Contents

1 Introduction **1**

1.1 Motivation and Problem Statement 1
1.2 Results 2
1.3 Thesis Structure 2

2 Related Work **5**

2.1 TeSSLa 5
2.2 Lola 6
2.3 Distributed Verification Techniques 7
2.4 Copilot 7
2.5 Rmor 8

3 System **11**

3.1 TeSSLa Runtime 11
3.1.1 Erlang and Elixir 11
3.1.2 Implementation 11
3.2 Trace Generation 12
3.2.1 Debie 12
3.2.2 TraceBench 12
3.2.3 Aspect oriented programming 12
3.2.4 CIL 12
3.2.5 Google XRay 12
3.2.6 GCC instrument functions 12
3.2.7 Sampling 12
3.2.8 LLVM/clang AST matchers 12

4 Concepts **13**

4.1 Definitions 13
4.1.1 Time 13
4.1.2 Transducers 13
4.1.3 Timed Transducers 16
4.1.4 Labeled Timed Transducers 21
4.1.5 Events 21
4.1.6 Streams 21

4.1.7 Functions . . . 22
4.1.8 Nodes . . . 23
4.1.9 TeSSLa Evaluation Engine . . . 23
4.1.10 TeSSLa Functions . . . 24
4.1.11 State and History . . . 29
4.1.12 Transitions . . . 30
4.1.13 Run . . . 31
4.2 Behaviour of Different Schedules Without Timing Functions . . . 34
4.2.1 Synchronous Evaluation Engines . . . 35
4.2.2 Asynchronous Evaluation . . . 37
4.3 Equivalence of Different Schedules Without Timing Functions . . . 37
4.3.1 Equivalence of Synchronous Systems . . . 38
4.3.2 Equivalence of Synchronous and Asynchronous Schedules . . . 39
4.4 Behaviour with Timing functions . . . 40
4.5 Equalitys with Timing functions . . . 40
4.6 Parallel computation . . . 40

Bibliography **41**

Glossary **47**

1 Introduction

1.1 Motivation and Problem Statement

Software Verification is an important tool to harden critical systems against faults and exploits. Due to the raising importance of computer based systems, verification has become a big field of research in computer science.

While pure verification approaches try to proof the correct behaviour of a system under all possible executions, Runtime Verification (RV) limits itself to single, finite runs of a system, trying to proof it conforms to a given specification under specific conditions, like input sequences or scheduling. These specifications can be given in various ways, e.g. as a Temporal Logic (TL) formula or in specification languages that are specifically developed for RV. Examples for this are RMOR [3], LOLA [1] and others [8, 6, 4], which we will look at more closely in Section 2.

The language TESSLA aims to make it easy to specify behaviour of streams. To gain this it introduces a number of language features and syntax sugar to expressively describe the conditions a stream should fulfill. The evaluation of TeSSLa specifications is done in two steps: first the specification is compiled by a compiler written at the Institute for Software Engineering and Programming Languages (ISP) of the University of Lübeck. The output is a canonical representation of the operations on the streams in the specification. In the second step the compiled specification is connected with a system that produces some kind of traces, which are treated as the input streams of the specification.

The second step can be done in different ways: online or offline, interweaving the monitors into the monitored program (e.g. [3]) or having a standalone system. These different approaches lead to different manipulations of the original program that should be monitored. When the monitors are interweaved into the program, they can produce new errors or even suppress others. When the monitors are run in a different process or even on different hardware, the overhead and influence to the system can be much smaller, but there will be a bigger delay between the occurence of events in the program and their evaluation in the monitor. Furthermore interweaved monitors can optionally react towards errors by changing the program execution, therefore eliminating cascading errors, while external executions of monitors can't directly modify the program but can still produce warnings to prevent such errors. While online monitoring can be used to actively react to

error conditions, either automatically or by notification of a third party, offline monitoring can be thought of as an extension to software testing ([1]).

At the beginning of this thesis there was one implementation of a runtime for TeSSLa specifications that is based on FPGAs that have to be manually reconfigured for each new specification. While this is a very performant approach for actual monitoring it isn't feasible for testing and prototyping. Therefore it is wanted to implement a runtime for TeSSLa specifications that can be run independent of specific hardware.

Furthermore most RV approaches are specific to one programming language or environment and combine ways of generating the data, which is used for monitoring, and the monitoring itself. TeSSLa specifications themself are independent of any implementation details of the monitored system, working only on streams of data, which can be gathered in any way. This can be used to implement a runtime that is also independent of the monitored system an how traces of it are collected.

During the thesis it is prooven, that the actual approach of this runtime, a functional, actor based, asynchronous system, will generate the same observations on input traces as a synchronous evaluation of the specification. While TeSSLa specifications can work on all kinds of streams, especially on traces on all levels of a program, e.g. on instruction counters or on spawning processes, in this thesis we will mainly focus on the level of function calls and variable reads/writes. Other applications of the system can easily extend it to use traces of drastically different fields, e.g. health data, temperatures, battery levels, web services and more.

To test the software based runtime, different specifications will be tested on multiple traces, some of which are generated by actually running a program, which was instrumented by hand to generate traces, others which are generated or modified by hand to deliberately introduce bugs which should be detected by the system.

1.2 Results

1.3 Thesis Structure

As the whole evaluation engine is built on top of different technical and theroetical ideas, it is structured to show the reasoning behind the decisions that were made during the development. Furthermore it will proof equalitys of different kinds of systems in multiple steps that build on one another. In the following a quick overview of the different parts of the thesis is given.

Chapter 2

In this chapter the theoretical foundation for the system is explained. Furthermore multiple approaches solving similar problems are shown and it is highlighted which concepts of them were used in the new system and which were disregarded and why.

Chapter 3

Moving towards the implementation of the new evaluation engine, in this chapter practical concepts and systems that are used to implement the runtime are shown. It is explained how they are used and which alternatives exist.

Chapter 4

Building on the theoretical and practical findings of the previous chapters different models for the runtime are specified. Afterwards it's shown that the different models describe valid systems to evaluate given specifications on streams.

Chapter ??

To show the value of the implemented system it is thoroughly tested with real world examples and traces. The results of this testing is used to evaluate the implementation.

2 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 is not restricted to monitor programs written in a specific language but can monitor anything that can produce streams of data.

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.

2.1 TeSSLa

The implemented runtime and the theoretic work of this thesis is built upon the TeSSLa project from [2]. For that project a 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 data over time, e.g. a variable value in a program or the temperature of a processor. To model streams TeSSLa defines a timing model. That model is

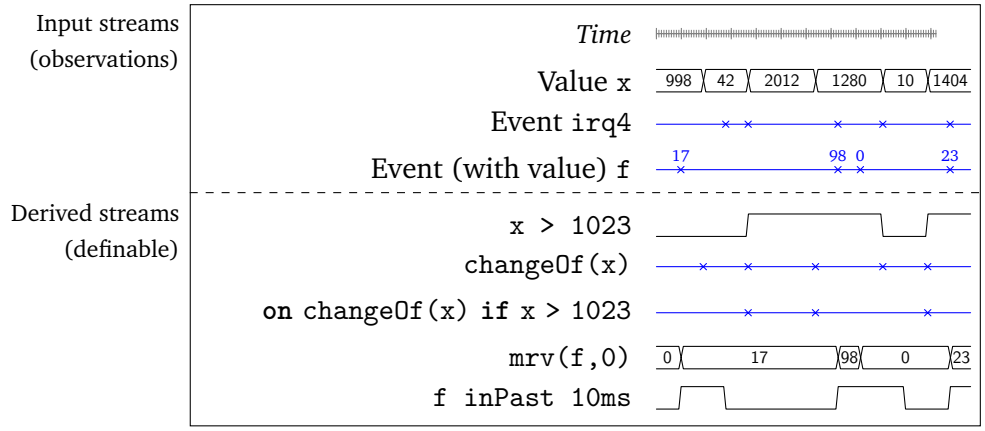

Fig. 2.1: Visualization of TeSSLa stream model, taken from [2]

based on timestamps that are isomorphic to real numbers $\mathbb{R}$. Figure 2.1 shows how streams behave over time.

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

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

2.2 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 TLs 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 a 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.

2.3 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 presented. To do this distributed monitors, which have to communicate with one another, are specified and 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.

2.4 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.

2.5 RMOR

RMOR [3] 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, which is strongly influenced by RCAT [7]. 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. In 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 a critical 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 possible extensions to support this.

3 System

Besides the theoretical basics presented in Section 2 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 itself but also examine related projects which can be used to obtain traces, which then can be monitored by the runtime. 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.

3.1 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.

3.1.1 Erlang and Elixir

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

3.1.2 Implementation

Timing model: reason

3.2 Trace Generation

3.2.1 Debie

3.2.2 TraceBench

3.2.3 Aspect oriented programming

3.2.4 CIL

3.2.5 Google XRay

3.2.6 GCC instrument functions

3.2.7 Sampling

3.2.8 LLVM/clang AST matchers

4 Concepts

In this chapter different ways to evaluate TeSSLa specifications are given and their equivalence is shown. To do so in Section 4.1 building blocks for evaluation approaches are defined, which are then used in later sections to define behaviour of them and show their equivalence.

4.1 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.

4.1.1 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 $\mathbb{R}$.

4.1.2 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, that: 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, that: 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 σ_i is used to get the element at position i and $\sigma_{[i,j]}$ to get the infix of σ which starts at position i and ends at position j (With 0 as the index of the first element).

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:*
$\forall \sigma \in \Phi^*$*:*

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

Lemma 1: Asynchronous equivalence is an equivalence Relationship.
Asynchronous equivalence is symmetric, reflexive and transitive.

Proof.

- Symmetry: trivial, since the second part of the definition is requiring it.
- Reflexivity: Also trivial, for $(\sigma_{[0,k]}, \vec{o})$ select $k' = k$.
- Transitivity:
  - Let $t_1 \equiv_a t_2$, $t_2 \equiv_a t_3$.
  - First case:

    Since $t_1 \equiv_a t_2$: $\forall(\sigma_{[0,k_1]}, \vec{o_1}) \in t_1$:
    $\exists k_2$ such, that $(\sigma_{[0,k_2]}, \vec{o_2}) \in t_2$ with $\vec{o_1} \sqsubseteq \vec{o_2}$
    and since $t_2 \equiv_a t_3$
    $\exists k_3$ such, that $(\sigma_{[0,k_3]}, \vec{o_3}) \in t_3$ with $\vec{o_2} \sqsubseteq \vec{o_3}$
    With $\vec{o_1} \sqsubseteq \vec{o_2} \sqsubseteq \vec{o_3}$ it follows, that $t_1 \equiv_a t_3$

  - The second case works the same, just change t_1 and t_3.

□

Example 4: Asynchronous equivalence of Transducers.
Let $\Phi = \{a\}, \Gamma = \{1\}$ and

$$
\begin{aligned}
t_1 &= \{ \quad (a, \epsilon), \quad (aa, \epsilon), \quad (aaa, 111) \quad \} \\
t_2 &= \{ \quad (a, 1), \quad (aa, 1), \quad (aaa, 111) \quad \} \\
t_3 &= \{ \quad (a, \epsilon), \quad (aa, 1), \quad (aaa, 11) \quad \}
\end{aligned}
$$

All three transducers are asynchronous and causal. Let's see which ones are asynchronous equivalent:

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

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

$\Rightarrow t_1 \equiv_a t_2$

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

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

$\Rightarrow t_1 \not\equiv_a t_3$

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

4.1.3 Timed Transducers

For the second kind of equivalence we need to introduce *timed sequences* and *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 Sequence.
A sequence is called timed, if every element of it is associated with a timestamp: $\sigma \in (\Gamma \times \mathbb{T})^*$. *For bravety a timed sequence can be written with the timestamps as the index of the elements:* $\sigma = e_0 e_{0.5} e_1$.
The function

$$timed : (\Gamma \times \mathbb{T})^* \rightarrow (\Gamma \times \mathbb{T})^*$$

reorders a timed sequence σ *by its timestamps, such that:*

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

The function

$$upto : \mathbb{T} \times (\Gamma \times \mathbb{T})^* \rightarrow (\Gamma \times \mathbb{T})^*$$

removes all elements from a timed sequence, that have a timestamp bigger than the first argument.
The function

$$maxTime : (\Gamma \times \mathbb{T})^* \rightarrow \mathbb{T}$$

returns the biggest Timestamp in a timed sequence.

Example 5: Functions on timed Sequences.
Let $\sigma = a_1 a_{0.5} a_{1.5} a_0$.
Then is

$$timed(\sigma) = a_0 a_{0.5} a_1 a_{1.5}$$
$$upto(1.3, \sigma) = a_1 a_{0.5} a_0$$
$$maxTime(\sigma) = 1.5$$

Definition 8: Monotonicity of Timed Sequences.
A timed sequence σ *with alphabet* Φ *is called monotonic, if* $timed(\sigma) = \sigma$

Definition 9: Timed Transducer.
A timed transducer t *with input alphabet* Φ *and output alphabet* Γ *works on monotonic, timed sequences as inputs and has timed sequences as outputs:*

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

Example 6: Timed Transducers.
Let $\Phi = \{a\}, \Gamma = \{b\}$.
$t_{tsc} = \{(a_0, b_0), (a_0 a_1, b_0 b_1)\}$ *is a timed, causal and synchronous transducer.*
$t_{tac} = \{(a_0, \epsilon), (a_0 a_1, b_0 b_1)\}$ *is a timed, causal and asynchronous transducer.*

For later theoretic work we have to restrict timed transducers:

Definition 10: 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*
$upto(maxTime(\vec{o}), timed(\vec{o} \circ \vec{o'})) = upto(maxTime(\vec{o}), timed(\vec{o} \circ \vec{o'} \circ \vec{o''}))$

Based on the definitions we can define an equivalence relationship on bounded timed transducers:

Definition 11: Observational Equivalence.
Let t_1, t_2 be two bounded 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:*

$$\forall \sigma \in (\Phi \times \mathbb{T})^* :$$
$$\forall (\sigma_{[0,k]}, \vec{o}) \in t_1 : \exists k', k'' \geq k \text{ such that}$$
$$(\sigma_{[0,k']}, \vec{o} \circ \vec{o'}) \in t_1$$
$$\text{and } (\sigma_{[0,k'']}, \vec{o_2}) \in t_2$$
$$\text{and } timed(upto(maxTime(\vec{o}), \vec{o} \circ \vec{o'})) = timed(upto(maxTime(\vec{o}), \vec{o_2}))$$

and the same for switched t_1, t_2.

Lemma 2: Observational Equivalence is an Equivalence Relationship for Bounded Transducers.
$\equiv_o$ is symmetric, reflexive and transitive for bounded timed transducers.

Proof.
Let t_1, t_2, t_3 be bounded timed transducers.

- Symmetry: By definition.

- Reflexivity: For $(\sigma_{[0,k]}, \vec{o})$ select $k' = k''$ as the k, for which the transducer is bounded for that input.

- Transitivity:

  - Let $t_1 \equiv_o t_2$, $t_2 \equiv_o t_3$.

– First case:

Since $t_1 \equiv_o t_2 : \forall(\sigma_{[0,k_1]}, \vec{o_1}) \in t_1 :$

$\exists k_1', k_2 > k_1$ with $(\sigma_{[0,k_1']}, \vec{o_1} \circ \vec{o_1'}) \in t_1$ and $(\sigma_{[0,k_2]}, \vec{o_2}) \in t_2$

with $timed(upto(maxTime(\vec{o_1}), \vec{o_1} \circ \vec{o_1'}))$

$$= timed(upto(maxTime(\vec{o_1}), \vec{o_2})) \quad (\star)$$

and since $t_2 \equiv_o t_3 : \exists k_2', k_3 > k_2$ with $(\sigma_{[0,k_2']}, \vec{o_2} \circ \vec{o_2'}) \in t_2$

and $(\sigma_{[0,k_3]}, \vec{o_3}) \in t_3$

with $timed(upto(maxTime(\vec{o_2}), \vec{o_2} \circ \vec{o_2'}))$

$$= timed(upto(maxTime(\vec{o_2}), \vec{o_3})) \quad (\star\star)$$

$maxTime(\vec{o_1})$ has to be smaller then $maxTime(\vec{o_2})$

else $(\star)$ couldn't hold, therefore, combined with boundedness and $(\star\star)$:

$timed(upto(maxTime(\vec{o_1}), \vec{o_2}))$

$$= timed(upto(maxTime(\vec{o_1}), \vec{o_3}))$$

which concludes $timed(upto(maxTime(\vec{o_1}), \vec{o_1} \circ \vec{o_1'}))$

$$= timed(upto(maxTime(\vec{o_1}), \vec{o_3}))$$

– The second case works the same, just switch t_1 and t_3.

□

Example 7: Observational Equivalence.
Let

$$\begin{aligned}
t_1 &= \{ \quad (a_0, \epsilon), \quad (a_0a_1, b_1), \quad (a_0a_1a_2, b_1b_2b_0) \quad \} \\
t_2 &= \{ \quad (a_0, \epsilon), \quad (a_0a_1, \epsilon), \quad (a_0a_1a_2, b_2b_1b_0) \quad \} \\
t_3 &= \{ \quad (a_0, b_0), \quad (a_0a_1, b_0), \quad (a_0a_1a_2, b_2b_1) \quad \}
\end{aligned}$$

All three are causal, asynchronous timed transducers.

Let's see which ones are observational equivalent:

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

$(a_0, \epsilon) \in t_1, k = 1, maxTime(\epsilon) = 0$

$\rightarrow k' = 1, (a_0, \epsilon) \in t_1$

$\rightarrow k'' = 1, (a_0, \epsilon) \in t_2$

$(a_0a_1, b_1) \in t_1, k = 2, maxTime(b_1) = 1$

$\rightarrow k' = 3, (a_0a_1a_2, b_1b_2b_0) \in t_1$

$\rightarrow k'' = 3, (a_0a_1a_2, b_2b_1b_0) \in t_2$

$timed(upto(1, b_1b_2b_0)) = b_0b_1 = timed(upto(1, b_2b_1b_0))$

$(a_0a_1a_2, b_1b_2b_0) \in t_1, k = 3, maxTime(b_1b_2b_0) = 2$

$\rightarrow k' = 3, (a_0a_1a_2, b_1b_2b_0) \in t_1$

$\rightarrow k'' = 3, (a_0a_1a_2, b_2b_1b_0) \in t_2$

$timed(upto(2, b_1b_2b_0)) = b_0b_1b_2 = timed(upto(2, b_2b_1b_0))$

$(a_0, \epsilon) \in t_2, k = 1, maxTime(\epsilon) = 0$

$\rightarrow k' = 1, (a_0, \epsilon) \in t_2$

$\rightarrow k'' = 1, (a_0, \epsilon) \in t_1$

$(a_0a_1, \epsilon) \in t_2, k = 2, maxTime(epsilon) = 0$

$\rightarrow k' = 2, (a_0a_1, \epsilon) \in t_2$

$\rightarrow k'' = 2, (a_0a_1, b_1) \in t_1$

$timed(upto(0, \epsilon)) = \epsilon = timed(upto(0, b_1))$

$(a_0a_1a_2, b_2b_1b_0) \in t_2, k = 3, maxTime(b_2b_1b_0) = 2$

$\rightarrow k' = 3, (a_0a_1a_2, b_2b_1b_0) \in t_2$

$\rightarrow k'' = 3, (a_0a_1a_2, b_1b_2b_0) \in t_1$

$timed(upto(2, b_2b_1b_0)) = b_0b_1b_2 = timed(upto(2, b_1b_2b_0))$

$\Rightarrow t_1 \equiv_a t_2$

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

$(a_0a_1a_2, b_1b_2b_0) \in t_1, k = 3, maxTime(b_1b_2b_0) = 2$

$\rightarrow k' = 3, (a_0a_1a_2, b_1b_2b_0) \in t_1$

$\rightarrow \not\exists(\vec{\imath}, \vec{o}) \in t_3$ *with* $\exists n \in \mathbb{N} : \vec{o}_n = b_0$

$\rightarrow \not\exists(\vec{\imath}, \vec{o}) \in t_3$ *with* $timed(upto(2, b_1b_2b_0)) = b_0b_1b_2 = timed(upto(2, \vec{o}))$

$\Rightarrow t_1 \not\equiv_a t_3$

If t_3 weren't bounded (and therefore not finite) there would be no way to know, if it was equivalent to t_1, because it could always produce a missing event at a later time.

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

4.1.4 Labeled Timed Transducers

Maybe necessary, maybe not

4.1.5 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 specification, 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 $stream$.

$E_e \subset E$ 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 dmathitnodeil, 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 $E_e \cup E_o \cup E_n = E$.

4.1.6 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 event-streams, 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 the product of a sequence of events $\langle e_1, \dots e_n \rangle$ where $\pi(e_i) < \pi(e_{i+1}),\ \forall i < n \in \mathbb{N}$ and a value from $\mathbb{T}$ which marks the progress of the stream and has to be equal or bigger to the timestamp of the last event. The set of all streams Σ is defined as all possible finite sequences of events $\Sigma = \{\sigma | \sigma \in E^*\} \times \mathbb{T}$. An external stream σ_e is a stream consisting only of external events, the set of all external streams is $\Sigma_e = \{\sigma_e | \sigma_e \in E_e^*\} \times \mathbb{T}$. 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 $\perp$. The progress of a stream can be obtained with $progress(\sigma) = t \in T$. Internal and output streams can be queried for the node that produced them with $node(\sigma) = n \in N$.

4.1.7 Functions

A TeSSLa specification consists of functions over streams. Functions generate new streams by applying an operation on existing 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 a state and will only return events. Unpure, or stateful, functions are evaluated over the values if its inputs at that timestamp and a state and will return not only new events but also an updated state. 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 events with that timestamp, timing functions can emit events with a changed timestamp. In this thesis we will only look at past time functions, meaning functions can only delay timestamps, therefore can't depend on future values.

Timing functions complicate the reasoning about schedules and causality and therefore aren't included in Section 4.2. In Section 4.4 the conclusions of earlier sections will be extended to include timing functions.

4.1.8 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 have a state, described in Section 4.1.11, which contains First In First Out (FIFO) queues, provided by the Erlang plattform, which buffer events from its inputs for later computation.

Every new event added to a queue has to have a bigger timestamp than the last event added to it. This means a queue has a kind of progress timestamp, which denotes the timestamp of the latest event added to it and which is strictly increasing over time. Queues support the standard operators for lists like $hd, tl, ++$ to respectively get the head, the tail or to append to the end.

4.1.9 TeSSLa Evaluation Engine

Because functions in TeSSLa specifications depend on other functions, and these dependencies have to be circle free, the specification can be represented as a Directed Acyclic Graph (DAG), where the functions are vertices and the relationship between functions are edges. This is exactly how the TeSSLa compiler outputs a specification. One can now use the DAG of a TeSSLa specification to synthesize a system to evaluate it over inputs: The vertices of the DAG become nodes representing the functions and the edges are the input and output streams between the nodes. We will call this synthesized system an *evaluation engine*.

When fed with inputs (or *traces*) the engine will produce outputs. The relationship between inputs and outputs that is produced can be seen as a timed transducer. The input to an evaluation engine has to have strictly increasing timestamps. This is needed to have a known progress which can be distributed through the system. If inputs weren't ordered by their timestamp for example the absense of input events on a specific stream couldn't be detected because they always could be present at a later position of the input trace. Especially for offline monitoring this obviously is no problem because the traces can simply be reordered into a strictly increasing sequence, except when multiple input events are at the same timestamp. This can be solved in two ways: either increase the timing precision when generating the

traces or manipulate the timestamps in the traces by adding a minimal offset to them if they are equal to another timestamp.

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. We will use the term *step* to denote that one node was scheduled and performed its 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.

This proof will be carried out for different kinds of schedules in Section 4.3 and Section 4.5, showing that all of them can be used by an evaluation engine. For now we can define when to evaluation engines are called equivalent based on the work in Section 4.1.3:

Definition 12: Equivalence of Evaluation Engines.
Two evaluation engines are called equivalent, if their produced relationships between inputs and outputs are observational equivalent.

Because the possible inputs are infinite we will restrict the definition of equivalence to a specified set of inputs:

Definition 13: Equivalence of Evaluation Engines for Specified Inputs.
Two evaluation engines are called equivalent for a set of inputs, if the relationship for theese inputs and the produced outputs is observational equivalent.

4.1.10 TeSSLa Functions

TeSSLa puts no restrictions on the semantics of functions other than that they have to work on streams or constants and produce streams, but allows to restrict them for evaluation approaches. We are taking advantage of that to categorize functions based on how or if at all they can be encoded in our evaluation approach. The categorization is based upon the relationship between consumption and production of input and output events that a node representing the function in an evaluation engine produces. For this we will use the terms node and functions somewhat interchangeable in the following subsections.

Nodes are only scheduled, if they are enabled, meaning they have events on their inputs buffered. The function implemented by a node is evaluated at the minimal timestamp of the buffered events of the node, this timestamp is called the *evaluation timestamp*. This is important to understand the completeness criteria: It means that

when the function is evaluated there is at least one event with that timestamp on one input. The inverse of that statement shows shows why this is important: a function is never evaluated at a timestamp where no event is present, therefore the system can't arbitrarily produce new timestamps.

Nodes having signals as inputs always have to remember the last occured change of them in their state. When such a node is scheduled, the function will work on the remembered value if no new change is present at the evaluation timestamp for the signal. If a new change is present at the evaluation timestamp, it will be used and the state of the node has to be updated to remember the new change.

It is important to note that all functions always have to consume at least one input event, else an evaluation engine can enter a livelock, where new events are produced forever out of nowhere. Also all functions will only produce a finite amount of events at every evaluation.

Complete Functions

Complete functions will consume one event from every input and produce one event at every timestamp they are evaluated. Most complete functions are pretty simple and often have eventstreams as inputs or only have one input. The complete functions that are present in the implemented runtime are explained in Table 4.1.

The first four functions are sources which take an external event and format them for internal use, e.g. $variable\_values$ takes a string containing the name and the value of a variable, casts the value to an appropriate type and produces a signal holding that produced value.

Output Complete Functions

Output complete functions will produce a new output everytime they are evaluated but only have to consume one event from any input and not one from every input. Table 4.2 summarizes all input complete functions.

Input Complete Functions

Input complete function consume one events from every input but can produce zero or more output events everytime they are evaluated. Table 4.3 summarizes all input complete functions.

Name	Domain	Range	Explanation
instruction_executions	Events	Events	Converts a string to an event that denotes the execution of a specific instruction in the monitored program.
function_returns	Events	Events	Converts a string to an event that denotes the return from a function in the monitored program.
function_calls	Events	Events	Converts a string to an event that denotes the call of a function in the monitored program.
variable_values	Events	Signal	Converts a string to a change that denotes the value of a variable in a monitored program.
signalAbs	Signal	Signal	Computes the absolute value of a signal.
eventAbs	Events	Events	Computes the absolute value of an event.
changeOf	Signal	Events	Emits an event everytime the signal changes its value holding the new value.
neg	Signal	Signal	Emits the mathematical opposite of the value of a signal.
signalNot	Signal	Signal	Emits the boolean negation of a signal.
eventNot	Events	Events	Emits the boolean negation of an event.
eventCount	Events	Signal	Emits a signal holding the number of times an event occured on the input.
timestamps	Events	Events	Emits an event holding the timestamp of an input event everytime one occurs.
sma	Events	Events	Emits an event holding the simple moving average over the last specified number of events that occured.

Tab. 4.1: List of complete functions

Name	Domain	Range	Explanation
$merge$	Events $\times$ Events	Events	Emits an event holding the value of the first input if there is an event at that timestamp or else of the second input. Not input complete because if an event on the second input occurs at a timestamp where no event of the first input occurs no event of the first input is removed.
$occurAny$	Events $\times$ Events	Events	Emits an event without a value everytime an event occurs on any input. Not input complete because events are only removed from both inputs if they have the same timestamp.

Tab. 4.2: List of output complete functions

Name	Domain	Range	Explanation
$signalMaximum$	Signal	Signal	Emits a change everytime the input has a bigger value than it had anytime before.
$eventMaximum$	Events	Signal	Emits a change everytime the input has a bigger value than it had anytime before or a default value if it is the biggest value occured yet.
$signalMinimum$	Signal	Signal	Emits a change everytime the input has a smaller value than it had anytime before.
$eventMinimum$	Events	Signal	Emits a change everytime the input has a bigger value than it had anytime before or a default value if it is the biggest value occured yet.
sum	Events	Signal	Emits the summed up value of all events that happened on the input upto that point.
mrv	Events	Signal	Emits a change everytime the input takes a new value. Not output complete because no new change is emitted if the last value of the input was the same as the current.

Tab. 4.3: List of input complete functions

Incomplete Functions

Incomplete functions always consume at least one input from any input but don't have to produce an output. Table 4.4 lists all supported incomplete functions. If no explanation is given, why the function is incomplete it is the following: The function is not input complete, because it only consumes events or changes that have the timestamp at which it is evaluated, if one input only has events with bigger timestamps no event or change is removed from them and the remembered last change of them is used as a base for computation if it is a signal. Also it is not output complete, because changes of a signal are only produced, if the value actually changes. For example the add function could consume two input events with the same timestamp, but the sum of their values is the same value as the old value of the signal.

Name	Domain	Range	Explanation
add	Signal $\times$ Signal	Signal	Adds both inputs.
and	Signal $\times$ Signal	Signal	Performs a boolean and over both inputs.
div	Signal $\times$ Signal	Signal	Divides the first input by the second input.
eq	Signal $\times$ Signal	Signal	Emits if both inputs are equal.
geq	Signal $\times$ Signal	Signal	Emits if the first input is greater or equal to the second input.
gt	Signal $\times$ Signal	Signal	Emits if the first input is greater than the second.
$implies$	Signal $\times$ Signal	Signal	Emits the boolean implies relationship between both inputs.
leq	Signal $\times$ Signal	Signal	Emits if the first input is smaller or equal to the second.
lt	Signal $\times$ Signal	Signal	Emits if the first input is smaller than the second.
$\max$	Signal $\times$ Signal	Signal	Emits the bigger value of both inputs.
$\min$	Signal $\times$ Signal	Signal	Emits the smaller value of both inputs.
mul	Signal $\times$ Signal	Signal	Multiplies the first input by the second.
or	Signal $\times$ Signal	Signal	Performs a boolean or over both inputs.
sub	Signal $\times$ Signal	Signal	Subtracts the second input from the first.

filter	Events × Signal	Events	Emits events whenever an event occurs on the first input with the value of that event if the second input has the value true. Is not output complete because it doesn't emit events when the second input is false.
ifThen	Events × Signal	Events	Emits an event with the value of the second input everytime an event occurs on the first input. Is not output complete because it only emits outputs when an event occured on the first input.
ifThenElse	Signal × Signal × Signal	Signal	Emits the value of the second input inf the first is true, else of the third input.
sample	Signal × Events	Events	Same as *ifThen* with switched arguments.
occurAll	Events × Events	Events	Emits an event whenever an event occurs on both inputs. Not output complete because it only emits events whenever event shappen on both inputs.

Tab. 4.4: List of incomplete functions

Timing Functions

Timing functions are a bit special, therefore they are mentioned here in their own section. Basically they are also incomplete functions, but they have to buffer multiple events until they are emitted. TODO

4.1.11 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 contains arbitray information, e.g. a counter for a *CountNode*, its input queues holding the non-processed events and, if they have signals as inputs, the last changes of them.

To distinguish between the two types of states, the state of the whole engine is called the *global state* and the state of a single 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 denote 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.

Nodes, and therefore the whole evaluation engine, change their state when they are scheduled. The transition between states is described in Section 4.1.12

The history of an evaluation engine is defined at every step (read: after every computation of a node) as all events that were produced by any node upto that step.

4.1.12 Transitions

A transition describes what happens when the evaluation engine schedules a node: Events from the the inputs of a node are removed (at least one), output events can be generated (but don't have to) and distributed and the internal state of nodes are updated. Because the function of a node is evaluated at the evaluation timestamp, which is the minimal timestamp of all events on the heads of inputs, the events which are removed are exactly the ones that have the evaluation timestamp. To look at it in another way: a transition models the computation of a node and the progressing of the stream it produces towards the evaluation timestamp, which has to be bigger than the previous progress, since input queues are strictly ordered by their timestamp and the events with the minimal timestamps are removed after the computation. Therefore when we say 'Node a is scheduled' we mean that a transition is taken which models the computation of that node.

The set of all transitions is written as T. The function $node : T \to N$ returns the node of which the transitions models the computation.

One part of a transition is a relation between two sets of events, why sometimes we write $\tau = (\{e_1, e_2\}, \{e_3\})$ to visualize a transition, but remember that there is more to a transition than that. E.g. the relation $\tau = (\{e_1, e_2\}, \{e_3\})$ means that two events were consumed by a node and one event was produced based on them.

The empty transition, meaning no input was consumed and no output produced, is labeled with λ. Note that all transitions, which produces events 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 (see Section 4.1.10). Furthermore with timing functions it's possible to create multiple events in one transition. For example think of an $EchoNode$, which duplicates an input after a specified amount of time.

Definition 14: Application of a Transition on a State.
Given a global state s_0 and a transition $\tau = (\widetilde{E}, \widetilde{E'}) = (\{e_1, e_2, \ldots, e_i\}, \{e'_1, e'_2, \ldots, e'_i\})$ with $n = node(\tau_1)$ and N_c the set of all nodes that are children of n. When we apply *τ to s_0 we get a new global state s_1 with*
$\forall \widetilde{n}_i = s_0(i)$

- *if $node(\widetilde{n}_i) \notin N_c \cup \{n\}$ then $s_1(i) = \widetilde{n}_i$ (no node states changes except the one from the node of the transition and its children)*
- *else if $node(\widetilde{n}_i) \in N_c$ then append all events in $\widetilde{E'}$ to the input representing the stream from n*
- *else remove all events in $\widetilde{E}$ from the inputs representing their streams and update the internal information based on the function n is modelling.*

This means that the new global state is built with the old global state by altering only the node states of the node identified by the transition and its children. The node states are altered by removing all events that were consumed by the function from the inputs of the scheduled node, updating its internal state and adding the produced events to the queues representing it in the node states of its children.

4.1.13 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 $length(r) = d \in \mathbb{N}$. A run can be queried by it's index to return the element at that index: $r(i) = (\tau_i, s_i),\ i \in [0, length(r)]$.

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 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. Note that two runs have no closeness if they don't contain the same transitions.

Example 8:.
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

- $\delta_{1,2} = \delta(r_1, r_2) = (3, 3)$ *because* r_1 *takes* tau_3 *at step* 3 *while* r_2 *takes* tau_5 *and* r_2 *takes* tau_3 *at step* $3 + 3 = 6$.
- $\delta_{1,3} = \delta(r_1, r_3) = (4, 1)$ *because* r_1 *takes* tau_4 *at step* 4 *while* r_3 *takes* tau_5 *and* r_2 *takes* tau_4 *at step* $4 + 1 = 5$.
- *The explanations for the remaining cases is analogous and therefore not stated here.*
- $\delta_{2,1} = \delta(r_2, r_1) = (3, 2),\ \delta_{2,3} = \delta(r_2, r_3) = (3, 1)$
- $\delta_{3,1} = \delta(r_3, r_1) = (4, 1),\ \delta_{3,2} = \delta(r_3, r_2) = (3, 2)$

The ordering of the distances is straightforward: $\delta_{1,3} < \delta_{2,3} < \delta_{2,1} < \delta_{2,1}$.

To reason about runs we have to restrict runs to the ones that are reasonable in the context of an evaluation engine. This means that only transitions are taken that are possible based on the global state. To do so we have to define when a node can compute based on its state. At first we will give a definition that only works for output complete TeSSLa functions. Based on that definition we will see the problems that output incomplete functions have and modify the transition model to fix the problem.

Definition 16: Preliminary Enabledness of a Node.
A node n *with the node state* $\widetilde{n}$ *containing the input queues* $\widetilde{\sigma}$ *in an evaluation engine is called* enabled *at a step* i *of a run* r *of that engine, if at least one input is buffered on each input queue:*

$$\forall \sigma_x \in \widetilde{\sigma} : hd(\sigma_x) = e_x$$

As stated this definition works for all output complete functions, while output incomplete functions have a problem. It is possible that an output incomplete functions will never produce a new output: E.g. think of a $FilterNode$ where the second input is always $false$. Even if new input events are added to the first input and the second input is known to be $false$ upto any timestamp, the children of the node will never receive new events for that input queue. Therefore all children can never again compute, because they don't have an event buffered on all inputs.

There are two ways to fix this:
All input queues could be extended with a progress timestamp, which denotes how far the parent node has progressed. Now everytime a node n with children N_c evaluates its function at a new evaluation timestamp the inputs of all nodes in N_c

representing the stream of n would be updated to have that evaluation timestamp as their new progress, even if no new event was produced at that timestamp.

The other way, which is used in this thesis, doesn't alter the input queues: First we need a new type of events: *progress events*. A progress event holds no value and exists only to notify a node, that a specific input has progressed upto the timestamp of the progress event. We will write a progress event as e_t^p where t is the timestamp of it. Now, whenever a node performs its computation at an evaluation timestamp and no new event was produced, a progress event with the evaluation timestamp is added to the corresponding input queue of all children. When a Node computes at an evaluation timestamp and one of the consumed events is a progress event there are two possibilities:

- If the progress event is on a signal, the last change of the signal, which is saved in the state of the node, is used as the argument of the function and the last change of that stream is not updated in the state.

- If the progress event is on an eventstream it means that no events were produced upto and including that timestamp. This information can be used to produce timeouts if needed.

No matter the kind of the stream the progress event happened on, if the node that consumed it produced no new output for the computation at that evaluation timestamp, it will propagate the progress event to its children. With the addition of progress events the preliminary definition of enabledness for a node works for all kinds of TeSSLa functions.

Now it is possible to restrict runs to a subset where each run models a rational evaluation of a specification.

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

- $\forall i \in [0, length(r)] : r(i) = (\tau_i, s_i) \wedge node(\tau_i)$ *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 $node(\tau_1)$ is independent of $node(\tau_2)$.

Lemma 3: 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 $a = node(\tau_1)$ is no descendant of $b = node(\tau_2)$, the stream σ with $node(\sigma) = node(\tau_2)$ 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 4: 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 3 □

Lemma 5: Duration of Enabledness.
A node which is enabled stays enabled at least until it is scheduled. Formally: If a node n is enabled at step i in a run r it will stay enabled at least until step $j > j$ with $r(j) = (\tau, s), node(\tau) = n$. 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. □

4.2 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 4.1.9. While there are 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 its 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.

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

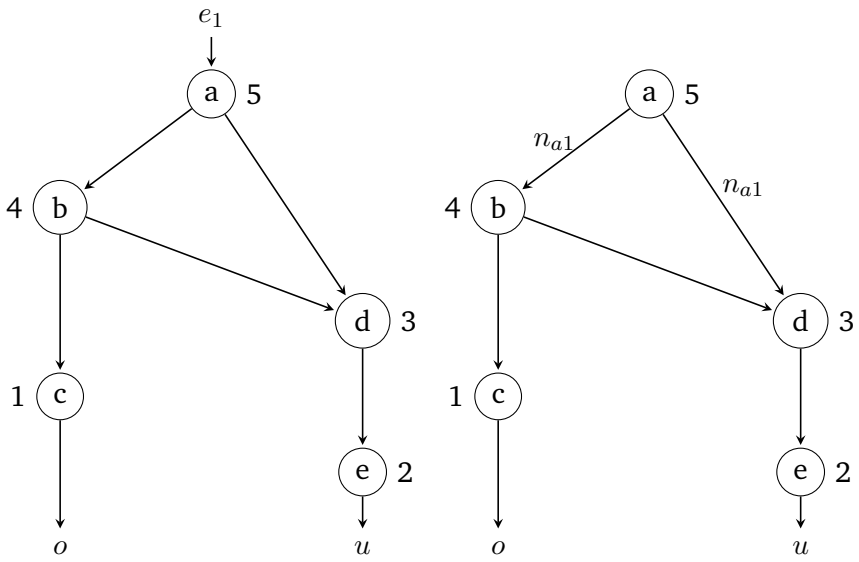

Fig. 4.1: Visualization of a simple asynchronous system with a reversed topological order.

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

Figure 4.1 visualizes a synchronous evaluation engine. It shows two DAGs 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.

4.2.2 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

4.3 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 6: 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 4.3.1 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 4.3.2 it is shown that any asynchronous evaluation engine is equivalent to a synchronous one.

4.3.1 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 4.2.1. 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 is closer 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 5, 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 4 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 4.2 visualizes how changing the order of the two transitions can't change the global state

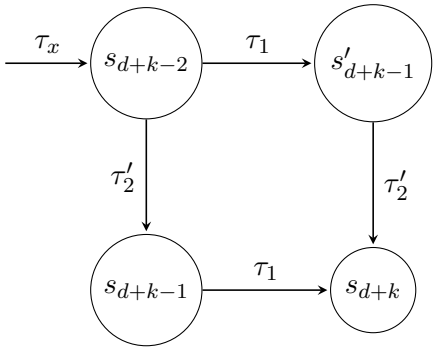

Fig. 4.2: Commutativity Diagramm of Node scheduling

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.

□

4.3.2 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.

4.4 Behaviour with Timing functions

4.5 Equalitys with Timing functions

4.6 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 pp. 1, 2, 6).

[2] Normann Decker, Daniel Thoma, and Jannis Harder. „TESSLA A Temporal Stream-based Specification Language". 2016 (cit. on pp. 5, 6).

[3] Klaus Havelund. „Runtime Verification of C Programs". In: (2008) (cit. on pp. 1, 5, 8).

[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 pp. 1, 7).

[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. 8).

[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. 1, 7, 8).

[7] Margaret H. Smith and Klaus Havelund. „Requirements capture with RCAT". In: *Proceedings of the 16th IEEE International Requirements Engineering Conference, RE'08* (2008), pp. 183–192 (cit. on p. 8).

[8] Xi Zheng, Christine Julien, Rodion Podorozhny, and Franck Cassez. „BraceAssertion: Runtime verification of cyber-physical systems". In: *Proceedings - 2015 IEEE 12th International Conference on Mobile Ad Hoc and Sensor Systems, MASS 2015* (2015), pp. 298–306 (cit. on p. 1).

List of Figures

2.1 Visualization of TeSSLa stream model, taken from [2] 6

4.1 Visualization of a simple asynchronous system with a reversed topological order. 36
4.2 Commutativity Diagramm of Node scheduling 39

List of Tables

4.1 List of complete functions 26
4.2 List of output complete functions 27
4.3 List of input complete functions 27
4.4 List of incomplete functions 29

Glossary

LOLA A specification language and algorithms for the online and offline monitoring of synchronous systems including circuits and embedded systems. 1, 6, 7

RCAT Requirements CApture Tool. 8

RMOR Requirement Monitoring and Recovery. 1, 8

TESSLA A temporal, stream based specification Language. 1, 2, 5–9, 11, 13, 21–24, 29, 32–34, 37, 38, 43

DAG Directed Acyclic Graph. 23, 35–38

FIFO First In First Out. 23

ISP Institute for Software Engineering and Programming Languages. 1

RV Runtime Verification. 1, 2, 7, 8

TL Temporal Logic. 1, 6

List of Theorems

1 Definition (Transducer) . . . 13
2 Definition (Deterministic Transducer) . . . 14
3 Definition (Synchronous Transducer) . . . 14
4 Definition (Asynchronous Transducer) . . . 14
5 Definition (Causal and Clairvoyant Transducers) . . . 14
6 Definition (Asynchronous equivalence of Transducers) . . . 15
1 Lemma (Asynchronous equivalence is an equivalence Relationship) . 15
7 Definition (Timed Sequence) . . . 16
8 Definition (Monotonicity of Timed Sequences) . . . 17
9 Definition (Timed Transducer) . . . 17
10 Definition (Boundedness of Timed Transducers) . . . 17
11 Definition (Observational Equivalence) . . . 17
2 Lemma (Observational Equivalence is an Equivalence Relationship for Bounded Transducers) . . . 18
12 Definition (Equivalence of Evaluation Engines) . . . 24
13 Definition (Equivalence of Evaluation Engines for Specified Inputs) . 24
14 Definition (Application of a Transition on a State) . . . 31
15 Definition (Closeness of Runs) . . . 31
16 Definition (Preliminary Enabledness of a Node) . . . 32
17 Definition (Valid Run) . . . 33
18 Definition (Independence of Nodes) . . . 33
19 Definition (Independence of Transitions) . . . 33
3 Lemma (Exchange of Independent Transitions) . . . 34
4 Lemma (Influence of Independent Nodes) . . . 34
5 Lemma (Duration of Enabledness) . . . 34
20 Definition (Valid Evaluation Engines) . . . 36
6 Lemma (Equivalence of Engines for one Input) . . . 37
1 Theorem (Equivalence of Different Synchronous Evaluation Engines) 38