which can be equal or greater than the timestamp of the last Event happened on them. The progress of a Stream can be obtained with $p(\sigma) = t \in T$.

Internal and output Streams can be queried for the Node that produced them with $\eta(\sigma) = n \in N$.

3.1.4 Transducers

Fundamentally TeSSLa is a special kind of a Transducer. Therefore in this Section we will define a model of Transducers which can be used to reason about the evaluation of a TeSSLa specification.

A Transducer is a System, which consumes an input and produces an output. Let Φ, Γ be two Alphabets and ϵ the empty word.

Definition 1: Transducer.
A Transducer t is a Relation $t \subseteq \Phi^* \times \Gamma^*$, Φ is called the input Alphabet, Γ the output Alphabet.

TeSSLa specifications are deterministic for any input, meaning they should produce the same output for the same input.

Definition 2: Deterministic Transducer.
A deterministic Transducer relates each input to at most one output.

Example 1: Deterministic and Nondeterministic Transducers.
$t_d = \{(a,1),(b,2),(ab,12),(ba,21)\}$ is a deterministic Transducer, $t_{nd} = \{(a,1),(a,2)\}$ is nondeterministic, because it relates a to 1 and 2.

Transducers can furthermore be categorized as synchronous, asynchronous, causal and clairvoyant Transducers: Synchronosity is a property over the behaviour of a Transducer when it's consuming input per element. If it is synchronous, it will produce an output element for each input element.

Definition 3: Synchronous Transducer.
Let $\vec{\imath} \in \Phi^*, i \in \Phi, \vec{o} \in \Gamma^*, o \in \Gamma$. A Transducer t is called synchronous, when it satisfies the formula: if $(\vec{\imath} \circ i, \vec{o} \circ o) \in t$ then $(\vec{\imath}, \vec{o}) \in t$

An asynchronous Transducer can produce zero, one or many outputs for each input it consumes.

Definition 4: Asynchronous Transducer.
Let $\vec{\imath} \in \Phi^*, i \in \Phi, \vec{o} \in \Gamma^*$. A Transducer t is called asynchronous when it satisfies the formula: if $(\vec{\imath} \circ i, \vec{o}) \in t$ then $\exists \vec{o'}, \vec{o''} \in \Gamma^*$ so that $\vec{o} = \vec{o'} \circ \vec{o''}$ and $(\vec{\imath}, \vec{o'}) \in t$

Example 2: Synchronous and Asynchronous Transducers.
$t_s = \{(a,1),(b,2),(ab,12),(ba,21)\}$ is a synchronous Transducer, $t_{as} = \{(a,\epsilon),(ab,12)\}$ is asynchronous.

A causal Transducer is one, where the output depends only on consumed inputs and not on future inputs:

Definition 5: Causal and Clairvoyant Transducers.
A Transducer t is called causal, when it satiesfies the condition: if $(\vec{\imath}, \vec{o}) \in t$ then $\forall \vec{\imath'} \in \Phi^*$ with $(\vec{\imath} \circ \vec{\imath'}, \vec{o'}) \in t$ it holds, that $\vec{o} \sqsubseteq \vec{o'}$

A Transducer that isn't casual is called clairvoyant.

Example 3: Causal and Clairvoyant Transducers.
$t_{cl} = \{(a,1),(b,2),(ab,12),(ba,21)\}$ *is a causal Transducer, because each output only depends on the inputs seen upto that point,* $t_{cl} = \{(a,1),(ab,22),(aa,11)\}$ *is clairvoyant, because the output when the letter* a *is seen depends on the next input.*

When talking about Transducers, it is interesting to know if two Transducers are *equivalent*. There are multiple possible definitions for equivalence of Transducers, we will look at two, which are interesting for this thesis. In the following $\sigma_{[i,j]}$ means the infix of σ which starts at position i and end at j.

Definition 6: Asynchronous equivalence of Transducers.
Let t_1, t_2 *be two asynchronous Transducers from* Φ^* *to* Γ^*. *They are called* asynchronous equivalent*, written* $t_1 \equiv_a t_2$*, if they satisfy:*
$\forall \sigma \in \Phi^*$*:*

- $\forall (\sigma_{[0,k]}, o) \in t_1$*:* $\exists k' \geq k$ *with* $(\sigma_{[0,k']}, o') \in t_2$ *and* $o \sqsubseteq o'$
- *and* $\forall (\sigma_{[0,k]}, o) \in t_2$*:* $\exists k' \geq k$ *with* $(\sigma_{[0,k']}, o') \in t_1$ *and* $o \sqsubseteq o'$

Lemma 1: Transitivity of asynchronous equivalence.
Let t_1, t_2, t_3 *be asynchronous Transducers. If* $t_1 \equiv_a t_2$ *and* $t_2 \equiv_a t_3$ *then* $t_1 \equiv_a t_3$.

Proof. Yep □

Example 4: Asynchronous equivalence of Transducers.
Let $\Phi = \{a\}, \Gamma = \{1\}$ *and*
$t_1 = \{(a,\epsilon),(aa,\epsilon),(aaa,111)\}$,
$t_2 = \{(a,1),(aa,1),(aaa,111)\}$,
$t_3 = \{(a,\epsilon),(aa,1),(aaa,11)\}$.
All three transducers are asynchronous and causal. Let's see which ones are asynchronous equivalent:

$t_1 \overset{?}{\equiv}_a t_2$

$$
\begin{aligned}
(a,\epsilon) \in t_1, k = 1 \rightarrow k' = 1, \quad & (a,1) \in t_2, \quad & \epsilon \sqsubseteq 1 \\
(aa,\epsilon) \in t_1, k = 2 \rightarrow k' = 2, \quad & (aa,1) \in t_2, \quad & \epsilon \sqsubseteq 1 \\
(aaa,111) \in t_1, k = 3 \rightarrow k' = 3, \quad & (aaa,111) \in t_2, \quad & 111 \sqsubseteq 111 \\
(a,1) \in t_2, k = 1 \rightarrow k' = 3, \quad & (aaa,111) \in t_1, \quad & 1 \sqsubseteq 111 \\
(aa,1) \in t_2, k = 2 \rightarrow k' = 3, \quad & (aaa,111) \in t_1, \quad & 1 \sqsubseteq 111 \\
(aaa,111) \in t_2, k = 3 \rightarrow k' = 3, \quad & (aaa,111) \in t_1, \quad & 111 \sqsubseteq 111
\end{aligned}
$$

$\Rightarrow t_1 \equiv_a t_2$

$t_1 \overset{?}{\equiv}_a t_3$

$$(aaa, 111) \in t_1, k = 3 \rightarrow \nexists k'$$

$\Rightarrow t_1 \not\equiv_a t_3$

Because of Lemma 1 $\Rightarrow t_2 \not\equiv_a t_3$.

3.1.5 Timed Transducers

For the second kind of equivalence we need to introduce *timed Transducers*. Let $\mathbb{T}$ be a timing model that is isomorphic to $\mathbb{R}$. For the examples we will use $\mathbb{R}$ for $\mathbb{T}$.

Definition 7: Timed Transducer.
A timed Transducer t with input Alphabet Φ and output Alphabet Γ works on timed inputs and produces timed outputs:

$$t \subseteq (\Phi \times \mathbb{T})^* \times (\Gamma \times \mathbb{T})^*$$

Example 5: Timed Transducers.
$t_{tsc} = \{((a, 0.0), (1, 0.0)), ((a, 0.0)(a, 1.0), (1, 0.0)(1, 1.0))\}$ *is a timed, causal and synchronous Transducer.* $t_{tac} = \{((a, 0.0), \epsilon), ((a, 0.0)(a, 1.0), (1, 0.0)(1, 1.0))\}$ *is a timed, causal and asynchronous Transducer.*

The Function timed:$(\Gamma \times \mathbb{T})^* \rightarrow (\Gamma \times \mathbb{T})^*$ reorders a timed sequence $\vec{o}$ by its timestamps, such that:

$$\forall i, j \in \mathbb{N} : \text{ if } i < j \text{ then } t_i < t_j \text{ with } (o_i, t_i) = \vec{o_i} \text{ and } (o_j, t_j) = \vec{o_j}$$

Example 6: Timed Function.
Let $\sigma = (a, 1.0)(a, 0.5)(a, 1.5)(a, 0.0)$. *Then is timed*$(\sigma) = (a, 0.0)(a, 0.5)(a, 1.0)(a, 1.5)$

Furthermore the Function upto: $\mathbb{T} \times (\Gamma \times \mathbb{T})^* \rightarrow (\Gamma \times \mathbb{T})^*$ removes all elements from a timed stream, that have a timestamp bigger than the first argument.

Example 7: Upto Function.
Let $\sigma = (a, 1.0)(a, 0.5)(a, 1.5)(a, 0.0)$. *Then is upto*$(1.3, \sigma) = (a, 1.0)(a, 0.5)(a, 0.0)$

For the next definitions we have to restrict timed Transducers in two ways:

Definition 8: Monotonicity of Timed Transducers.
QUESTION: Should this be part of Timed Transducer or an extra definition? A timed Transducer t with input Alphabet Φ and output Alphabet Γ is called monotonic, if $\forall(\vec{i}, \vec{o}) \in t$ it holds, that timed$(\vec{i}) = \vec{i}$

Definition 9: Boundedness of Timed Transducers.
A timed Transducer t with input Alphabet Φ and output Alphabet Γ is called bounded, if it satisfies:

$\forall \sigma \in (\Phi \times \mathbb{T})^*$*:*

if $(\sigma_{[0,k]}, \vec{o}) \in t$

then $\exists k' > k$ *with*

$(\sigma_{[0,k']}, \vec{o} \circ \vec{o'}) \in t$

and $\forall k'' > k'$ *with* $(\sigma_{[0,k'']}, \vec{o} \circ \vec{o'} \circ \vec{o''}) \in t$ *it holds, that*

$\mathit{timed}(\vec{o} \circ \vec{o'}) \sqsubseteq \mathit{timed}(\vec{o} \circ \vec{o'} \circ \vec{o''})$

Based on the *timed* and *upto* Functions and the notion of boundedness and monotonicity, timed Transducers can be compared with respect to the timestamps of their output:

Definition 10: Observational Equivalence.
Let t_1, t_2 be two bounded, monotonic timed Transducers with input Alphabet Φ and output Alphabet Γ. They are called observational equivalent*, written $t_1 \equiv_o t_2$, if they satisfy:*
$\forall \sigma \in (\Phi \times \mathbb{T})^*$*:*

$\forall (\sigma_{[0,k]}, \vec{o}) \in t_1$*:* $\exists k', k'' \geq k$ *such that*

$(\sigma_{[0,k']}, \vec{o} \circ \vec{o'}) \in t_1$

and $(\sigma_{[0,k'']}, \vec{o_2}) \in t_2$

and (work in progress)$\mathit{timed}(\mathit{upto}(\mathit{maxTimeStampIn}\ o, \vec{o} \circ \vec{o'})) = \mathit{timed}(\mathit{upto}(\mathit{maxTimeStampIn}\ o, \vec{o_2}))$

Lemma 2: Transitivity of observational Equivalence.
Let t_1, t_2, t_3 be timed Transducers. If they are bounded and $t_1 \equiv_o t_2$ and $t_2 \equiv_o t_3$ then $t_1 \equiv_o t_3$.

Proof. YEP □

Example 8: Observational Equivalence.
Let $t_1 = \{((a, 0.0), \epsilon), ((a, 0.0)(a, 1.0), (1, 1.0)), ((a, 0.0)(a, 1.0)(a, 2.0), (1, 1.0)(1, 2.0)(1, 0.0))\}$,
$t_2 = \{((a, 0.0), \epsilon), ((a, 0.0)(a, 1.0), \epsilon), ((a, 0.0)(a, 1.0)(a, 2.0), (1, 2.0)(1, 1.0)(1, 0.0))\}$,
$t_3 = \{((a, 0.0), (1, 0.0)), ((a, 0.0)(a, 1.0), a), ((a, 0.0)(a, 1.0)(a, 2.0), (1, 2.0)(1, 1.0)(1, 0.0))\}$
All three are causal, monotonic, asynchronous timed Transducers.

Let's see which ones are observational equivalent:

$t_1 \stackrel{?}{\equiv}_o t_2$

$((a, 0.0), \epsilon) \in t_1, k = 1$

$\rightarrow k' = 1, ((a, 0.0), \epsilon) \in t_2$

and timed$(\epsilon) \sqsubseteq$ *timed*(ϵ)

$((a, 0.0)(a, 1.0), (1, 1.0)) \in t_1, k = 2$

$\rightarrow k' = 3, ((a, 0.0)(a, 1.0)(a, 2.0), (1, 2.0)(1, 1.0)(1, 0.0)) \in t_2$

and timed$((1, 1.0)) = (1, 1.0) \sqsubseteq (1, 0.0)(1, 1.0)(1, 2.0)$

$\Rightarrow t_1 \equiv_a t_2$

$t_1 \stackrel{?}{\equiv}_a t_3$

$\Rightarrow t_1 \not\equiv_a t_3$

Because of Lemma 2 $\Rightarrow t_2 \not\equiv_a t_3$.

3.1.6 Labeled Timed Transducers

Maybe necessary, maybe not

3.1.7 Functions

A TeSSLa specification consists of Functions. Functions generate new Streams by applying an operation on other Streams. TeSSLa itself defines a syntax to write a specification, a set of types and a standard library of Functions, but an implementation is free to choose the functions it supports.

An example Function is $add(S_D, S_D) \rightarrow S_D$: It takes two Signals, which have to hold values of some numerical type, and produces a signal which holds values of the same type. The produced stream can either be assigned to a named identifier (think: a variable) or consumed by another function (function composition).

Functions can be divided into three categories: pure, unpure and timing. Pure Functions, also called stateless, are evaluated only on the values their inputs have at the timestamp they are evaluated, therefore they don't have to *remember* anything about earlier Events. Unpure, or stateful, Functions are evaluated over the whole input Stream, meaning they can look at all Events that happened on its inputs before the time of evaluation and also at all it's previous output Events. E.g. a Function *eventCount* has to *remember* how many events already happened on it's

input stream and increment that counter on every new event. Timing Functions are evaluated not only on the value of Events but also on their timestamp and can also manipulate it: While non timing Functions will consume events at a specific timestamp and emit an event with that timestamp, timing functions can emit Events with a different timestamp.

Timing Functions complicate the reasoning about schedules and causality and therefore aren't included in Section 3.2. In Section **??** the conclusions of earlier sections will be extended to include timing Functions.

3.1.8 Nodes

Nodes are the atomic unit of computation for the evaluation of a TeSSLa specification. A Node implements a single Function, e.g.: there is an *AddNode* which takes two input Signals and produces a new Signal. Therefore a Node is the concrete implementation of a Function in a runtime for TeSSLa specifications. The set of all Nodes is called N. The Function of a Node $n \in N$ is written as f_n.

Each Node has a set of inputs, which are either external or internal Streams, and one output, which is either an internal or an output Stream. Nodes, which have at least one external Stream as an input, are called *sources*. Nodes use a FIFO queue, provided by the Erlang plattform, to process new received Events in multiple steps:

1. Check if a new output Event can be produced (see Section 3.1.8)
2. If so, compute all timestamps, where new Events might be computed and
   a) Compute the Events, add them to the History as new Events on the output
   b) Distribute the new Events to the children of the Node
3. Else wait for another input

Determination of processable Events

Based on the asynchronous nature of Nodes, Events from different Streams can be received out of order. E.g. if a Node c is a child of Node a and b, it can receive Events from Node a at timestamps t_1, t_2, t_3, t_4 before receiving an event with timestamp t_1 from Node b. Therefore a Node can not compute it's output upto a timestamp unless it has informations from all predecessors that they did progress to that timestamp. When Node c receives the first four Events from Node a, it will only add them to it's input queues but won't compute an output. When it finally receives the first Event

from Node b it can compute all Events upto t_1. To do so it will compute *change timestamps*: The union of all timestamps where an Event occured on any input between the timestamp of the last generated output and the minimal progress of all inputs.

To see why this is necessary lets assume that Node c will receive a new Event from Node b with timestamp t_4: All inputs have progressed to t_4, but on the Stream from Node a there are changes between t_1 (where the last output was generated) and t_4, therefore the *change timestamps* are t_2, t_3, t_4 and the Node will have to compute it's output based on the values of the streams at that timestamps.

3.1.9 TeSSLa Evaluation Engine

Because Functions in TeSSLa specifications itself depend on other Functions, and these dependencies have to be circle free, the specification can be represented as a DAG@. This DAG can be directly translated into an evaluation Engine for that specification: The Nodes of the DAGs are Nodes representing the functions and the Edges are the input and output Streams between the Nodes.

Definition 11: Equivalence of evaluation Engines.
Two evaluation Engines are called equivalent, if they produce the same relationship between external and output Events for all valid sequences of external Events (where valid means ordered by their timestampt).

Definition 12: Equivalence of evaluation Engines for an input.
Two evaluation Engines are called equivalent for an input, if they produce the same output for that input.

To evaluate a specification over Traces, the evaluation Engine has process the Events that were traced. To do so the Nodes have to run their computations until no more Events are present (or the specification found an error in the trace). This leads to the question in which order Nodes should be scheduled to perform their computation. While some schedules are simply not rational (think of unfairness and causality) there are many different schedules that are feasible. It has to be prooven that a chosen schedule produces the correct conclusions for a specification, else the evaluation Engine is not valid.

In this Thesis it is shown that multiple schedules will lead to the same conclusions and therefore an implementation of an evaluation Engine is free to choose between them.

3.1.10 State and History

All TeSSLa evaluation Engines have to hold a State, which encodes information necessary to continue the evaluation, and a History, which encodes what happened on all Streams in the evaluation Engine. The State of a whole evaluation Engine is made up of the States of its Nodes.

Each Node has a State, which can hold arbitray information, e.g. a counter for a *CountNode*, and its input queues. Input queues are Fifo queues, which hold the Events produced by predecessors of the Node, that weren't consumed yet by the Node.

To distinguish between the two types of States, we will call the State of the whole Engine the *global State* and the State of a Node the *Node State*. The Set of all valid node States is called $\widetilde{N}$.

The global State of an evaluation Engine at a certain step is a map from its Nodes to their node State. We will call the Set of all global States as S. A global State can be queried like $s(n) = \widetilde{n}$ to yield the State of the Node n.

Everytime a Node n is scheduled, it changes its own node State and the node State of its direct children N_c by

- Take one Event e_i from each input queue q_i: $e_i = hd(q_i)$
- Produce new Events based on the internal information and the taken Events
- Change its children node State by appending the produced Events to the queues of all Nodes in N_c
- Update its own node State by
  - Updating the internal information if necessary
  - Replace all input queues q_i with $q_i' = tl(q_i)$

TODO History

3.1.11 Transitions

A Transition describes what happens when the evaluation Engine switches State by performing a Step: The consumption of one Event from each input of a Node and the optional generation of one or more new output Events of that Node. To look at it in another way: A Transition is the computation of a Node, therefore when we say 'Node A is scheduled' we mean that a Transition is taken which models the computation of that Node.

Formally Transitions are a Relation between two Sets of Events. E.g. the Transition $\tau = (\{e_1, e_2\}, \{e_3\})$ specifies that two Events were consumed by a Node and one Event was produced based on them. A Transition is valid, if it has at least one input Event, all of its input Events are from different Streams and all of its output Events are from the same Stream: The one from the Node which computation is modeled by the Transition. The Set of all valid Transitions is

$$T = \{(\widetilde{E}, \widetilde{E'}) | \widetilde{E} \subseteq (E_e \cup E_n) \wedge \widetilde{E} \neq \emptyset \wedge \widetilde{E'} \subseteq (E_n \cup E_o) \\ \wedge (\forall e_i, e_j \in \widetilde{E'} : \varsigma(e_i) = \varsigma(e_j)) \wedge (\forall \widetilde{e_i}, \widetilde{e_j} \in \widetilde{E} : \varsigma(\widetilde{e_i}) \neq \varsigma(\widetilde{e_j}))\}$$

where the Node which caused the Transition is $\eta(\varsigma(e_i))$. For brevity we define $\eta : T \to N$ with $\eta((e, e')) = \eta(\varsigma(e_i)),\ e_i \in e'$, which yields the Node that is modeled by the Transition.

The empty Transition, meaning no input was consumed and no output produced, is labeled with λ. Note that all Transitions, except the empty one, have to consume at least one Event (therefore no Events can be created from nowhere) and that it's possible that no Event was produced based on the consumed Events (think of a *FilterNode*). Furthermore it's theoretically possible to create multiple Events in one Transition. This makes only sense in the context of Timing Nodes, because else the generated Events would have the same timestamp, which is forbidden by the definition of Streams. With Timing Nodes one could for example implement an *EchoNode*, which duplicates an input after a specified amount of Time.

Definition 13: Application of a Transition on a State.
Given a global State s_0 and a Transition $\tau_1 = (\widetilde{E}, \widetilde{E'}) = (\{e_1, e_2, \ldots, e_i\}, \{e'_1, e'_2, \ldots, e'_i\})$, when we apply *$\tau_1$ to s_0 we get a new global State s_1 with*
$\forall \widetilde{n}_i = s_0(i)$

- *if $\eta(\widetilde{n}_i) \neq \eta(\tau_1)$ then $s_1(i) = \widetilde{n}_i$ (No Node States changes except the one from the Node of the Transition)*
- *else let $(\{\sigma_1, \ldots, \sigma_n\}, \sigma_o) = n_i$. Then is $s_1(i) = (\widetilde{\sigma}, \sigma'_o)$ with*
  - *$\sigma'_o = \sigma_o + +e'$ with $++$ defined as appending all Events in e' to σ_o ordered by their timestamp*
  - *$\widetilde{\sigma} = \{\sigma'_1, \sigma'_2, \ldots, \sigma'_n\}$ with $\sigma'_l, l \in [1, n] : \sigma_l + \widetilde{e}$ where $\widetilde{e} \in e$ and $\eta(\varsigma(\widetilde{e})) = \eta(\sigma_l)$ and $+$ is the known append operator for sequences.*

This means, that the new global State is built with the old global State by altering only the Node State of the Node identified by the Events in the second element of the Transition. The Node State is altered by appending all Events in the second Element of the Transition to the output ordered by their timestamp and by adding

each Event in the first element of the Transition to the input corresponding to the Stream of that Event.

3.1.12 Run

A Run of an evaluation Engine is a sequence of Transitions and States. The first element of the sequence is the empty Transition and the initial State of the evaluation Engine. It is a representation of the steps the Engine takes to evaluate a specification over input Streams. The length of a Run can be retrieved with $l(r) = d \in \mathbb{N}$. A Run can be queried by it's index to return the element at that index: $r(i) = (\tau_i, s_i),\ i \in [0, l(r)]$.

The Run $\langle(\lambda, s_0), (\tau_1, s_1)\rangle$ means, that the Engine was in it's initial State, took the transition τ_1 and thereby reached the state s_1.

Definition 14: Equivalence of Runs.
Two Runs are called equivalent if they have an equal State at their last position.

Because the State can be built from the Transitions that were taken, equivalence can also be defined over Transitions.

Lemma 3: Equivalence of Runs over Transitions.
If the Set of Transitions of two Runs are equal, the Runs are equivalent.

Proof. Let r_1, r_2 be the Runs of two Engines with the same Set of Transitions $\widetilde{T}$. Let s_1 be the final State of the Run r_1 and s_2 of r_2. If the two global States weren't equal, there would have to be at least one i with $s_1(i) \neq s_2(i)$, meaning the same Node has to have a different State in both Engines. Let $\widetilde{n}_1, \widetilde{n}_2$ be the Node States of one such Node in both Engines. If the two Node States are different, one of them has to contain at least one Event on an input or output that isn't on the same Stream in the other State. Let that Event be e_d. To be added to the State, there has to be a Transition $\tau = (e, e')$ with $e_d \in e \vee e_d \in e'$. This Transition has to be in $\widetilde{T}$, which means it was taken by both Engines, therefore e_d is in the History of the Node in both Runs, therefore its the two Node States are equal in both Engines. □

Definition 15: Closeness of Runs.
The Closeness δ of a Run r_1 to a Run r_2 is a pair $\delta(r_1, r_2) = (x, y)$, where x is the index before the first position where the two Runs differ and y is the number of Steps between the index of the first difference and the position where r_2 takes the Transition that r_1 took after step x. The Closeness of Runs is ordered element-wise: $(x, y) > (x', y') \leftrightarrow ((x > x') \vee (x = x' \wedge y < y))$. Therefore two Runs with length d are equal, if their Closeness is $d, 0$, which is the maximal Closeness two Runs of length d can have at all.

Example 9:.
Let

$$r_1 = \langle(\lambda, s_0), (\tau_1, s_1), (\tau_2, s_2), (\tau_3, s_3), (\tau_4, s_4), (\tau_5, s_5), (\tau_6, s_6)\rangle$$

$$r_2 = \langle(\lambda, s_0), (\tau_1, s_1), (\tau_2, s_2), (\tau_5, s_3'), (\tau_4, s_4'), (\tau_6, s_5'), (\tau_3, s_6')\rangle$$

$$r_3 = \langle(\lambda, s_0), (\tau_1, s_1), (\tau_2, s_2), (\tau_3, s_3), (\tau_5, s_4''), (\tau_4, s_5''), (\tau_6, s_6'')\rangle$$

Then is

$$\delta_{1,2} = \delta(r_1, r_2) = (3, 3),\ \delta_{1,3} = \delta(r_1, r_3) = (4, 1)$$

$$\delta_{2,1} = \delta(r_2, r_1) = (3, 2),\ \delta_{2,3} = \delta(r_2, r_3) = (3, 1)$$

$$\delta_{3,1} = \delta(r_3, r_1) = (4, 1),\ \delta_{3,2} = \delta(r_3, r_2) = (3, 2)$$

And $\delta_{1,3} < \delta_{2,3} < \delta_{2,1} < \delta_{2,1}$.

Definition 16: Enabledness of a Node.
A Node n *with the input Streams* $\widetilde{\sigma}$ *in an evaluation Engine is called* enabled *at a Step* i *of a Run* r *of that Engine, when it satisfies all of the following conditions:*

- *All consumed Events are either internal Events that were produced at an earlier step or external Events that are present:*
  - $\forall \sigma_x \in \widetilde{\sigma} \cap \Sigma_n : \exists j \in [1, i] : r(j) = (\tau_j, s_j)$ *with*
    - $(e, e') = \tau_j \wedge e_x \in e'$ *with* $\varsigma(e_x) = \sigma_x$
  - *and* $\forall \sigma_e \in \widetilde{\sigma} \cap \Sigma_e : \exists e_e$ *with* $\varsigma(e_e) = \sigma_e$
- *and weren't consumed earlier by that Node*
  - $\forall e_x$ *(the same* e_x *as in the steps above)*
    - $\nexists h \in [j, i] : (\widetilde{e}, \widetilde{e'}) = \tau_h \wedge \eta(\tau_h) = n \wedge (\tau_h, s_h) = r(h) \wedge e_x \in \widetilde{e}$

Definition 17: Valid Run.
A Run r *is called* valid *if*

- $\forall i \in [0, l(r)] : r(i) = (\tau_i, s_i) \wedge \eta(\tau_i)$ *is enabled at step* i
- *and* s_i *is built by applying* τ_i *to* s_{i-1}

Definition 18: Independence of Nodes.
A Node A is called independent *of Node B in an evaluation Engine, if it is no descendant of that Node*

Definition 19: Independence of Transitions.
A Transition τ_1 *is called independent of another Transitions* τ_2*, if* $\eta(\tau_1)$ *is independent of* $\eta(\tau_2)$.

Lemma 4: Exchange of independent Transitions.
If a Transition τ_1 is independent of a Transition τ_2, then for all Runs of the evaluation Engine that produces the Runs the following holds:

$$r_1 = \langle(\lambda, s_0), \ldots, (\tau_2, s_i), (\tau_1, s_j), \ldots, (\tau_l, s_l)\rangle \textit{is a valid Run}$$
$$\rightarrow r_2 = \langle(\lambda, s_0), \ldots, (\tau_1, s'_i), (\tau_2, s'_j), \ldots, (\tau_l, s'_l)\rangle \textit{is a valid Run}$$

Proof. Because $a = \eta(\tau_1)$ is no descendant of $b = \eta(\tau_2)$, the Stream σ with $\eta(\sigma) = \eta(\tau_2)$ can be no input of a. Therefore the enabledness of a can't be changed by τ_2 as you can see from the Definition of enabledness. So a has to be enabled before τ_2 was taken by r_1, else it couldn't be enabled afterwards. Therefore r_2 also fullfills the requirements for a valid Run. □

Lemma 5: Influence of independent Nodes.
When a Node a is independent of a Node b, then it has no influence on the enabledness of Node a if Node b is scheduled before or after it.

Proof. Follows directly from Lemma 4 □

Lemma 6: Duration of Enabledness.
A Node which is enabled stays enabled at least until it is scheduled. Formaly: If a Node n is enabled at step i in a Run r it will stay enabled at least until Step $j > j$ with $r(j) = (\tau, s), \eta(\tau) = n$. Note that it doesn't have to be enabled after step j, because there could have been multiple Events buffered on it's inputs.

Proof. TODO Idea: Two parts of enabledness: every consumed event by the transition of the Node has to be produced earlier: This doesn't change, once an event was created it will be created forever Second part: The Event wasn't consumed by that node earlier, stated otherwise: there is no Transition of that Node which contains one of the events Should be straightforward. □

3.2 Behaviour of different schedules without timing functions

For a first step we specify and compare behaviours of different approaches to evaluate TeSSLa specifications without timing Functions. Without timing Functions, all Nodes work only on values or the presence of Events. This leads to behaviours that can be easily reason about, as seen in the next sections.

All Systems to evaluate TeSSLa specifications we will look at are based on the described structure in Section 3.1.9. While there may be other approaches to evaluation, a DAG based approach seems to fit most naturaly and focusing on one structure makes comparing Systems easier.

Each evaluation Engine will work in steps, where each step is synonymous with an index in the Run of the System. Therefore at each Step one Node is scheduled to perform it's operation, represented as the Transition in the Run. The Transition will encode one of the following three Things that can happen:

- The next external Event (external Events have to be totally ordered by their timestamp) can be consumed by a source in the DAG, which generates internal Events, that are propagated to it's children.
- An internal Node, which has at least one new input buffered on all of its input queues, can perform its computation and generate a new internal Event, which is propagated to the children of that node, which therefore can compute in the next step.
- An output node, which has at least one new input buffered on all of its input queues, can produce a new output.

Evaluation Engines are free in the way they are scheduling their Nodes, only limited by causality (no Event can be consumed before it's produced). In the following evaluation Engines are classified by their scheduling approaches.

3.2.1 Synchronous Evaluation Engines

The first class of evaluation Engines are synchronous ones. They are characterized by a specific, fixed schedule. The scheduling algorithm is as follows:

- Select all Nodes that are no sources, let their count be ß
- Label them with unique natural numbers from $[1, i]$ in reversed topological order
- Label the remaining Nodes with unique natural numbers bigger than i
- Whenever a Node has to be scheduled, schedule the enabled one with the lowest label

Obviously for many DAGs there is no unique reversed topological order, therefore one can be chosen by the evaluation engine. This schedule ensures that no Node is scheduled which has a successor that can be scheduled, therefore Events are *pushed*

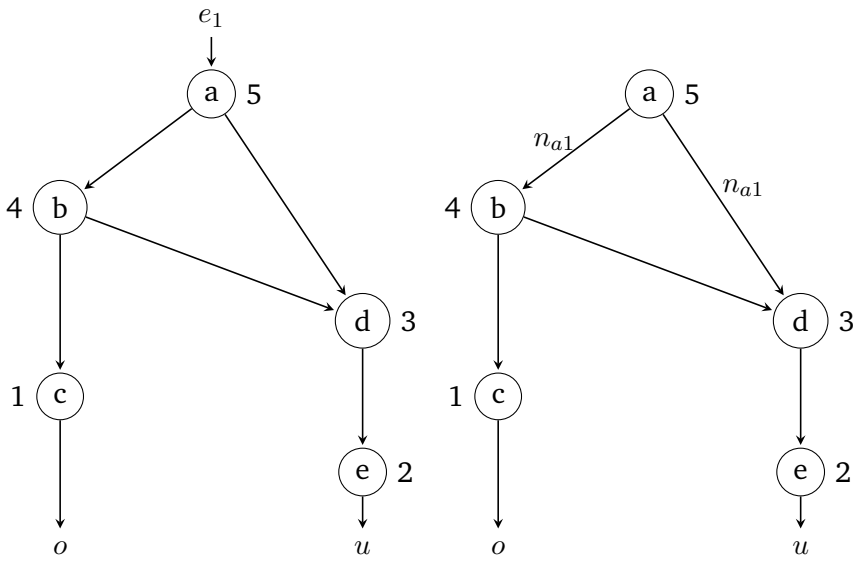

Fig. 3.1: Visualization of a simple asynchronous system with a reversed topological order.

through the DAG towards an output Node as fast as possible. As shown in Section **??** any schedule built like this is fair.

Definition 20: Valid evaluation Engines.
An evaluation Engine is called valid*, if it is equivalent to a synchronous evaluation Engine.*

Figure 3.1 visualizes a synchronous evaluation Engine. It shows two DAG representations of an evaluation Engine where the Nodes a to e are labeled in a reversed topological order and o and u represents the output Streams with that name. The left System is in its initial State and an input Event e_1 is present and can be consumed by the input Node a. When a Node is chosen to compute by the scheduler, only Node a is enabled, therefore it is scheduled. The right System is the representation of the next step: Node a has consumed the external Event and produced an internal Event n_{a1} which is propagated to all it's children: Node b and d. In the next step Node b would be scheduled, because it has the lowest number of any node that can compute (actually it's the only node that can compute at all, because d has to wait for the event from b). After B was scheduled, it would have produced the internal Event n_{b1} which would then be distributed to Nodes c and d.

The complete Run of the synchronous Engine for one Input is the following, where the States are not further defined:

$$\langle(\lambda, s_0), (\{\{n_{a1}\}, \{n_{b1}\}), s_1), (\{\{n_{b1}\}, \{o_1\}), s_2),$$
$$(\{\{n_{a1}, n_{b1}\}, \{n_{d1}\}), s_3), (\{\{n_{d1}\}, \{u_1\}), s_4)\rangle$$

If there were more then one input Event, at this point Node a would be scheduled again. It would consume the next external Event and the following Nodes would be scheduled in the same order as before, extending the Run in an obvious way.

3.2.2 Asynchronous evaluation

An asynchronous evaluation Engine is one with a fair, but not fixed schedule.

In contrast to the synchronous evaluation Engine it has no fixed schedule, the only requirement is that the schedule is fair. Therefore predecessors of enabled Nodes can perform multiple computations before their children are scheduled and Events are not *pushed* through the DAG as fast as possible.

more later

3.3 Equivalence of different schedules without timing functions

Based on the described behaviours of the approaches we now can proof the equivalence of different Schedules for the same evaluation Engine for a TeSSLa specification.

Lemma 7: Equivalence of Engines for one Input.
Two evaluation Engines are equivalent for an input, if their Runs are equivalent for that input.

Proof. Since two Runs are equal if they have the same last State, and all Events which were produced are stored in the State, during both Runs the same output Events had to be generated or else their State would differ. □

As defined by Definition 20 any evaluation Engine has to be equivalent to a synchronous one to be valid.

The Equivalence is shown in two steps: First in Section 3.3.1 it is shown, that all possible synchronous Engines for a specification are equivalent, so there is only one valid Evaluation for a Specification over a fixed Input. Afterwards in Section 3.3.2 it is shown that any asynchronous evaluation Engine is equivalent to a synchronous one.

3.3.1 Equivalence of synchronous Systems

When given a series of input Events, two synchronous evaluation Engines for a specification with different schedules will have different Runs. But both will produce all outputs that can be produced after consuming one specific input before the next Input is consumed as reasoned in Section 3.2.1. Also both Runs will obviously have the same length (both Engines are the same DAG, so they have the same number of Nodes), let that length be l.

To proof the equivalence of both Engines we can prove the equivalence of their Runs. To show the equivalence we will show that there is always another Run, which is equivalent to the second one, that has a bigger Closeness to the first one. If such a closer Run always exist, we will show that the Run with Closeness $l, 0$ to the first Run, which has to be the first Run itself, is also an equivalent Run to the second Run.

Theorem 1: Equivalence of different synchronous evaluation Engines.
Two synchronous evaluation Engines for a specification with different schedules are equivalent.

Proof. Let r_1, r_2 be the Runs of the two Engines for a given TeSSLa specification. Because each TeSSLa specification contains only a finite amount of Functions and works on finite traces, the Runs also have to be finite.

If the two Runs aren't equal, they must have a Closeness which is smaller than $(l, 0)$. Let $[r_2]$ be the Set of all Runs that are equivalent to r_2. All of those Runs will also have a Closeness from r_1 which is smaller than $(l, 0)$. Select one Run $r_2' \in [r_2]$ which has a minimal Closeness from r_1. Let $(d, k) = \delta(r_1, r_2')$.

This means that at Step d the Run r_2' has taken a different Transition than Run r_1. Let the Transitions the Runs have taken be τ_1 for r_1 and τ_2 for r_2'. Run R_2' will take Transition τ_1 at step $d + k$ (as per the definition of the Closeness). Obviously the two Transitions have to be independent of each other, else they couldn't have been taken in different order by the two Runs.

If $k > 1$ there will be a Transition $\tau_2' \neq \tau_1$ which is taken by the Run r_2' at Step $d + (k - 1)$. While this Transition τ_2' must also be taken in the first Run as per Lemma 6, it's not possible, that it was taken before τ_1, beause then the two runs wouldn't have been the same upto the point where τ_1 was taken. Therefore τ_1 has to be independent of τ_2', and because τ_2' was scheduled by the second Run before τ_1 both Transitions are independent of each other.

As of Lemma 5 which one of them is taken first can't change the enabledness of the Node of the second Transition. Therefore there is a Run r_2'', which is equal to r_2', except that the Transitions τ_1, τ_2' are scheduled the other way around. Figure 3.2

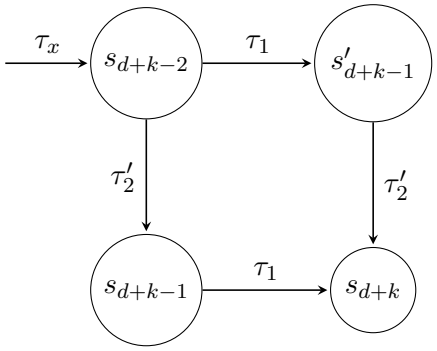

Fig. 3.2: Commutativity Diagramm of Node scheduling

visualizes how changing the order of the two Transitions can't change the global State of the Engine after both were taken. Therefore the Runs r'_2 and r''_2 are equivalent and their Closeness to r_1 is $d, k-1$, which contradicts the initial statement that r'_1 was a Run with a maximal Closeness. This means that there is an equivalent Run to r_2 which has at least the Closeness $(d, 1)$.

If $k = 1$ the Transition τ'_2 from the previous case is equal to τ_2. Based on the same reasoning there also exist a Run r''_2 which is equal to r'_2, except that the order of τ_2 and τ_1 is changed, and which is also equivalent to r'_2 and to r_2. This Run has the Closeness $(d, 0)$ to r_1. This obviously doesn't make sense: The first element of the Closeness is the last step where both Runs are equal, the second element describes how many Steps afterwards the differing Transition was taken. But if it was taken right in the step after the last equal step, there is no difference at that position, so the Closeness of r_1 and r''_2 can be at least $(d + 1, x), x \in \mathbb{N}_{>0}$. This also contradicts our initial statement that r'_2 was the Run with the biggest Closeness to r_1 which is equivalent to r_2.

Combined we can now say, that there is no upper bound on the Closeness of equivalent Runs of r_2 to r_1, therefore the Run with the Closeness $(l, 0)$ also has to be equivalent to r_2.

□

3.3.2 Equivalence of synchronous and asynchronous schedules

When the Nodes of a aren't scheduled in reversed topological order, the System can consume inputs before producing all outputs based on the last consumed input. Therefore the reordering of Runs has to be performed over wider parts of the Run.

3.4 Behaviour with Timing functions

3.5 Equalitys with Timing functions

3.6 Parallel computation

Bibliography

[1] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, et al. „LOLA: Runtime monitoring of synchronous systems". In: *Proceedings of the International Workshop on Temporal Representation and Reasoning* (2005), pp. 166–175 (cit. on p. 2).

[2] Normann Decker, Daniel Thoma, and Jannis Harder. „TESSLA A Temporal Stream-based Specification Language". 2016 (cit. on pp. 1, 2).

[3] Klaus Havelund. „Runtime Verification of C Programs". In: (2008) (cit. on p. 1).

[4] Menna Mostafa and Borzoo Bonakdarpour. „Decentralized Runtime Verification of LTL Specifications in Distributed Systems". In: *2015 IEEE International Parallel and Distributed Processing Symposium* (2015), pp. 494–503 (cit. on p. 3).

[5] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. „CIL: Intermediate language and tools for analysis and transformation of C programs". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 2304 (2002), pp. 213–228 (cit. on p. 4).

[6] Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. „Copilot: A hard real-time runtime monitor". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6418 LNCS.Rv (2010), pp. 345–359 (cit. on pp. 3, 4).

List of Figures

List of Tables

List of Theorems