UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

# An Asynchronous Evaluation Engine for Stream Based Specifications

Asynchrone Evaluierung von Strombasierten Spezifikation

**Masterarbeit**

im Rahmen des Studiengangs
**Informatik**
der Universität zu Lübeck

vorgelegt von
**Alexander Schramm**

| | |
|---|---|
| *ausgegeben und betreut von* | Prof. Dr. Martin Leucker |
| *mit Unterstützung von* | Cesar Sanchez, Ph.D. |

Die Arbeit ist im Rahmen einer Tatigkeit beim IMDEA Software Institute entstanden.

Lübeck, den 20. November 2016

**Alexander Schramm**
*An Asynchronous Evaluation Engine for Stream Based Specifications*
Masterarbeit, 20. November 2016
Reviewers: Prof. Dr. Martin Leucker
Supervisors: Cesar Sanchez, Ph.D.

**University of Luebeck**
Institute For Software Engineering and Programming Languages
Ratzeburger Allee 160
23562 Luebeck

# Abstract

# Abstract (Deutsch)

# Acknowledgement

# Contents

# 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 [**Havelund2008**], LOLA [**DAngelo2005**] and others [**Zheng2015**, **Pike2010**, **Mostafa2015**], which we will look at more closely in Chapter 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. [**Havelund2008**]) 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 ([**DAngelo2005**]).

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

The main contributions of this thesis consist of three parts. The first is a theoretical approach to asynchronously evaluate timed specifications over streams. The second is an implementation that can synthesize systems to evaluate such specificationss based on the theoretical approach. And the third is a proof of concept implementation of a system that can instrument code which is compiled with Low Level Virtual Machine (LLVM), mainly targeted at C and C++. TODO: more

## 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**

Building on the theoretical and practical findings of the previous chapters new definitions are presented, which are needed to reason about the implemented system.

**Chapter 4**

The work from the previous chapter is put to work to reason about the semantics of the implemented runtime and to show its correctness.

**Chapter 5**

This chapter highlights technical details of the implementation. It will present alternative implementation approaches and the reasoning why specific choices were made during the development of the system.

**Chapter 6**

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

**Chapter 7**

On the basis of the evaluation in the conclusion the results of the thesis are summarized. Furthermore it is highlighted what remains to do and which future challenges exist.

# 2 Related Work

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

As stated in [**Havelund2008**] 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 platform 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 [**Decker2016**]. 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 based on timestamps that are isomorphic to real numbers $\mathbb{R}$. Figure 2.1 shows how streams behave over time.

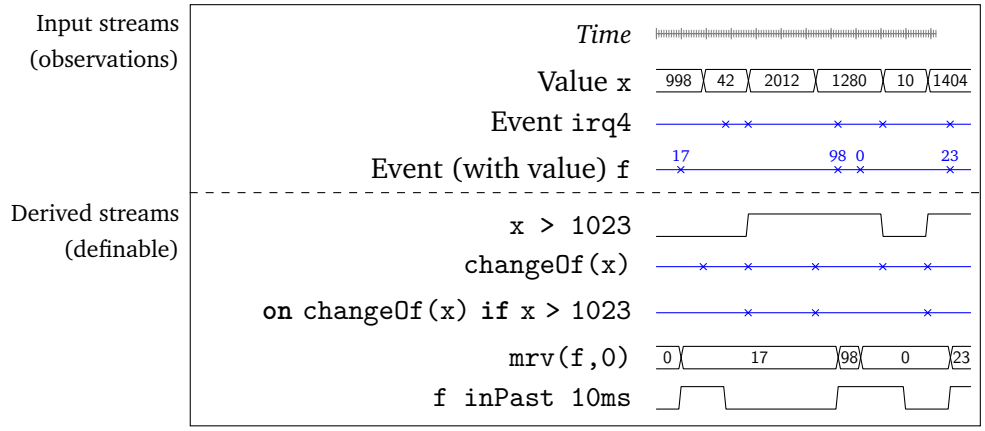


Fig. 2.1: Visualization of TeSSLa stream model, taken from [**Decker2016**]

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 [**DAngelo2005**] 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 [**DAngelo2005**]. 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 [**Mostafa2015**] 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 [**Pike2010**]. 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. [**Pike2010**]

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 [**Havelund2008**] 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 [**Smith2008**]. The specifications are then interweaved into the program using CIL [**Necula2002**]. 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 6.3.

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 7.3.1 we will look at possible extensions to support this.

## 2.6 BeepBeep 3

TODO: Sync event stream processing. Note the concept of the front.

## 2.7 Trace Data

Problem: Many traces don't carry timestamp (see DeCapo, CRV 15)

`http://lttng.org` `http://diamon.org/ctf/` `https://github.com/efficios/barectf`

### 2.7.1 Debie

### 2.7.2 TraceBench

### 2.7.3 Aspect oriented programming

### 2.7.4 CIL

### 2.7.5 Google XRay

### 2.7.6 Sampling

### 2.7.7 LLVM

# 3 Preliminaries

In this chapter we will define concepts that are used in Chapter 4 to reason about the implemented runtime.

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 runtime is built (Note that TeSSLa doesn't define an operational semantic, therefore we will define our own) and how it behaves.

## 3.1 Time

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

## 3.2 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 $\Phi, \Gamma$ be two alphabets and $\epsilon$ the empty word.

**Definition 1: Transducer.**
*A transducer $t$ is a relation $t \subseteq \Phi^* \times \Gamma^*$, $\Phi$ is called the input alphabet, $\Gamma$ 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 input* $a$ *to both outputs* $1$ *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 $\sigma_i$ is used to get the element at position $i$ and $\sigma_{[i,j]}$ to get the infix of $\sigma$ 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 $\Phi^*$ to $\Gamma^*$. 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 Relation.**
*Asynchronous equivalence is symmetric, reflexive and transitive.*

*Proof.*
Symmetry is trivial, since the second part of the definition is requiring it.
Reflexivity is also trivial, for $(\sigma_{[0,k]}, \vec{o})$ select $k' = k$.
For 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* $\Rightarrow t_2 \not\equiv_a t_3$.

## 3.3 Timed Transducers

For the second kind of equivalence we need to introduce *timed sequences*, originally introduced as timed words in [**Alur1994**], and *timed transducers*. Note that timed sequences don't have to be monotonically increasing like in the original definition. Quite on the contrary the unorderdness of outputs is an important key principle to much of the later work as you will see.

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* $\sigma$ *by its timestamps, such that:*

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

*The function*

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

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

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

*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* $\sigma$ *with alphabet* $\Phi$ *is called monotonic, if* $timed(\sigma) = \sigma$

**Definition 9: Timed Transducer.**
*A timed transducer* $t$ *with input alphabet* $\Phi$ *and output alphabet* $\Gamma$ *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 $\Phi$ and output alphabet $\Gamma$ 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 $\Phi$ and output alphabet $\Gamma$. 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* $timed(upto(maxTime(\vec{o}), \vec{o} \circ \vec{o'})) = timed(upto(maxTime(\vec{o}), \vec{o_2}))$

*and the same for switched $t_1, t_2$.*

What does observational equivalence between two transducers intuitively mean? It means that two transducers eventually produce the same output values for the same timed inputs, maybe in a different order, but with the same timestamps, which is very important. Since the values are associated with timestamps the outputs can be reordered by them and therefore be exactly equal.

**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 follows directly from the 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:*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$\Rightarrow t_1 \equiv_a t_2$

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

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

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

$\rightarrow \nexists(\vec{\imath}, \vec{o}) \in t_3$ *with* $\exists n \in \mathbb{N} : \vec{o}_n = b_0$

$\rightarrow \nexists(\vec{\imath}, \vec{o}) \in t_3$ *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 $\Rightarrow t_2 \not\equiv_a t_3$.*

## 3.4 Labeled Timed Transducers

Maybe necessary, maybe not

## 3.5 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 $\upsilon$, its timestamp with $time$ 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 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 $E_e \cup E_o \cup E_n = E$.

## 3.6 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 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 $\sigma$ can be represented as the product of a sequence of events $\langle e_1, \dots e_n \rangle$ where $time(e_i) < time(e_{i+1}),\ \forall i < n \in \mathbb{N}$. The set of all streams $\Sigma$ is defined as all possible finite sequences of events $\Sigma = \{\sigma | \sigma \in E^*\}$. An external stream $\sigma_e$ 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 $\sigma$ at a timestamp $\pi$ it can be queried like a function: $\sigma(\pi) = e$ with $time(e) = \pi$. When working with signals, the function will return the latest event that happened at or before t while an eventstream may return $\bot$. The progress of a stream, which is the timestamp of the last event that happened on them, can be obtained with $progress(\sigma) = \pi \in \mathbb{T}$. Internal and output streams can be queried for the node that produced them with $node(\sigma) = n \in N$.

## 3.7 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 4.1. In Section 4.3 the conclusions of earlier sections will be extended to include timing functions.

## 3.8 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 3.11, which contains First In First Out (FIFO) queues, provided by the Erlang platform, which buffer events from its inputs for later computation.

Every new event added to a queue has to have a bigger timestamp than the previous event added to the queue. 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.

## 3.9 TeSSLa Evaluation Engine

Because functions in TeSSLa specifications depend on other functions, and these dependencies have to be cycle 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 events 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 proven that a chosen schedule produces the correct conclusions for a specification, else the evaluation engine is not valid.

An evaluation engine is run inside an environment. The environment has knowledge over the state of the nodes, most important which nodes are enabled. Based on that information the environment is responsible of feeding the trace data to the engine: only when no sources are enabled the next trace is added to the queue of an input node. This ensures that the consumption of input signals is strictly ordered by timestamp, which is important as we will see in Chapter 4.

## 3.10 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 can only be scheduled when they are enabled, meaning they have events on their inputs buffered. When a node is scheduled, it will compute the minimal timestamp of all buffered events. The function implemented by the node is then evaluated at that timestamp, which 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 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 will be updated to remember the new change.

It is important to note that all functions always have to consume an event from at least one input queue, 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. Especially only timed functions can produce more than one event at an evaluation timestamp.

### 3.10.1 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 3.1.

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.

### 3.10.2 Output Complete Functions

Output complete functions will produce a new event everytime they are evaluated but only have to consume events from some inputs and not from all. Table 3.2 summarizes all input complete functions.

### 3.10.3 Input Complete Functions

Input complete function consume one events from every input but can produce zero or one output events everytime they are evaluated. Table 3.3 summarizes all input complete functions.

| Name | Domain | Range | Explanation |
|---|---|---|---|
| *instruction_executions* | Events | Events | Converts a trace to an event that denotes the execution of a specific instruction in the monitored program. |
| *function_returns* | Events | Events | Converts a trace to an event that denotes the return from a function in the monitored program. |
| *function_calls* | Events | Events | Converts a trace to an event that denotes the call of a function in the monitored program. |
| *variable_values* | Events | Signal | Converts a trace 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 real signal. |
| *signalNot* | Signal | Signal | Emits the Boolean negation of a Boolean signal. |
| *eventNot* | Events | Events | Emits the Boolean negation of a Boolean 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. 3.1:** List of complete functions

### 3.10.4 Incomplete Functions

Incomplete functions always consume at least one input from any input and will produce zero or one events. Table 3.4 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

| Name | Domain | Range | Explanation |
|---|---|---|---|
| *merge* | Events × Events | Events | Merges two eventstreams. When an event is present on the first input, will emit an event with the same value, else with the value from the event on 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 × 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. 3.2:** 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. 3.3:** List of input complete functions

not output complete, because changes of a signal are only produced if the value actually changes. For example, if the values of the inputs of an *add* are switched at a timestamp it would not produce a new change for that timestamp but consume changes from both inputs.

| 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 $\times$ 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. It is not output complete because it doesn't emit events when the second input is false. |
| $ifThen$ | Events $\times$ Signal | Events | Emits an event with the value of the second input everytime an event occurs on the first input. It is not output complete because it only emits outputs when an event occured on the first input. |

| Name | Domain | Range | Explanation |
|---|---|---|---|
| $ifThenElse$ | Signal $\times$ Signal $\times$ Signal | Signal | Emits the value of the second input inf the first is true, else of the third input. |
| $sample$ | Signal $\times$ Events | Events | Same as $ifThen$ with switched arguments. |
| $occurAll$ | Events $\times$ Events | Events | Emits an event whenever events occur on both inputs. Not output complete because it only emits events whenever events happen on both inputs. |

**Tab. 3.4:** List of incomplete functions

### 3.10.5 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 and they can produce more than one event when evaluated. TODO

## 3.11 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 3.12

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.

## 3.12 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 $\lambda$. 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 3.10). 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 12: Application of a Transition on a State.**
*Given global state* $s_0$ *and 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 *$\tau$ to $s_0$, written $apply(s_0, \tau) = s_1$, 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 $s_1(i) = \widetilde{n}_i$*

*(nothing changes for independent nodes)*

*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.

## 3.13 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 $\tau_1$ and thereby reached the state $s_1$.

**Definition 13: Closeness of Runs.**
*The closeness $\delta$ 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: Closeness of Runs.**
*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 14: 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:*

$$\forall \sigma_x \in \widetilde{\sigma} : \neg empty(\sigma_x)$$

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

**Definition 15: 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* $s_i = apply(s_{i-1}, \tau_i)$

As stated, the definition for enabledness 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 no children can ever 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 $e^p_\pi$ where $\pi$ 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 is scheduled and one of the consumed events is a progress event, what happens depends on the function of the node. Some examples are:

- An $AddNode$ has at an evaluation timestamp a progress event as the first input and no event as the second input, which means there is an event buffered on input queue two, else the node wouldn't have been scheduled, and that event has a bigger timestamp than the evaluation timestamp, else its timestamp would have been the evaluation timestamp. The node can't produce a new change for its output, since all new information is, that input one hasn't changed upto the new timestamp. Therefore the node will distribute a progress event with the evaluation timestmap to its children.

- If the $AddNode$ would have received a progress event on input one and a new change on input two at the same timestamp, it could emit a new change. The new event would have the value of the last change on the first input, which the node has to hold in its state, added with the value of the new change on the second input.

- A $MergeNode$ has a progress event as the first input at an evaluation timestamp and no event on the second input. By the same reasoning as in the first example, the node can't produce a new output and therefore will produce a progress event.

These examples show that progress events alters the categorization of TESSLA functions: output complete functions could produce no new normal output, if inputs are progress events. This is no problem, since the main reason for the categorization was to explain the necessity of progress events. With progress events all functions are output complete, because they always emit either normal events or progress events.

For the comparision of runs some more definitions are needed.

**Definition 16: Independence of Nodes.**
*A node $a$ is called* independent *of node $b$ in an evaluation engine, if $a$ is no descendant of $b$.*

**Definition 17: Independence of Transitions.**
*A transition $\tau_1$ is called independent of another transitions $\tau_2$, if $node(\tau_1)$ is independent of $node(\tau_2)$.*

**Lemma 3: Exchange of Independent Transitions.**
*If a transition $\tau_2$ is independent of a transition $\tau_1$, then for all runs of the evaluation engine that produces the runs the following holds:*

$$\begin{aligned} \textit{If } r_1 = \langle(\lambda, s_0), \ldots, (\tau_1, s_i), (\tau_2, s_j), \ldots, (\tau_l, s_l)\rangle &\textit{is a valid run} \\ \textit{then } r_2 = \langle(\lambda, s_0), \ldots, (\tau_2, s_i'), (\tau_1, s_j), \ldots, (\tau_l, s_l)\rangle &\textit{is a valid run} \end{aligned}$$

*Proof.* As a first step we will show, that the transitions $\tau_1$ and $\tau_2$ can be exchanged because their enabledness doesn't depend on each other.

Because $b = node(\tau_2)$ is no descendant of $a = node(\tau_1)$, the stream $\sigma$ with $node(\sigma) = node(\tau_1)$ can be no input of $b$. Therefore the enabledness of $b$ can't be changed by $\tau_1$, taken directly from the definition of enabledness. So $b$ has to be enabled before $\tau_1$ was taken in $r_1$ or else it couldn't be enabled afterwards and $\tau_2$ couldn't be taken in the next step. Therefore $r_2$ also fulfills the requirements of a valid run up to and including the steps $(\tau_2, s_i')(\tau_1, s_j)$.

As a second step we have to show, that the state $s_j$ will stay the same, no matter the order of the two transitions and only the state $s_i$ may be changed to $s_i'$ in $r_2$. If this holds, the subsequent states can't change either, since they are deterministically built by applying the same transitions. Now if all subsequent states stay the same, all subsequent transitions will stay enabled, since enabledness of a node only depends on the state. Figure 3.1 visualizes the exchange of the transitions and the argument why the state stays the same.

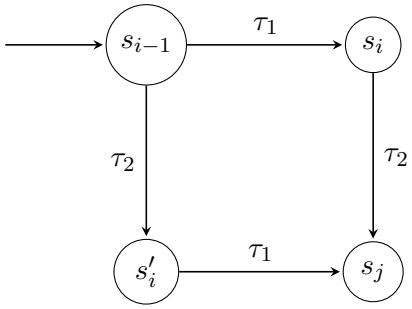


**Fig. 3.1:** Influence of the order of independent transitions on the global state of an evaluation engine

Remember that $s_i = apply(s_{i-1}, \tau_1)$ and $s'_i = apply(s_{i-1}, \tau_2)$. We have to show that $apply(apply(s_{i-1}, \tau_1), \tau_2) = apply(apply(s_{i-1}, \tau_2), \tau_1)$. This is straightforward when recalling what a transition does when it's being applied. $apply(apply(s_{i-1}, \tau_1), \tau_2)$ appends all events produced by $\tau_1$ to the queue representing $n_1 = node(\tau_1)$ in the node states of the children of $n_1$ and updates the node state of $n_1$. Then it appends all events produced by $\tau_2$ to the queue representing $n_2 = node(\tau_2)$ in the node state of the children of $n_2$ and updates the node state of $n_2$.

Since $n_1$ and $n_2$ are independent of each other, neither of them have input queues representing the other, therefore no event can be added to a queue of $n_2$ with $t_1$ and the same with $n_1$ and $\tau_2$. Therefore all events that were appended by applying $\tau_1$ will still be in the queues after $\tau_2$ was applied, since only events from the queues of $n_2$ can be consumed, and the same if they were applied in reversed order.

To conclude: All events appended to any node states will still be present after both transitions were applied and the node states of $n_1$ and $n_2$ will be updates based on the same events, no matter in which order the transitions are run. Therefore the state after both transitions are applied have to be the same, no matter the order in which they are applied.

□

**Lemma 4: 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 the first step $j > i$ with $r(j) = (\tau, s), node(\tau) = n$. Note that it doesn't have to be disabled after step $j$, because there could have been multiple events buffered on it's inputs.*

*Proof.* Let $n$ be a node enabled at step $i$ in a run $r$. Let $(\tau_x, s_x) = r(x), x = i + 1$. If $node(\tau_x) \neq n$, then the only influence that $\tau_x$ can have on the node state $\widetilde{n}$ of $n$ is by appending produced events to one of the input queues, as per the definition of application of a transition. Since $\tau_x$ can't remove events from the input queues in $\widetilde{n}$ and on all input queues was at least one event buffered before the transition was

applied, else $n$ wouldn't have been enabled at step $i$, $\widetilde{n}$ will have at least as many input events buffered on every input as in the step before. This means that $n$ is still enabled at step $x$, and by induction at every later step until a transition with $n$ as its node is taken. □

**Lemma 5: Finiteness of Enabledness.**
*Whenever a node is enabled in a run, it can only be scheduled continously a limited number of times until becoming disabled.*

*Proof.* Let $n$ be an enabled node at step $i$ with the node state $\widetilde{n}$.

First let's assume that no parent of $n$ are scheduled after step $i$ and before $n$ becomes disabled. Since the input queues of $n$ are filled by previous transitions and only finite number of events are produced at every step, all of the queues can only have a finite number of events buffered at step $i$. Since no parents of $n$ are scheduled, the queues can't get fuller. Because all nodes represent functions, and all functions have to consume at least one input event (compare Section 3.10), everytime $n$ is scheduled at least one input queue of $n$ will have one event less buffered after it performed its computation. Therefore the worst case is when $n$ only consumes one event per computation. The maximum number of time $n$ can be scheduled is therefore bounded by the sum of the number of events on all input queues at step $i$.

Now let's assume that also input nodes of $n$ can be scheduled after step $i$. Everytime an input node is scheduled it will add a finite amount of events to one input queue of $n$. This leads to a cyclic behaviour: If an input queue can be scheduled infinitely often, infinite many events will be added to the input queue and $n$ can possibly be scheduled infinitely often. If input queues can not compute infinitely often, only a finite amount of events are added to the inputs of $n$ and therefore $n$ can only be scheduled a limited number of times. Closer inspection of the nature of evaluation engines show why this is no problem: Only if the external trace fed to an evaluation engine is infinte the source nodes can compute infintely often. Hence for finite traces no node can compute infinitely often. Again the worst case is when $n$ only consumes one event per computation. The number of times it can be scheduled is limited by the sum of the number of events buffered on the inputs at step $i$ and the number of events that are produced by inputs of $n$ after step $i$. □

With the notion of runs and especially enabledness we can now define which schedules are seen as fair.

**Definition 18: Fair Schedules.**
*A schedule of an evaluation engine is called fair, if for all runs $r$ it produces the following holds:*

$$\forall i < length(r): \quad \begin{array}{l} \textit{if } n \textit{ is enabled at step } i \\ \textit{then } \exists\, j \geq i \textit{ such that n is scheduled at step } j \end{array}$$

*In other words, every enabled node is scheduled after a finite number of steps.*

Building on this we will investigate different fair schedules in the next chapter.

# 4 Behaviours of Evaluation Engines

Based on the definitions in Chapter 3 we will now look at different schedules of evaluation engines and compare them. This is done in multiple steps: starting with a small subset of allowed schedules and functions and iteratively adding more complex cases.

For the comparision we will use timed transducers from Section 3.3. To do this the notion of a run of an evaluation engine is not sufficient: transducers describe a relationship between inputs and outputs, runs describe stepwise generation of internal and output events. Therefore the *behaviour* of a run is defined, which maps a run to relationship between inputs and outputs.

**Definition 19: Behaviour of a Run.**
*Let $r$ be a run of an evaluation engine. The behaviour $\beta_r$ of it is a timed transducer: A set of tuples of timed sequences. It is calculated as follows:*

1. *Let $\beta_r = \emptyset$ and $r_p$ an empty prefix of $r$.*
2. *Remove the prefix from $r$, where the first transition consumes an input upto but not including the next transition where an input is consumed, and append it to $r_p$.*
3. *Select the sequence of all output events $O_p$ (which is possible empty) that are produced at any step in $r_p$.*
4. *Select the sequence of all input events $E_p$ that are consumed at any step in $r_p$.*
5. *Add the tuple $(E_p, O_p)$ to $\beta_r$.*
6. *Goto step 2 if $r$ is not empty, else terminate.*

*Stated simple the run is chopped into pieces, where each piece begins with the consumption of an input events and ends before the next input is consumed. The pieces are then merged from left to right: First take all inputs and outputs consumed and produced upto the current piece and add them to the behaviour, then merge the current piece with the next and repeat.*

**Example 9: Construction of a Behaviour.**
*In Table 4.1 it is shown how a behaviour is built from a run. The run ist denoted only by its transitions, events are labeled based on their type: $e_i$ are external (read: input)*

| | |
|---|---|
| Run | $(\{e_1\},\{i_1\})\ (\{i_1\},\{i_2\})\ (\{i_1,i_2\},\{o_1\})\ (\{e_2\},\{i_3\})\ (\{i_2,i_3\},\{o_2\})\hookleftarrow$ $(\{i_3\},\{o_3\})\ (\{e_3\},\{i_4\})\ (\{e_4\},\{i_5\})\ (\{i_4,i_5\},\{o_4\})$ |
| Tuples | $(e_1,o_1)\quad (e_1e_2,o_1o_2o_3)\quad (e_1e_2e_3,o_1o_2o_3)\quad (e_1e_2e_3e_4,o_1o_2o_3o_4)$ |

**Tab. 4.1:** Example how the behaviour of a run is constructed

*events, $i_i$ are internal events and $o_i$ are output events. Parts of the run in the same color are the transitions that end up in the same piece when applying the constuction algorithm. The tuples show how the sequence of input and output events from the pieces are extracted. The behaviour of the run is the set of the tuples.*

The behaviour of a run is a timed transducer since all events have timestamps and all consumed events are strictly ordered by their timestamp, since inputs to an evaluation engine are required to be ordered by their timestamp.

Since the behaviour encodes the relationship between inputs and outputs a run produces it provides the foundation to reason about equivalence between different runs and whole evaluation engines.

**Definition 20: Equivalence of Runs.**
*Two runs are called equivalent if their behaviour is observational equivalent.*

Now we can define when two evaluation engines are called equivalent based on their runs

**Definition 21: Equivalence of Evaluation Engines.**
*Two evaluation engines are called equivalent, if for every run that one can produce there is an observational equivalent run in the other.*

## 4.1 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 and will emit exactly one event at every computation, either a normal or a progress event. 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 3.9. 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.

Let's recap and summarize how an evaluation engine performs its computation. 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 enabled node is scheduled to perform its 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.
- 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), which is guaranteed by the enabledness criteria. In the following evaluation engines are classified by their scheduling approaches.

### 4.1.1 Greedy Evaluation Engines

The first class of evaluation engines are called greedy.

**Definition 22: Greedy schedule.**
*A schedule of an evaluation engine built by the following steps is called* greedy.

1. *Select all nodes that are no sources, let their count be* $i$
2. *Label them with unique natural numbers from* $[1, i]$ *in reverse topological order*
3. *Label the remaining nodes with unique natural numbers bigger than* $i$
4. *Schedule the enabled node with the lowest label first*

*We also call an evaluation engine greedy if we mean it's run with a greedy schedule.*

Obviously for many DAGs there is no unique reverse topological order, therefore one can be chosen by the evaluation engine. We will show in Section 4.2.1 that all topological orders will produce observational equivalent behaviours.

The greedy 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 4.1.1 any schedule built like this is fair.

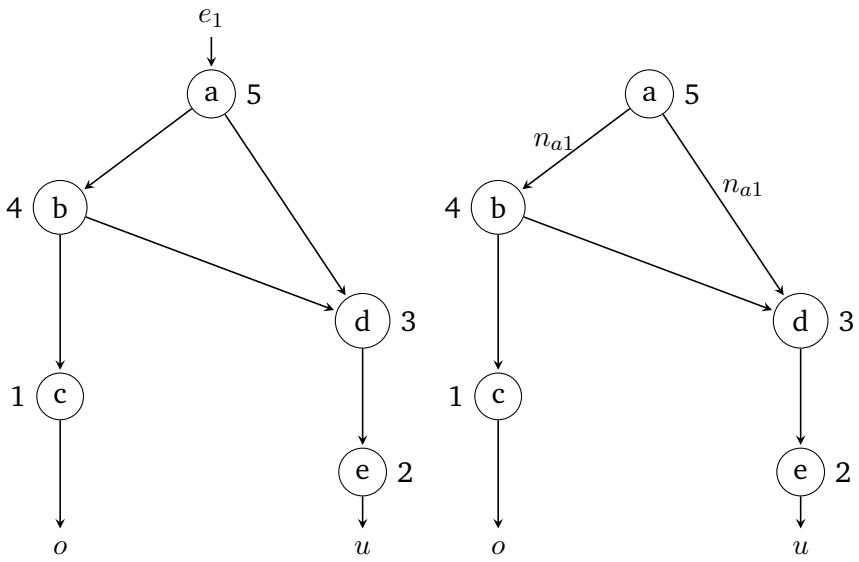


**Fig. 4.1:** Visualization of a simple evaluation engine with a greedy schedule.

Greedy evaluation engines offer a good start to reason about behaviours and will be used as a comparision for all other evaluation engines.

**Definition 23: Valid Evaluation Engines.**
*An evaluation engine is called* valid *if it is equivalent to a greedy evaluation engine.*

Figure 4.1 visualizes a greedy 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 two output streams. 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 its children: nodes $b$ 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 $b$ is scheduled, it would produce the internal event $n_{b1}$ which would then be distributed to nodes $c$ and $d$.

The complete run of the greedy engine for one input is the following, where the states are omitted:

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

If there were more than 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.

#### Fairness of Greedy Schedules

It remains to show that greedy schedules are fair.

**Lemma 6: Greedy Schedules are Fair.**
*Any greedy schedule is fair.*

*Proof.* Let $a$ be a node with the label $n$, which is enabled at step $i$ and is no source. Because evaluation engines can only contain a finite number of nodes there can only be a finite number of enabled nodes with a smaller label than $n$. Because of Lemma 5, all nodes with a smaller label than $n$ will become disabled after a finite number of steps. Let that number be $j$. The only way new events could enter the system are through sources, but they have bigger labels than $n$, as by the definition of the schedule, and therefore can't be scheduled before $n$. Because of Lemma 4, $a$ will still be enabled after these steps. So $a$ is the enabled node with the lowest label at step $i + j$ and therefore will be scheduled.

Now let $a$ be a source. Sources are only scheduled when no internal node is enabled since they are labeled with higher numbers than all internal nodes. Based on the same reasoning as in the first case at some point all internal nodes will become disabled, therefore a source node has to be scheduled. This source can either be $a$ or another source, recall that only one source is enabled at any time because of the environment of an evluation engine. If another source was scheduled, after a finite amount of steps all internal nodes will have to become disabled again. Since finite traces are evaluated at some point either the trace will end without ever feeding an input to $a$, then $a$ will never be enabled, or at some point $a$ will receive an external event. When $a$ receives an external event, it will be the only enabled source, else no input would be fed to an input at that step. Therefore $a$ will be scheduled the next time no internal nodes are enabled. □

### 4.1.2 Fair Evaluation Engines

Obviously greedy schedules are only a small subset of all fair schedules. As the next step we will look at the rest of them.

**Definition 24: Fair Evaluation Engines.**
*A fair evaluation engine is one with a fair schedule.*

In contrast to a greedy evaluation engine a fair one has no fixed schedule, meaning that at each step any enabled node can be scheduled. 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.

The difference between greedy and fair schedules are similar to the ones of synchronous and asynchronous transducers: A greedy schedule will ensure that outputs are produced as fast as possible while a fair can *delay* the outputs by consuming multiple inputs first and scheduling internal nodes multiple times before scheduling an output node. But note that there is an important difference between synchronous transducers and the behaviour of a greedy evaluation engine: A greedy evaluation engine can produce multiple events at every step.

## 4.2 Equivalence of Different Schedules Without Timing Functions

The behaviour of a run of an evaluation engine with a given schedule allows us to reason about equivalence.

As by Definition 23 any evaluation engine has to be equivalent to a greedy one to be valid.

The equivalence is shown in two steps: first in Section 4.1.1 it is shown, that all possible greedy engines for a specification are equivalent, so there is only one valid evaluation for a specification over a fixed input. Afterwards in Section 4.2.2 it is shown that any fair evaluation engine is equivalent to a greedy one.

### 4.2.1 Equivalence of Greedy Systems

When given a series of input events, two greedy 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 4.1.1. 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 it is shown that two runs $r_1$ and $r_2$ of two evaluation engines based on the same graph but with a different greedy schedule can always be reordered to become closer while preserving observational equivalence. If such

a closer run always exist, we will show that the run with closeness $(l, 0)$ to $r_1$, which has to be $r_1$ itself, is also observational equivalent to $r_2$.

**Theorem 1: Equivalence of Different Greedy Evaluation Engines.**
*Two greedy evaluation engines for a specification with different schedules are equivalent.*

*Proof.* Let $r_1, r_2$ be the runs of two greedy evaluation engines for the same specification which received the same inputs. 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 observational equivalent to $r_2$. If $r_1$ is in this set, we would be done. Let's assume that $r_1$ is not in the set. Therefore all runs in the set have to have a smaller closeness than $(l, 0)$ to $r_1$, since the only run with a closeness of $(l, 0)$ to another run is the run itself. Select one run $r_2' \in [r_2]$ which has the biggest closeness to $r_1$. 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 $\tau_1$ for $r_1$ and $\tau_2$ for $r_2'$. Run $r_2'$ will take transition $\tau_1$ 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 $\tau_2'$ must also be taken in the first run as per Lemma 4, it's not possible, that it was taken before $\tau_1$, beause then the two runs wouldn't have been the same upto the point where $\tau_1$ was taken. Therefore $\tau_1$ has to be independent of $\tau_2'$, and because $\tau_2'$ was scheduled by the second run before $\tau_1$ both transitions are independent of each other.

As of Lemma 3 which one of them is taken first won't change the rest of the run at all after both transitions were applied and the run stays valid. Therefore there is a valid run $r_2''$, which is equal to $r_2'$, except that the transitions $\tau_1, \tau_2'$ are scheduled the other way around. See Figure 4.2 for a visualization of the runs. Dotted edges represent multiple transitions. The visualization shows how the three runs *branch* and *merge* after certain steps. The run $r_1$ is marked in red, $r_2''$ in green and $r_2'$ in blue. As you can see the run $r_2''$ has a bigger closeness to $r_1$ since it takes $\tau_1$ before $\tau_2'$. Note that $r_1$ is only known upto step $d$, especially the transitions $\tau_2, \tau_2'$ will be taken eventually, but it isn't known when or in which order.

When two adjacent transitions are exchaned besically two things can happen:

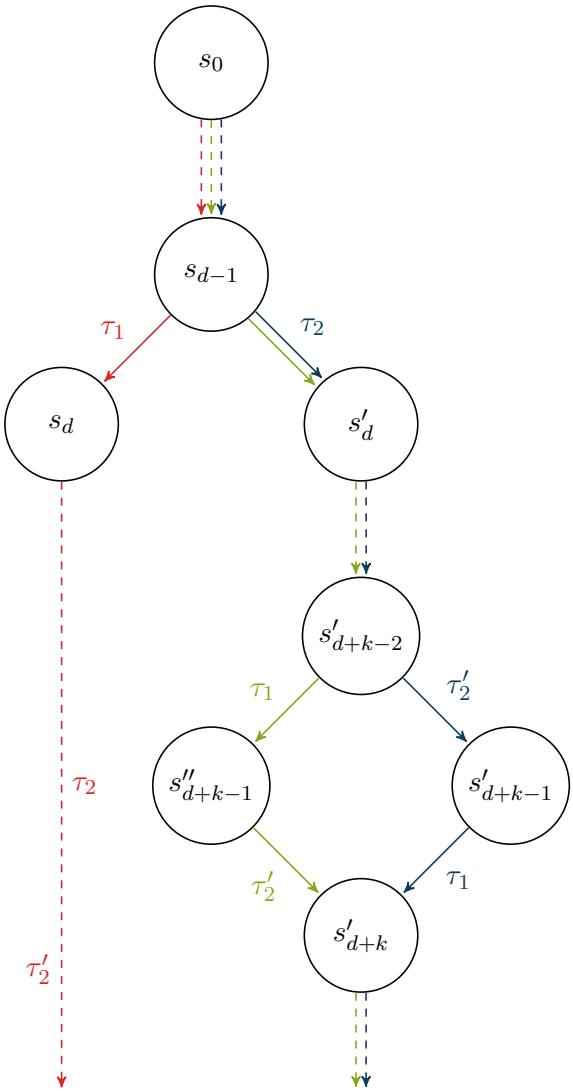


**Fig. 4.2:** Visualization of three runs of an evaluation engine as explained in Section 4.2.1.

- Outputs can be produced later, maybe moving them to the next piece in the contsruction of a behaviour (this happen when a transition producing outputs is exchanged with the next transition which consumes an external event)
- Outputs can be produced earlier, which can move them to an earlier piece in the construction of the behaviour (this happens in the opposite case, where a transition consuming an external event is pushed before one that produces outputs)

While the order of outputs in the tuples of the behaviour will change when two transitions producing outputs are exchanged, observational equivalence isn't influenced, since it is defined over reordered outputs.

Let's test if $r_2''$ is observational equivalent to $r_2'$: We will compare how the behaviour of the run $r_2'$ changes when the transition $\tau_1$ is exchanged with $\tau_2'$. The comparision is based on the different cases the transitions can encode. Some of the cases are not feasible, they are listed for the sake of completeness and to explain why they can't happen. The cases are:

1. No inputs are consumed and no outputs produced by either transition. This obviously won't change the behaviour at all.
2. $\tau_2'$ consumes an input and doesn't produce an output and
   a) $\tau_1$ doesn't consume an input and doesn't produce an output. This won't change the behaviour, since $\tau_1$ didn't added anything to it in the first place.
   b) $\tau_1$ doesn't consume an input, but produces one or more outputs. This changes the behaviour, since $\tau_1$ is now part of the piece starting before $\tau_2'$. Therefore the produced outputs are now part of one more tuple of the behaviour. But note that it still is part of all tuples built from the later pieces of the run. Therefore the two behaviours are still observational equivalent: All tuples in the behaviour are still the same, except that one or more output events are produced one step earlier, which doesn't hurt observational equivalence.
   c) $\tau_1$ consumes an input but doesn't produce an output. This case can't happen, since inputs to evaluation engines are totally ordered by their timestamp and therefore $\tau_2'$ couldn't have been scheduled after $\tau_1$ in $r_1$ and before $\tau_1$ in $r_2$.
   d) $\tau_1$ consumes an input and produces outputs. This case can't happen for the same reason.
3. $\tau_2'$ produces one or more outputs and doesn't consume an input and

a) $\tau_1$ doesn't consume an input and doesn't produce an output. This won't change the behaviour, since $\tau_1$ didn't added anything to it in the first place.

b) $\tau_1$ doesn't consume an input, but produces one or more outputs. This only changes the order of the output events in the behaviour, but since they are reordered by their timestamp and the order of events with the same timestamp isn't important for observational equivalence, the new run is still observational equivalent.

c) $\tau_1$ consumes an input but doesn't produce an output. This will *delay* the production of the outputs from $\tau_2'$ by one piece of the chopped run. While this changes the behaviour it preserves observational equivalence.

d) $\tau_1$ consumes an input and produces outputs. This is kind of a combination of the previous two cases. The outputs from $\tau_2'$ are *delayed* by one piece and the outputs of $\tau_1$ are now produced before them. But still all outputs are produced, only in different order and maybe one step later. Therefore observational equivalence holds. Also note that such a transition isn't very useful as argued in the next case.

4. $\tau_2'$ produces one or more outputs and consumes an input. First of all note that this is a rather made up combination that can only happen when a source is an output node at the same time and therefore doesn't have much of a purpose. But for the sake of completeness let's look at the cases following from this:

   a) $\tau_1$ doesn't consume an input and doesn't produce an output. Again won't change the behaviour, as in earlier cases.

   b) $\tau_1$ doesn't consume an input, but produces one or more outputs. Preserves observational equivalence since the outputs of $\tau_1$ are only produced one step earlier than before.

   c) $\tau_1$ consumes an input. This case can't happen as reasoned in Case 2c.

So for all cases that can happen, the run which is obtained by exchanging the two adjacent transitions is observational equivalent to the run without the change. The exchange of the two transitions brings the closeness of the new run by construction one step closer to $r_1$. This means, since observational equivalence is transitive, that there is an observational equivalent run to $r_2$, the run $r_2''$, which has at least the closeness $(d, 1)$.

If $k = 1$ the transition $\tau_2'$ from the previous case is equal to $\tau_2$. The reasoning for all cases stays exactly the same, in the end we will obtain a run $r_2''$ which is observational equivalent to $r_2'$ but 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 observational equivalent to $r_2$.

Combined we can now say, that there is no upper bound on the closeness of observational equivalent runs of $r_2$ to $r_1$, therefore the run with the closeness $(l, 0)$ also has to be equivalent to $r_2$. And as already stated, only the run $r_1$ can have the closeness of $(l, 0)$ to $r_1$. Therefore $r_1$ has to be observational equivalent to $r_2$. □

This characteristic of greedy schedules gives us a baseline to compare other schedules to. Since all greedy schedules of an evaluation engine produce observational equivalent behaviours we can choose any run produced by such a schedule and compare any other run to it. In the next section we will do this for all fair schedules.

### 4.2.2 Equivalence of Greedy and Fair Evaluation Engines

Let's recap what fairness of a schedule means: Whenever a node becomes enabled in a run it has to be scheduled eventually. What makes fair schedules harder to reason about than greedy ones is that for one they don't have to be deterministic and furthermore that it's possible that an enabled node is not scheduled for a very long time.

Before reasoning about equivalence of greedy and fair schedules let's look at a kind of fair schedule that can be seen as worst case. Basically it is the reverse of a greedy schedule: Always schedule the enabled node that is closest to a source. Note that this schedule is not fair for infinite input traces. Stated simple this schedule will consume all input events and produce all output events per *level* of the DAG, starting at the sources and moving towards the outputs. The behaviours of runs with such a schedule are pretty special: Since no output is produced before all external events are consumed (except if a source is also an output) only one tuple of the behaviour will contain any outputs, to be specific the one which contains the sequence of all inputs as the first element. An abbreviated example of such a run is $(e_1, ())\ (e_1e_2, ())\ (e_1e_2e_3, ())\ (e_1e_2e_3e_4, o_1o_2o_3o_4o_5)$. Such a run needs obviously more reordering than a greedy one to become observational equivalent to another greedy one since greedy schedules try to produce outputs as early as possible.

This example shows what the difference is when reordering a fair run in contrast to a greedy run: basically more transitions have to be reordered since outputs can be produced later.

Let's revisit the cases from Section 4.2.1: Actually the Cases 2b and 4b can't happen for greedy runs. If in a run $r_1$ a transition $\tau_1$, which produces an output, has to be exchanged with an earlier transition $\tau_2$, that consumed an external event, to become closer to a greedy run $r_2$, $r_1$ couldn't have been greedy in the first place: Greedy schedules ensure by construction that all outputs that can be produced based on consumed external events are produced before the next external event is consumed. If $\tau_1$ happened before $\tau_2$ in a greedy run, the outputs from $\tau_1$ can be produced without consuming the external event from $\tau_2$ before, hence a run in which $\tau_2$ happens before $\tau_1$ couldn't be produced by a greedy schedule.

It's noteworthy that actually the whole proof of Theorem 1 doesn't depend on the fact, that the runs are produced by greedy schedules. The only real requirement is fairness, meaning that all transitions that can happen will eventually happen. Therefore the proof does hold without change for Theorem 2.

**Theorem 2: Equivalence of Fair and Greedy Evaluation Engines.**
*Any fair evaluation engine is equivalent to a greedy evaluation engine for the same specification.*

## 4.3 Timing functions

## 4.4 Parallel computation

# 5 Implementation Details

Besides the theoretical basics presented in Chapter 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 the 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 the runtime can consume them.

## 5.1 TesslaServer

The runtime to evaluate specifications against traces is implemented in the programming language Elixir, which itself is built on top of Erlang, the Bogdan/Björn's Erlang Abstract Machine (BEAM) Virtual Machine (VM) and Open Telecom Platform (OTP). To understand why this platform was choosen we will look at the Erlang ecosystem in the next section. Note that in the following sections terminology from Chapters 2 and 3 as well as from Erlang and Elixir is heavily used.

### 5.1.1 Erlang and Elixir

Erlang was originally developed 1987 as a language to program systems with limited resources which had to be highly fault tolerant. The primary purpose of the language were phone switches, which have to handle large amounts of connections at the same time. Since the switches weren't deployed at a central location but wherever they are needed crashes would entail long downtimes of the system. Also, since customers expect permanent service, the platform had to provide a way to update the software without downtimes. With these requirements the language and the OTP platform were developed.

While the requirements of TesslaServer are quite different we will see that the Erlang platform is a great fit for the implementation.

The rather new programming language Elixir[1] can be seen as a dialect of Erlang. Elixir code is compiled into bytecode for BEAM and can therefore interoperate with Erlang code. The rationale to use Elixir instead of Erlang is twofold: On the one hand Erlangs syntax is pretty different from that of most modern general purpose programming languages while Elixirs syntax was developed based on modern language design principles. Also Elixir supports metaprogramming, meaning you can write code that generates code at compile time, which is heavily used as described in Section 5.1.2.

One of the core strengths of the Erlang platform is it support to use multiple processor cores, even if these cores are deployed over multiple machines in a network. The platform offers tools to develop code that can be distributed over multiple processors. This distribution is transparent to the developer. The underlying concept of the distribution is the actor model, first introduced in [**Hewitt1973**].

An actor is basically a self contained entity, that holds a state and can receive and send messages to other actors. Since an actor holds its own state and is the only one that can manipulate it, an actor can be scheduled on any core as long as the runtime guarantees transparent message delivery. When an actor receives a message from another actor the BEAM VM will eventually schedule the code responsible for handling the message on an available core. This code can then access the state, alter it and send a response to the sender of the message. In this sense an actor can be seen as a state machine, which alters the state everytime it receives a message. Since actors are independent of each other they can be scheduled in parallel on multiple processors. Only when two actors synchronously communicate one has to wait for the other.

Another reason to choose the Erlang platform was its support for multiple plattforms, including resource restricted ones. While this thesis only considers offline monitoring it may be a future goal to perform online monitoring with TesslaServer. To enable this the runtime has to be able to run on the same hardware architecture the monitored program runs on sharing resources with it. An example of the variety of the supported platforms of Erlang and Elixir is the Nerves project[2] which allows developers to build embedded software.

[1] `http://elixir-lang.org`
[2] `http://nerves-project.org`

### 5.1.2 Architecture

As described in Section 3.9 TESSLA specifications form a DAG of nodes, which perform transformations on streams and send their computed streams to children nodes. Streams can be seen as a sequence of events or changes that can be represented as messages between the nodes. This form of specification can be easily implemented as an actor based system, where each vertex in the DAG is implemented as an actor and the communication between adjacent vertices is realised with message passing. Interestingly this architecture enables the simulation of greedy schedules described in Definition 22 while the Erlang runtime itself only guarantees a fair scheduling. To understand how this work the details of message passing and handling in Erlang is important. Basically there are two ways two Erlang processes can communicate with each other through messages: synchronous and asynchronous. To send a message synchronous the *call* Application Programming Interface (API) is used. A *call* will send a message to another process and block until a response is received. The *cast* API is the asynchronous counterpart, which will send a message and immediately passes back control to the process that used it.

It can be easily seen, that the usage of the synchronous API will lead to a valid greedy schedule, since the sources in the DAG will have to wait for their children to finish and the children will transitively have to wait for their children. This means that each new message gets distributed through the whole graph as fast as possible. While this behaviour is an interesting observation the actual implementation uses the asynchronous API to take better advantage of parallel evaluation.

An important characteristic of TESSLA specifications is, that they can specify properties targeting realtime characteristics. On the one hand this enables specifications that aren't feasible with more classical specification approaches, like Linear Temporal Logic (LTL) or LOLA, which work on synchronous streams, on the other hand it adds complexity to monitoring approaches, since it adds asynchronicity to all parts of the system. One point where this is important is in the way, how systems have to be monitored, or more precisely how their generated events are encoded. For synchronous monitoring approaches, the information, that an event happened is sufficient, for asynchronous ones it is important to know at which exact time each event happened. For an implementation of an asynchronous monitoring approach this simply means, that the representation of events has to include information about the time of the event. Another consequence of the asynchronous nature is discussed in Section 5.1.4 with the notion of the *front* of events.

```
{
  "type":"java.util.Collections$UnmodifiableSet",
  "items":[{
    "id":1,
    "nodetype":"TesslaServer.Node.Lifted.Add",
    "operands":[3,2]
  },{
    "id":2,
    "nodetype":"TesslaServer.Node.Literal",
    "options":{
      "value":5
    }
  },{
    "id":3,
    "nodetype":"TesslaServer.Node.Literal",
    "options":{
      "value":3
    }
  }]
}
```

**Listing 5.1:** Minimal example of the JSON based specification format. The specification describes a DAG with three nodes: two literals as the sources and an adder as their child.

### 5.1.3 Synthesis of the Evaluation Engine

The first step that TesslaServer has to perform to evaluate a specification is to synthesise the evaluation engine that will consume the traces and perform the computation. For this step a specification is compiled into a JavaScript Object Notation (JSON) based format that describes the nodes and their relationship. Listing 5.1 shows a minimal example of the format, which includes three nodes: two literals and a node adding their value.

The compiler performs multiple checks, e.g. type checks and ensures that no loops are present in the specification, and transformations, e.g. resolve macros and other syntactic elements of the specification language. Since the compiler acts as a safety guard, TesslaServer assumes that a given specification is error free and performs no redundant safety checks. Invalid specifications can therefore lead to all kinds of wrong behaviour if fed to TesslaServer.

The JSON based specification is then translated into actors as follows: For each node described in the *items* object an actor is started with the Elixir module specified by the *nodetype* key as the message handling code. When a node is started as an actor it will receive the additional information present in the JSON object, e.g. the value for the *operands* keys, as an argument to its initialization handler. It will use this

information to build up the initial state and to register itself with a central process registry provided by erlang under its $id$.

After all actors are started, TesslaServer will send each actor a message asking them to subscribe to their operands. When a node subscribes to an operand it will send the operand a message containing the nodes $id$ with the request to add this node as a child. The node representing the operand will add the $id$ the list of children in its internal state. Later, during the actual evaluation of the traces, each node will use the list of children to send messages of new generated events using the central process registry.

The evaluation engine is synthesized, when all nodes subscribe to their operands and can start to evaluate the specification over traces. To understand how the evaluation works, the next section will explain the implementation of nodes.

### 5.1.4 Node Implementation

*Nodes* (or *computations* due to namespace errors with the Erlang standard library) are responsible for the actual evaluation of a specification over traces. TESSLA defines a standard set of nodes (called functions in the original specification) but leaves it open to the runtime to support only part of them or extend it. To support extensions of TESSLA and the runtime one of the main focuses was to make it easy to implement new node types.

This is achieved by building upon an abstraction from OTP called *GenServer* and providing a new abstraction which is tailored towards implementing a node for stream transformation, called *GenComputation*. Figure 5.1 shows the control flow of a single node with two inputs and one child node.

The *GenServer* abstraction is provided by the Erlang and Elixir platform and is basically an implementation of the actor pattern mentioned before. It provides an API to register actors, send messages to other actors and to handle incomming messages as well as maintaining the actor state. Furthermore it enables monitoring, crash recovery and hot code upgrades, mechanism that aren't used in TesslaServer as of now.

The next layer is provided by the *GenComputation* abstraction. Before examining its responsibilities we will look at it's implementation. *GenComputation*, similar to *GenServer*, heavily relies on Elixir metaprogramming, which is achieved with macros. Elixir makes heavy use of macros, a mechanism mainly known from the LISP programming language. Macros enables the programmer to write code that generates code. Since all nodes perform a similar task, performing computations

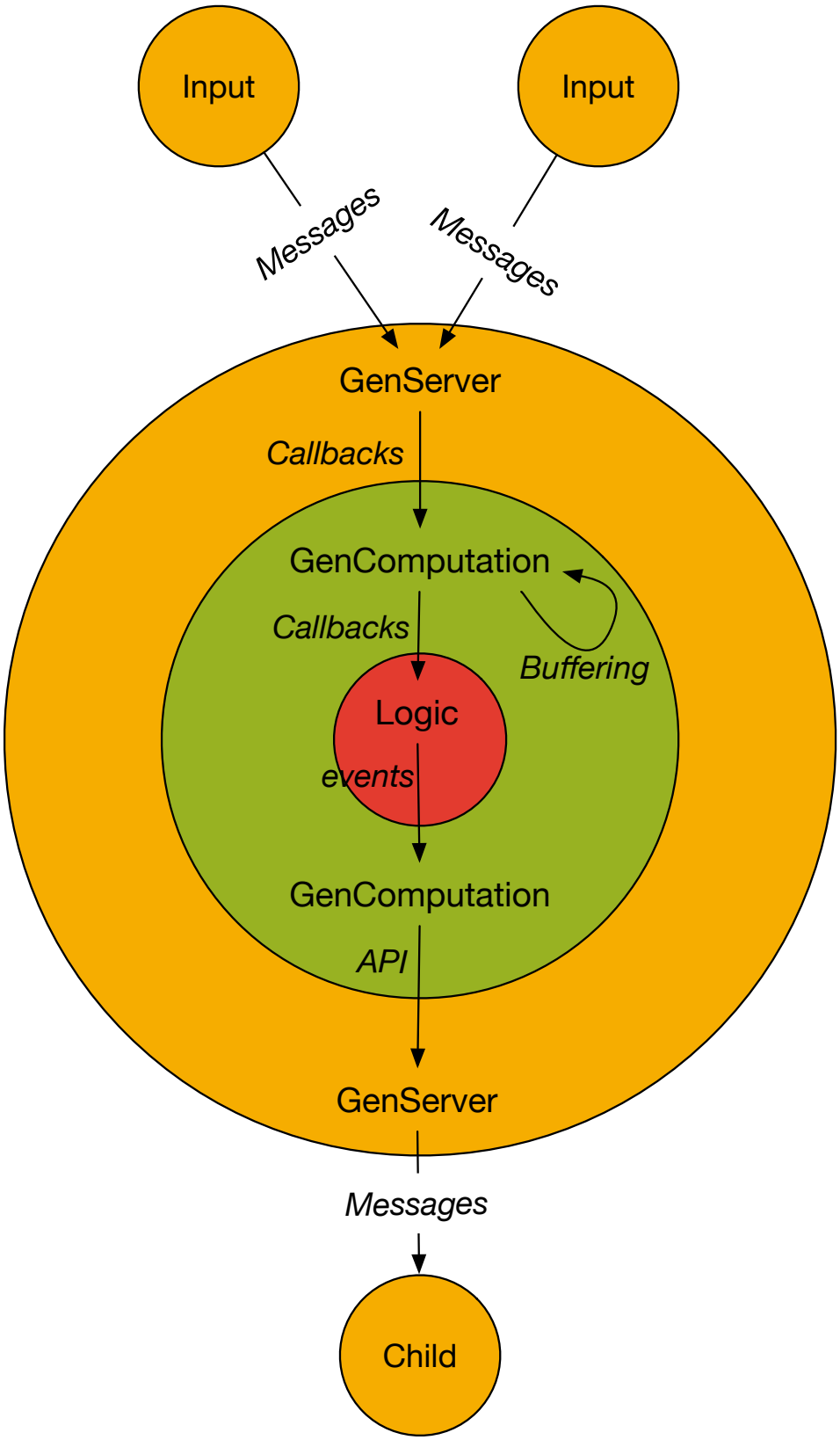


**Fig. 5.1:** Schema of the control flow of a node. A node interacts with other nodes using the *GenServer* abstraction, buffers events using the *GenComputation* abstraction and performs its calculation in the *logic* part. Computed events are then send to children using the same abstraction mechanisms.

on one or more input streams, it makes sense to generate code for shared responsibilities instead of duplicating it for every node. In this special case macros are used to achieve similar goals as inheritance in object oriented programming languages, namely code reuse.

Let's now focus on the responsibilities of *GenComputation*. When a node receives a message from another node, the message will be handled by the *GenServer* abstraction. *GenServer* itself requires the user to implement a callback method which is responsible for actually handling a received message, e.g. by updating the state or sending new messages. Since TesslaServer uses exactly one format for messages *GenComputation* is able to handle them for all actual node implementations in a general way and only invoke actual node logic when needed. The concrete handling of messages differs beween the two versions of TesslaServer, therefore the details are described in Sections 5.1.5 and 5.1.6.

On a high level *GenComputation* buffers messages until it can be sure, that all inputs have progressed since the last time outputs were generated by the node, which means that either new events can be generated or at least the output stream can be progressed. A similar concept is presented in [**TODOBEEPBEEP**] which was discussed in Section 2.6: the *front* of the input streams. Because BeepBeep isn't using timed specifications the computation of the front is easy: the head of each input when every input has at least one event buffered. For TeSSLa this is not as easy, since the timestamps of the events in the front can differ. As a result *GenComputation* has to perform multiple steps to determine the appropriate actions to take based on the events of the front. The exact steps are described in the respective sections of the two versions of TesslaServer. When *GenComputation* has determined that at least one new transformation can happen it invokes the actual node logic by calling a callback method that each node has to implement.

### 5.1.5 TesslaServer V1: Stream passing

The first version of TesslaServer was built with two goals in mind: safety and hiding complexity. This led to implementation decisions that were on one hand performance critical (see Section 6.1) and on the other hand made it hard to implement complex node types.

One of the main ideas of this implementation is to make streams a central data structure that is able to guarantee some safety aspects, e.g. ordering of events. Each node has a set of streams: one for each parent node and one output stream. Whenever a node computes new events, its output stream is updated and the whole stream is send to all of its children. When a node receives a message from a parent node containing an updated stream, it will update its own state by replacing the

last saved stream from the parent with the newly received. Then it will determine if with this updated stream new events can be computed. To do so it looks at the input stream with the minimal progress and compare it with the progress of its own output. If theese two times are equal the node can't produce a new front, since at least one input hasn't progressed since the last computation. Only if the minimal progress of all inputs is bigger than the progress of the output the node has to determine the appropriate computations.

This happens by generating a sequence of partial fronts for specific timestamps like following:

1. Look at all input streams and find all events that happened after the progress of the output upto the minimal progress.

2. Take the timestamps of theese events in chronological order, we will call them the *change timestamps* since they denote that at least on one input stream something changed.

3. Iterate over the timestamps in order, built partial fronts by getting all events that happened on any input stream at that timestamp.

4. Invoke the actual logic of the node for each partial front to perform its transformation.

5. Add the generated events to the output stream and send the updated output stream to all children.

It is important to understand, that all steps except Step 4 are performed by the *GenComputation* abstraction. Hence, a node implementation, at least in theory, only has to implement the logic to combine a partial front to a new output.

The problems of this approach are twofold: First, to implement more complex node types it was necessary to overwrite a lot of the provided abstractions, e.g. to manipulate timestamps. But more important were scaling problems: Since every node had a copy of all input streams stored in its state and streams contained all events that were ever produced the random access memory (RAM) usage did grow exponentially with thenumber of nodes and input events used to evaluate a specification. This can be seen in Section 6.1.2. Another problem was that the messages between nodes also grow with the number of events, since the whole stream has to be send everytime.

Therefore the TesslaServer version 2 architecture was implemented.

### 5.1.6 TesslaServer V2: Event passing

The second version used the insights of the first version to provide better abstractions. Scalability and making complex node types possible to implement were the main goals of the architecture.

To achieve theese goals some changes has to be made: Streams are no longer an explicit data structure in the system but mere an attribute of events to denote on which stream they happened. At the cost of some safety guarantees, which explicit streams provided, a simpler and clearer control flow of nodes and a very small API of the new version of *GenComputation* was achieved.

In the new architecture simple node types have to implement more logic, since they have to decide how to handle progress events, which became necessary in the new architecture to propagate that a stream has progressed to a new timestamp without an event happening on it. In the old architecture the *GenComputation* abstraction handled theese cases for all nodes, which wasn't appropriate for some node types. To avoid too much code duplication a new abstraction *GenLifted* is provided, which can be used as the building block for node types that *lift* a function, which normaly would run on two values, to run on two signals. This approach avoids the problems of the old architecture by moving concerns out of the base *GenComputation* abstraction and making it optional to use the new *GenLifted* abstraction.

The new approach to sending messages between nodes is to send each generated event as one message. This will lead to an overall increase in messages but simplifys the handling of them. With the new architecture nodes have a buffer for each parent node in their state. In this buffer all events received from that parent are saved, that weren't part of a front upto that point in time. The process of handling new messages and computing the partial fronts implemented in the new version of *GenComputation* is the following:

- Add the newly received event to the end of the buffer that stores events of that parent node.
- Test if on each buffer at least one event is stored.
- End if at least one buffer has no events.
- Else determine the minimal timestamp over the first events of all buffers.
- Remove all events from the head of the buffers with that exact timestamp, they form a partial front.
- Invoke the actual logic of the node for the partial front to perform its transformation.

- This will generate at least a new progress event or one or more normal events, send them to all children of the node.
- Go to Step 2.

Note again how only at Step 5.1.6 the actual node logic is invoked, meaning only that part has to be implemented for each new node type. Nonetheless this procedure adds more responsibilities to the programmer of new node types: In the old approach the actual node logic didn't have to handle progress events and caching of events that are important for future computations. This can be described as the concept of making complexity explicit: The implementation has to actually handle complex edge cases in contrast to the old approach which tried to hide this complexity.

One side effect of the new implementation is that one limitation on input traces is no longer needed: In the first implementation traces had to be totally ordered over all streams, the new implementation works as long as traces are ordered per stream. This is especially useful when using traces that were generated by multithreaded systems: When it can be guaranteed, that each stream in the trace is exclusive to one thread, the generated trace file can be directly fed to TesslaServer. This can easily be achieved by including an unique identifier per thread in the stream identifier.

## 5.2 Instrumentation Pass

While the implementation of the actual runtime was the main goal of this thesis another project was also developed, mainly to provide test traces for the runtime. Since this project uses some interesting technologies and can be extended to support a wide variety of trace data generation, the used technologies will be described in this chapter.

The need for a tool to generate traces arised, when no suitable test data for the runtime could be found. As reasoned in Section 2.7 all trace data that was available wasn't suitable for TeSSLa for a number of different reasons. Therefore a tool was implemented to generate traces tailored towards the needs of TeSSLa. This did open up the opportunity to think about how the ideas from the runtime can be translated into a trace collection tool. One of the central ideas of the runtime is, that it doesn't make assumptions about the platform of the monitored program. While the instrumentation of code with the goal to emit trace data obviously relies on the language the code is written in, the LLVM project provides abstractions that can be used to implement an instrumentation mechanism that doesn't rely on the language the instrumented code was written in.

```
variable_values:write_ptr 843071489 1463991050 176761
variable_values:write_ptr 843071490 1463991050 176832
function_calls:process nil 1463991050 176901
```

**Listing 5.2:** Trace data generated by the instrumentation pass. Each line represents an event and each event consists of four pieces seperated by spaces: a stream name, an optional value and the timestamp in unix format followed by the amount of microseconds since that timestamp.

To understand how this works recall Section 2.7.7 and the way LLVM works: a frontend compiles code from a source language into an Intermediate Representation (IR), perform compiler passes on the IR and then finally compiles it to native machine code. If the instrumentation pass works on the level of the IR it would work for all languages that have a frontend for LLVM.

With the provided C++ API from LLVM to implement compiler passes that can analyse and transform IR code it is easy to implement such an instrumentation. The implementation uses the *ModulePass*[3] base class to analyse whole modules. Modules represent constructs like classes from C and C++ in IR. The pass iterates over the *Functions*[4] the module consists of and checks if the current function is one that should be instrumented. If so it builds a *CallInst*[5], which is the IR equivalent of calling a function. The *CallInst* will call a specified log method provided by a logging library. The instruction is then inserted as the first instruction of the *Function* that is instrumented.

At runtime the instrumented program will then log an event everytime the instrumented function is called. Events generated by the instrumented program will have the format shown in Listing 5.2.

While the compiler pass works and was used to generate authentic test data it is limited and has some challenges remaining. As we will see in Section 6.2 it can add a lot of overhead and interfere with compiler optimizations. As a first measure to reduce the overhead the logging mechanism is implemented on top of a library called *zlog*[6] to buffer generated events. Without this buffer the overhead would be much higher since constant writes of the generated events to an output device would have to happen.

A second measure is that the pass will only instrument functions that are specified by the user when invoking the pass. This makes it possible to generate multiple different instrumented versions of the same code, of which each will only generate traces that are interesting for a certain specification. E.g. one could have two

[3] http://llvm.org/docs/doxygen/html/classllvm_1_1ModulePass.html
[4] http://llvm.org/docs/doxygen/html/classllvm_1_1Function.html
[5] http://llvm.org/docs/doxygen/html/classllvm_1_1CallInst.html
[6] https://github.com/HardySimpson/zlog

instrumented versions of the same code, one which generates events used by a specification to monitor buffer size limits and the other which generates events used by a specification describing performance constraints.

As of now the instrumentation will also produce erroneous traces when the instrumented program is multithreaded and will generate events on more than one stream. Due to race conditions the logged events can then be in the wrong order with respect to their timestamps. For offline monitoring this can be solved by simply reordering the events, for online monitoring this obviously isn't possible. For the second version of TesslaServer the problem only occurs, when different threads can generate events with the same stream, since it doesn't require the inputs to be totally ordered over all input streams. This can be solved by adding a thread unique identifier to the stream name and adapting the specifications to take into account that events might happen on different threads out of order.

As a final limitation the instrumentation pass right now only supports function calls to be instrumented. To do so it would have been enough to implement a *FunctionPass*[7]. That a *ModulePass* was chosen is justified by the fact that the pass could be easily expanded to generate other events as well, for example function returns, variable definitions, variable overwrites and assignments of null values.

[7] `http://llvm.org/docs/doxygen/html/classllvm_1_1FunctionPass.html`

# 6 Evaluation

In this chapter performance characteristics of the implemented runtime and also of the instrumentation pass are presented. To do so both systems are tested under a number of different circumstances to get an overview of how they might perform in real world usage. As the complete benchmark data is to big in this chapter only summarys are presented. The complete datasets can be found in Chapter 8.

## 6.1 Runtime Benchmarks

Over the course of this thesis two different approaches to handling the progress of streams in the runtime are explored, which were also briefly mentioned in Section 3.13: progress timestamps and progress events. Progress timestamps were used first, which worked, but lead to big overhead when a great number of computations had to happen, since the whole stream had to be send to all children of the node that performed the computation. Therefore, later the runtime was refactored to use progress events, so that only events had to be sent between nodes. The following benchmarks will show the performance characteristics of both approaches under different circumstances.

### 6.1.1 Number of Processors

Since one of the main motivations to choose Erlang as the platform for the runtime was its support for parallel execution on many processor cores, we explore the performance characteristic in regard to available cores in this section. To do so a simple TeSSLa specification is given, that includes as many nodes as the maximal number of processors available. Such a spec for eigth processors is given in Listing 6.1.

The specification is evaluated over appropriate traces with a specified amount of processor cores available. The needed time of the evaluation engine from its start until it emits the conclusion, that the stream $done$ is $true$, is measured for different amounts of processors. All benchmarks were performed on the same machine with upto 16 cores and 48GB of RAM. Figure 6.1 shows the results of the benchmarks. The complete data can be found in Tables 8.1 to 8.1.

```
define num_events: Signal<Int> := literal(10000)

define add_calls: Events<Unit> := function_calls("add")
define add_call_sum: Signal<Int> := eventCount(add_calls)

define overhead_0: Signal<Int> := signalAbs(add_call_sum)
define overhead_1: Signal<Int> := signalAbs(overhead_0)
define overhead_2: Signal<Int> := signalAbs(overhead_1)
define overhead_3: Signal<Int> := signalAbs(overhead_2)
define overhead_4: Signal<Int> := signalAbs(overhead_3)

define done Signal<Boolean> := eq(overhead_0, num_events)
```

**Listing 6.1:** TeSSLa specification with eight nodes on the critical path

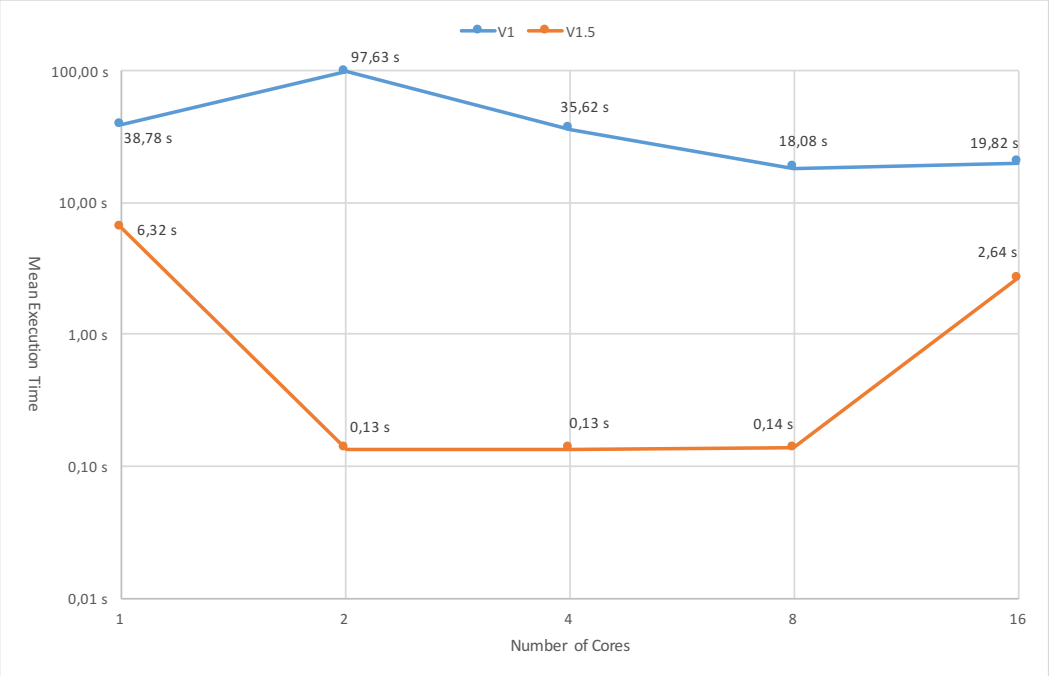


**Fig. 6.1:** Performance of the runtime with different number of used cores. V1 denotes the old implementation, V1.5 the adaption with limited amount of saved events and V2 the new approach.

One thing that was recognized during benchmarking was that the older implementation approach used big amounts of RAM and the usage grows exponentially with the amount of cores used . For example, with two used cores the runtime would use around 2GB RAM, with 4 Cores already over 8GB. This can be explained easily: in the old implementation all data has to be sent between all cores while in the new implementation only the generated events have to be sent around. The excessive usage of RAM even lead to timeout under some circumstances and crashes the whole runtime. The amount of data send between processes can even lead to negative performance impacts when more cores are added.

This behaviour was one of the main reasons to switch to a new approach. To test if reduced RAM usage would lead to better performance and eliminate crashes, the old architecture was changed in a small way: Each stream was limited to only hold the last 20 events, all older events were thrown away. Since the specification used for benchmarking only works on the most recent event of each stream, this would not alter the conclusion of the engine. For example a specification computing the average of the last 21 events wouldn't work with this adaption. See Section 6.1.2 for a benchmark of RAM usage. The obvious reduction of RAM usage of the adapted first approach motivated the decision to refactor the runtime to use the new approach.

TODO: describe how V2 scales with core number.

### 6.1.2 Number of Events

Another metric looked at, is how the runtime behaves in regard to the number of events that are fed to the engine, and therefore how many events are generated during its execution. To obtain these metrics the specification from Listing 6.1 is evaluated with different values for the comparision node are run. The specification is extended to use 16 nodes, which means that each input event will generate 16 internal events. All benchmarks in this section were performed on a processor with 4 cores at 2.4GHz speed and 8GB of RAM.

The first benchmark compares the execution time of the implementation approaches when fed with different number of input events. Figure 6.2 gives an overview of the data, the whole dataset can be found in Tables 8.4 to 8.6.

The number of events also increase memory usage, espacially in the first implementation approach. The second benchmark therefore tests the RAM usage of the different approaches under different number of input events. Figure 6.3 illustrates the RAM usage of the different approaches. The complete data can be found in Tables 8.7 to 8.9.

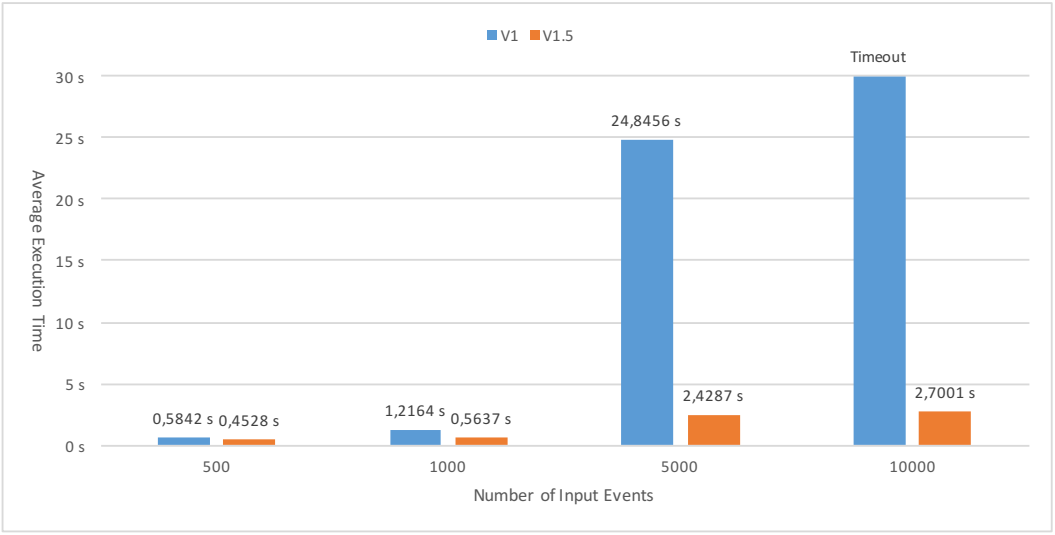


**Fig. 6.2:** Average execution time of the different implementations when fed with different amounts of input events. V1 refers to the old approach, V1.5 to the adapted old approach and V2 to the new approach. V1 has no data for 10 000 events since the enormous RAM usage constantly lead to timeouts.

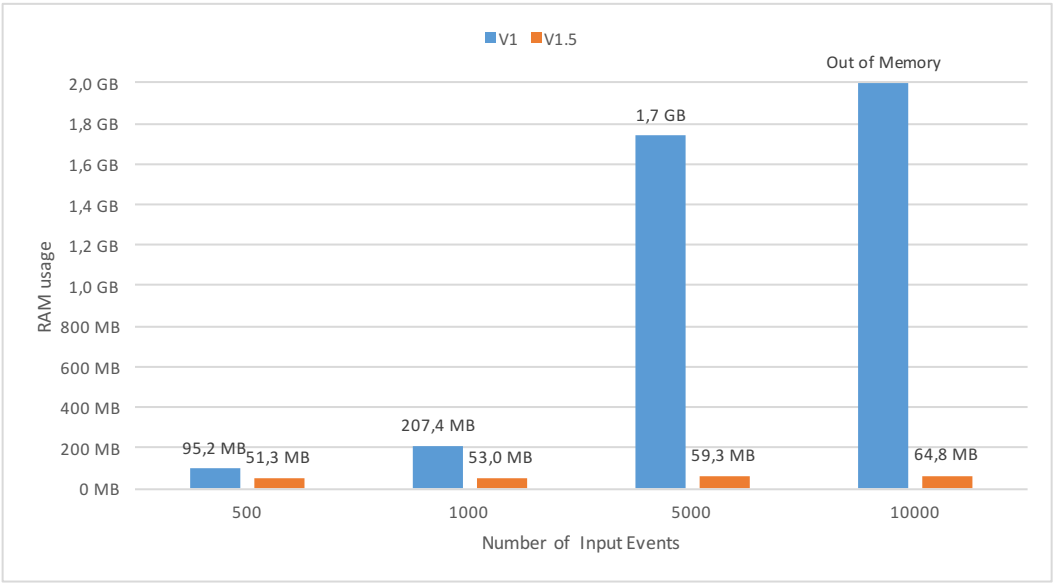


**Fig. 6.3:** RAM usage of the different versions of the runtime in regard to input events. V1 refers to the old approach, V1.5 to the adapted old approach and V2 to the new approach. V1 has no data for 10 000 events since the enormous RAM usage constantly lead to timeouts.

### 6.1.3 Number of Nodes

As a last metric the performance of the runtime is evaluated over specifications with different amount of nodes. Therefore the specification from Listing 6.1 was modified to include higher number of nodes by appending more nodes computing the absolute. All benchmarks in this section were performed on a processor with 4

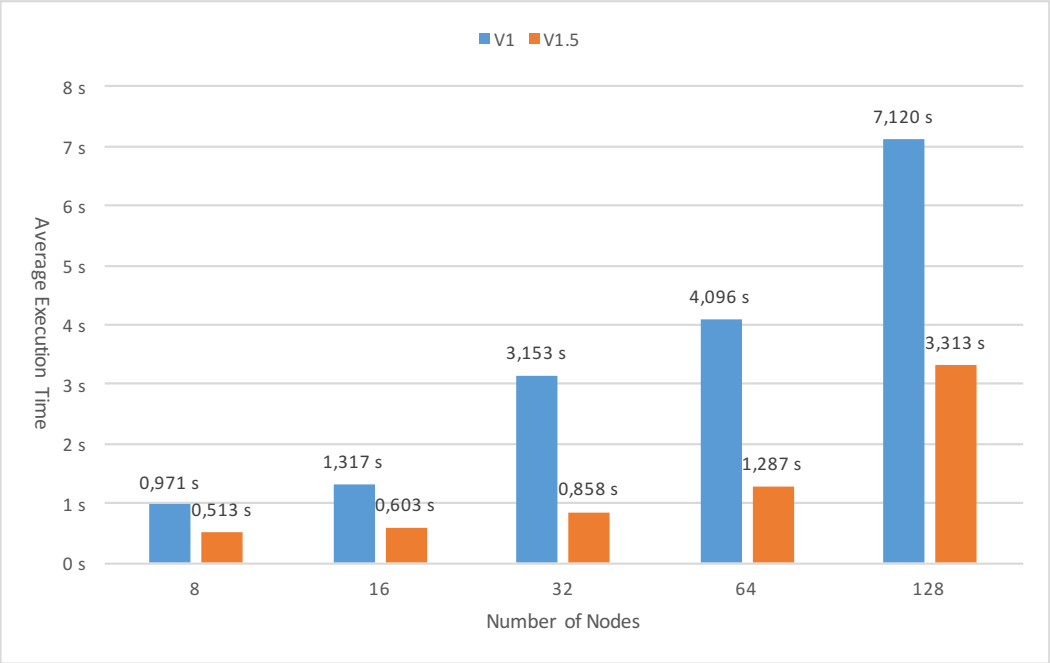


**Fig. 6.4:** Average execution time of the runtime over specifications with different amount of nodes. All specifications did count the number of function calls and stopped when it reached 1 000. V1 refers to the old approach, V1.5 to the adapted old approach and V2 to the new approach.

cores at 2.4GHz speed and 8GB of RAM. Figure 6.4 visualizes the performance, the complete data can be found in Tables 8.10 to 8.12.

While there is no real world data of the size of typical TeSSLa specifications, 128 nodes seems to be a rational upper bound. As the data shows the runtime scales nicely with big specification, the data even suggests a somewhat logarithmic increase of the execution time in regard to node count.

## 6.2 Instrumentation Benchmarks

After the evaluation of the runtime itself it remains to evaluate the C instrumentation program. Note that the instrumentation is mostly a proof of concept and its main purpose for now was to generate test data for the runtime. But it seems feasible that it can be optimized and extended to become a general purpose trace collection tool, therefore some benchmarks were performed.

For reliable trace collection of software the performance impact of the instrumentation is important. To measure this a trivial C program was exercised with and without instrumentation. The program increments each integer from 0 to 100 000 000 by one and, if the incremented number is divisible by a given parameter $c$, adds it

```c
#include <stdio.h>

int intermediate = 0;

int inc(int input) {
  return input + 1;
}

void add(int input) {
  // Instrumentation Point
  intermediate += input;
}

int main() {
  for(int i = 0; i < 100000000; i++) {
    int result = inc(i);
    if(result % 100 == 0) {
      add(result);
    }
  }
  printf("%i", intermediate);
  return 0;
}
```

**Listing 6.2:** Example C program for benchmark purposes

to an intermediate result. Note that the program can and will perform some integer overflows. The code is shown in Listing 6.2.

The code is then instrumented to log each call of the `add` method. For each benchmark the compiled program was run 50 times. All benchmarks in this section were performed on an Intel Core i5 with four cores at a clock speed of 2.4GHz and 8GB of Ram.

### 6.2.1 Performance Comparison with non Instrumented Code and Compiler Optimizations

One interesting metric is the intrusiveness of the instrumentation, describing how an instrumented program performs in contrast to the same program without the added instrumentation. For this the parameter $c$ of the example program is set to 100, so that around 1% of all function calls are instrumented.

One thing that can be recognized is that the impact of instrumentation is not predictable when using compiler optimizations. Aggressive compiler optimizations can

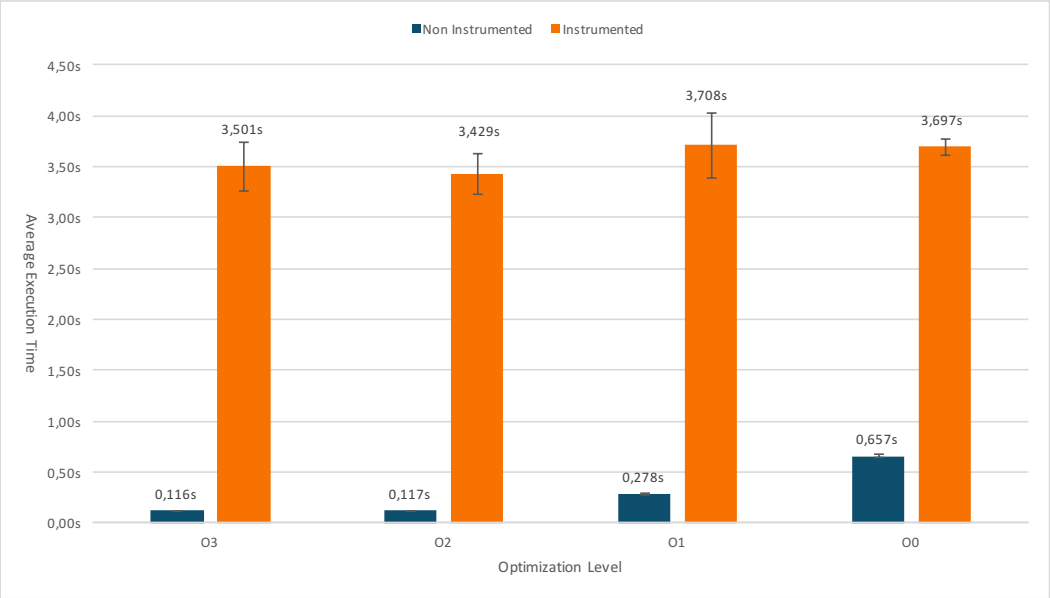


**Fig. 6.5:** Performance Comparision of an example C program with and without instrumentation.

remove function calls or inline values if the optimization doesn't change the program behaviour. When instrumentation calls are added no such optimization can happen, since otherwise the behaviour of the program would be altered, namely the logging would be removed when the function calls are removed.

Therefore we compare the instrumented and non-instrumented code for all optimization levels. The comparison can be seen in Figure 6.5. The complete data can be found in Tables 8.13 and 8.14.

The choice to instrument around 1% of the function calls was made arbitrary after some experimentation showed that a higher amount leads to huge performance impacts as shown in Section 6.2.2. Intuitively it sounds reasonable that in a real life example only a small subset of the function calls are interesting for trace generation, since TeSSLa specifications should be used to monitor only critical parts of a system.

### 6.2.2 Performance Impact in Regard to Instrumentation Percentage

To further investigate the impact of instrumentation we will look at the performance impact with respect to the percentage of function calls that are instrumented. Therefore the variable $c$ of the program is changed, which leads to different amount of calls to the instrumented function. For all values the program was compiled with the maximal optimization setting. In Figure 6.6 the results of changing the parameter can be seen. The complete dataset can be found in Table 8.15. TODO: Maybe change y-axis to times of original performance.

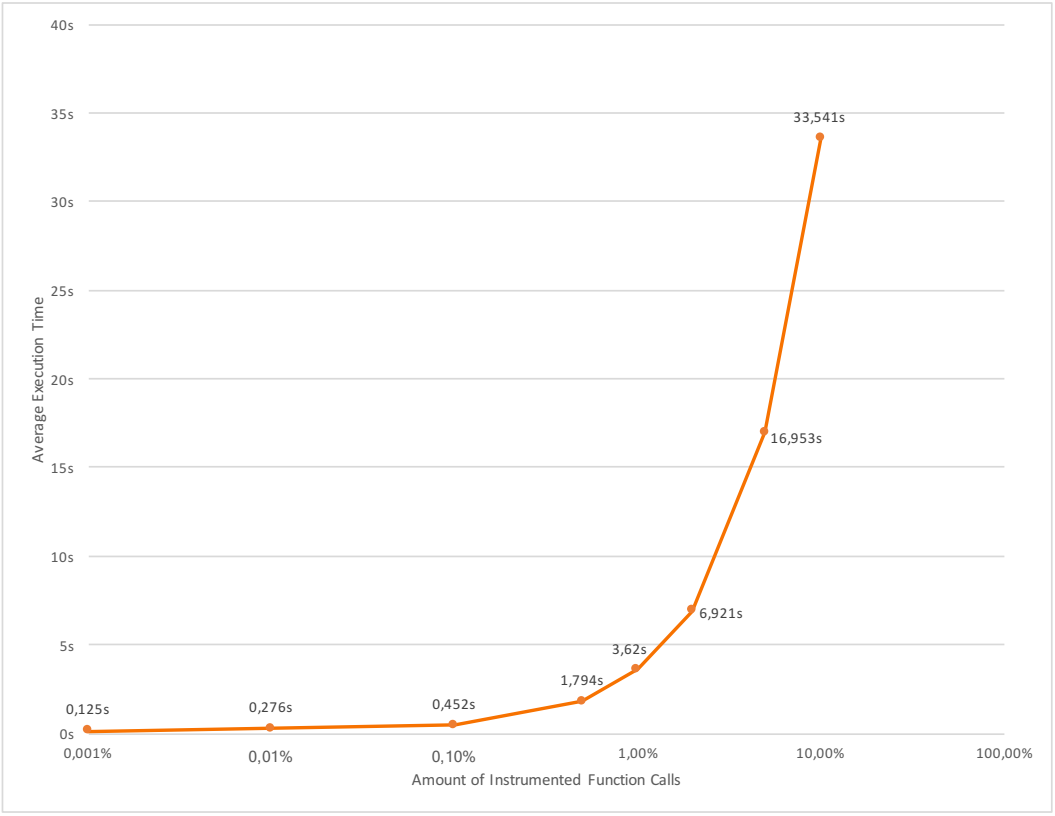


**Fig. 6.6:** Execution time of a program with different amount of calls to an instrumented function

Note that the $x$-axis is logarithmic. As expected the performance impact scales directly with the amount of calls to the instrumented function. At some point no more speed up will happen, since no calls to the instrumented functions will be made.

## 6.3 Practical Examples

Additionally to the theoretic benchmarks it was also important to evaluate the runtime against some more practical examples and test a bigger amount of node types. While the specifications for the benchmarks from the previous section did use some node types and showed that these types perform the correct computations (else the benchmarks wouldn't have worked), they only used a subset of all available types. Thus the following examples concentrate on a wider range of different nodes.

To do so traces from real world examples were collected, either with the developed instrumentation pass or by adjusting existing logs to fit the needs of TeSSLa. The programs or traces were then modified to deliberately include errors to test if appropriate TeSSLa specifications will find them when evaluated in the runtime.

As the main source for test trace data a ringbuffer, implemented in C, was modified by the instrumentation to log specific events. The code of the ringbuffer can be found in Listing 8.1. Events that are logged are of two types: calls to the `process` function and changes of the `write_ptr` variable. The generated traces can be used for a variety of specifications, e.g.: performance tests, race conditions or wrong initilizations.

The Following specifications were tested over the traces:

- Data is only processed when it is available, meaning that at each point of the trace the amount of changes of `write_ptr` is bigger than the number of calls to `process`.
- The given buffer size of five is never exceeded, meaning the difference of the amount of changes of both event types is never bigger than five.
- Everytime new data is added, there is a call to process within the next milisecond.

The specifications were intentionally chosen and written in such a way, that they use a big amount of different nodes. The actual TeSSLa specifications used to describe the requirements can be found in Section 8.4. For each of these specification the traces were deliberately altered to violate them. Evaluation of the specifications over the traces with TesslaServer always led to the correct conclusions.

# 7 Conclusion

## 7.1 Conclusion for TesslaServer

note limitations of Type system of compiler and Runtime

Optimizations: Bundle up events, throw away unneeded progress events in bundles For some notes only subset of inputs is important for output progress (think if then else) -> Some things na be optimized by compiler (literal as cond to ite) others only during evaluation.

Implement 'end' notification to flush all values. Total ordering of inputs necessary, relax to per channel. Argue that channels could be split by thread id.

## 7.2 Conclusion for the Instrumentation Pass

TODO: Timing model, maybe somehow harness Vector clocks -> wwould complicate instrumentation! Also would need support in TeSSLa language. Extract logging to different thread -> run on own core, less overhead. Configuration file for instrumetnation -> e.g. include conditions

In the benchmarks of the instrumentation it is obvious that it adds a lot of overhead, especially when compiler optimizations are turned on or the percentage of instrumented function calls is large. While the overhead is big, it is stable (see the standard deviation in Section 6.2.1).

For now the instrumentation should be seen as a proof of concept and test tool. The generated traces should be mostly used for analysing test settings, where compiler optimizations are turned off. When used for timing specifications, the instrumented code can be benchmarked against a non instrumented version of the code and the results can be used to transform the actual timing requirements to corresponding ones for the instrumented code. This will at least give an approximation of the actual results. Also it is recommended to write small and specific TeSSLa specifications, so that the instrumentations only have to be applied to a small subset of functions. Obviously it is strongly discouraged to use the instrumented code in any production setting.

## 7.3 Further Work

Composition of Transducers/evalEngines Port to Scala/akka Different evaluation model: Pull not push port to genstage: Concept of backpressure -> Nodes generating events out of nothing

online monitoring -> possible infinite traces

Architecture is similar to a vm: Tessla specs are code, compiler produces intermediate representation (json), runtime executes Ir. Therefore: Maybe define new functions (read nodes) in the spec itself and not in the runtime? Also recursion in specifications

### 7.3.1 Error prevention

Ways of sending observations back to the program to recover from or prevent errors

# 8 Appendix

## 8.1 Runtime Benchmark Data

This section includes all data from the benchmarks of TesslaServer.

### 8.1.1 Execution Time in Regard to Used Processor Cores

**Tab. 8.1:** Execution time of multiple runs of TesslaServer V1 with 10 000 input events with different number of used processor cores.

| Run # | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| 1 | 38.2684 s | 95.8390 s | 33.2289 s | 18.3235 s | 21.9672 s |
| 2 | 38.9833 s | 98.9827 s | 32.3660 s | 17.9681 s | 19.4063 s |
| 3 | 38.1630 s | 77.4360 s | 36.0462 s | 17.0076 s | 27.6495 s |
| 5 | 38.4424 s | 222.9723 s | 29.4773 s | 17.2925 s | 17.3606 s |
| 6 | 39.2239 s | 75.5560 s | 32.1869 s | 15.3906 s | 19.1662 s |
| 7 | 38.7815 s | 81.0928 s | 44.7217 s | 18.3882 s | 21.9974 s |
| 8 | 38.7514 s | 119.1012 s | 38.0911 s | 18.0104 s | 20.2454 s |
| 9 | 37.8491 s | 60.2407 s | 35.0671 s | 17.9517 s | 17.2825 s |
| 10 | 38.2721 s | 83.5956 s | 37.5839 s | 17.1108 s | 15.5887 s |
| 11 | 38.5657 s | 105.2514 s | 43.8580 s | 15.8287 s | 22.0196 s |
| 12 | 38.1699 s | 75.6985 s | 32.4470 s | 15.7641 s | 18.3230 s |
| 13 | 37.9775 s | 104.0994 s | 37.0281 s | 24.5554 s | 17.0857 s |
| 14 | 38.8215 s | 75.0534 s | 34.8001 s | 14.3733 s | 19.5598 s |
| 15 | 38.3913 s | 74.5584 s | 32.7344 s | 17.9163 s | 17.3179 s |
| 16 | 40.4536 s | 111.3214 s | 35.7434 s | 16.6162 s | 20.2855 s |
| 17 | 39.0735 s | 87.0386 s | 35.6367 s | 15.5962 s | 21.7681 s |
| 18 | 38.8830 s | 114.2697 s | 33.4537 s | 28.1556 s | 17.2382 s |
| 19 | 39.0799 s | 103.1956 s | 32.7134 s | 17.7034 s | 23.5276 s |
| 20 | 39.3866 s | 117.2423 s | 39.1457 s | 15.6895 s | 18.4146 s |

**Tab. 8.2:** Execution time of multiple runs of TesslaServer V1.5 with 10 000 input events with different number of used processor cores

| Run # | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| 1 | 6.3722 s | 0.1256 s | 0.1309 s | 0.1378 s | 2.6756 s |
| 2 | 6.2417 s | 0.1300 s | 0.1333 s | 0.1444 s | 2.6292 s |
| 3 | 6.2953 s | 0.1301 s | 0.1325 s | 0.1379 s | 2.6453 s |
| 5 | 6.5148 s | 0.1433 s | 0.1315 s | 0.1451 s | 2.5836 s |
| 6 | 6.6828 s | 0.1347 s | 0.1480 s | 0.1468 s | 2.6670 s |
| 7 | 6.8090 s | 0.1386 s | 0.1294 s | 0.1414 s | 2.6671 s |
| 8 | 6.3252 s | 0.1307 s | 0.1330 s | 0.1385 s | 2.7755 s |
| 9 | 6.0975 s | 0.1304 s | 0.1339 s | 0.1362 s | 2.6097 s |
| 10 | 6.1274 s | 0.1328 s | 0.1325 s | 0.1371 s | 2.6335 s |
| 11 | 5.7481 s | 0.1479 s | 0.1299 s | 0.1398 s | 2.6477 s |
| 12 | 6.0317 s | 0.1318 s | 0.1435 s | 0.1311 s | 2.6343 s |
| 13 | 6.1565 s | 0.1293 s | 0.1260 s | 0.1361 s | 2.6603 s |
| 14 | 6.3439 s | 0.1223 s | 0.1309 s | 0.1313 s | 2.5560 s |
| 15 | 6.3749 s | 0.1318 s | 0.1335 s | 0.1338 s | 2.6739 s |
| 16 | 6.3289 s | 0.1321 s | 0.1323 s | 0.1503 s | 2.5466 s |
| 17 | 6.3894 s | 0.1253 s | 0.1323 s | 0.1397 s | 2.6240 s |
| 18 | 6.3943 s | 0.1288 s | 0.1320 s | 0.1316 s | 2.6939 s |
| 19 | 6.4089 s | 0.1319 s | 0.1347 s | 0.1302 s | 2.5940 s |
| 20 | 6.3865 s | 0.1698 s | 0.1336 s | 0.1312 s | 2.6347 s |

**Tab. 8.3:** Execution time of multiple runs of TesslaServer V2 with 10 000 input events with different number of used processor cores

| Run # | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|

### 8.1.2 Execution Time in Regard to Number of Input Events

**Tab. 8.4:** Execution time of multiple runs of TesslaServer V1 with different number of input events

| Run # | 500 | 1000 | 5000 | 10000 |
|---|---|---|---|---|
| 1 | 0.5815 s | 1.1913 s | 39.0046 s | timeout |
| 2 | 0.5958 s | 1.1991 s | 33.5771 s | timeout |
| 3 | 0.5963 s | 1.2122 s | 26.8252 s | timeout |
| 5 | 0.580 s | 1.2569 s | 23.8944 s | timeout |
| 6 | 0.5948 s | 1.2280 s | 23.8837 s | timeout |
| 7 | 0.5825 s | 1.2519 s | 23.4110 s | timeout |
| 8 | 0.6271 s | 1.2636 s | 23.1929 s | timeout |
| 9 | 0.5705 s | 1.2324 s | 23.3227 s | timeout |
| 10 | 0.5765 s | 1.1783 s | 21.9043 s | timeout |
| 11 | 0.5893 s | 1.1928 s | 22.9516 s | timeout |
| 12 | 0.6101 s | 1.2759 s | 22.8377 s | timeout |
| 13 | 0.5684 s | 1.1955 s | 26.1974 s | timeout |
| 14 | 0.5746 s | 1.2047 s | 25.7536 s | timeout |
| 15 | 0.5588 s | 1.2044 s | 24.6289 s | timeout |
| 16 | 0.5698 s | 1.1958 s | 22.3709 s | timeout |
| 17 | 0.5911 s | 1.2271 s | 23.9404 s | timeout |
| 18 | 0.5955 s | 1.1981 s | 21.4453 s | timeout |
| 19 | 0.5702 s | 1.1718 s | 23.3482 s | timeout |
| 20 | 0.5863 s | 1.2571 s | 21.9753 s | timeout |

**Tab. 8.5:** Execution time of multiple runs of TesslaServer V1.5 with different number of input events

| Run # | 500 | 1000 | 5000 | 10000 |
|---|---|---|---|---|
| 1 | 0.4512 s | 0.5901 s | 2.1607 s | 2.8476 s |
| 2 | 0.4349 s | 0.5703 s | 2.0242 s | 2.8715 s |
| 3 | 0.4424 s | 0.5511 s | 2.1535 s | 2.8250 s |
| 5 | 0.4373 s | 0.5487 s | 2.1715 s | 2.7025 s |
| 6 | 0.4239 s | 0.5564 s | 2.2763 s | 2.7671 s |
| 7 | 0.4670 s | 0.5466 s | 4.2085 s | 2.6719 s |
| 8 | 0.4492 s | 0.5401 s | 2.6771 s | 2.6956 s |
| 9 | 0.4649 s | 0.6345 s | 1.9132 s | 2.5634 s |
| 10 | 0.440 s | 0.5618 s | 1.5309 s | 2.6090 s |
| 11 | 0.4641 s | 0.5577 s | 2.5762 s | 2.6738 s |
| 12 | 0.4306 s | 0.5415 s | 1.6873 s | 2.7178 s |
| 13 | 0.4255 s | 0.5829 s | 2.0342 s | 2.7294 s |
| 14 | 0.460 s | 0.5442 s | 3.0314 s | 2.6789 s |
| 15 | 0.4401 s | 0.5443 s | 3.2066 s | 2.6045 s |
| 16 | 0.4516 s | 0.5848 s | 2.5746 s | 2.5837 s |
| 17 | 0.4927 s | 0.5406 s | 2.4933 s | 2.7866 s |
| 18 | 0.4635 s | 0.5669 s | 2.6895 s | 2.6719 s |
| 19 | 0.4506 s | 0.5576 s | 2.2802 s | 2.7056 s |
| 20 | 0.5080 s | 0.5898 s | 2.4456 s | 2.6697 s |

Tab. 8.6: Execution time of multiple runs of TesslaServer V2 with different number of input events

| Run # | 500 | 1000 | 5000 | 10000 |
|---|---|---|---|---|

### 8.1.3 Ram Usage with Respect to Number of Input Events

**Tab. 8.7:** RAM usage of multiple runs of TesslaServer V1 with different number of input events

| Run # | 500 | 1000 | 5000 | |
|---|---|---|---|---|
| 1 | 98.50 MB | 182.93 MB | 1.55 GB | timeout |
| 2 | 95.47 MB | 226.23 MB | 1.69 GB | timeout |
| 3 | 84.59 MB | 223.75 MB | 1.62 GB | timeout |
| 5 | 102.98 MB | 185.86 MB | 1.78 GB | timeout |
| 6 | 90.24 MB | 222.20 MB | 1.77 GB | timeout |
| 7 | 84.54 MB | 181.79 MB | 1.75 GB | timeout |
| 8 | 89.31 MB | 171.32 MB | 1.60 GB | timeout |
| 9 | 89.84 MB | 228.12 MB | 1.75 GB | timeout |
| 10 | 106.07 MB | 184.48 MB | 1.67 GB | timeout |
| 11 | 89.41 MB | 194.26 MB | 1.47 GB | timeout |
| 12 | 97.62 MB | 163.42 MB | 1.80 GB | timeout |
| 13 | 97.54 MB | 160.26 MB | 1.51 GB | timeout |
| 14 | 88.77 MB | 221.16 MB | 1.96 GB | timeout |
| 15 | 99.99 MB | 223.81 MB | 1.87 GB | timeout |
| 16 | 97.40 MB | 216.00 MB | 1.87 GB | timeout |
| 17 | 87.62 MB | 250.93 MB | 1.68 GB | timeout |
| 18 | 120.70 MB | 206.97 MB | 1.71 GB | timeout |
| 19 | 91.89 MB | 246.55 MB | 1.64 GB | timeout |
| 20 | 88.91 MB | 252.26 MB | 1.87 GB | timeout |

**Tab. 8.8:** RAM usage of multiple runs of TesslaServer V1.5 with different number of input events

| Run # | 500 | 1000 | 5000 | 1000 |
|---|---|---|---|---|
| 1 | 52.09 MB | 53.71 MB | 61.49 MB | 58.37 MB |
| 2 | 48.77 MB | 56.98 MB | 59.74 MB | 60.16 MB |
| 3 | 52.06 MB | 52.60 MB | 57.21 MB | 75.88 MB |
| 5 | 52.42 MB | 51.23 MB | 57.45 MB | 61.79 MB |
| 6 | 54.13 MB | 52.44 MB | 56.55 MB | 74.53 MB |
| 7 | 50.30 MB | 50.41 MB | 76.10 MB | 62.92 MB |
| 8 | 47.91 MB | 52.48 MB | 53.78 MB | 63.47 MB |
| 9 | 50.97 MB | 51.99 MB | 58.85 MB | 60.24 MB |
| 10 | 52.26 MB | 51.00 MB | 57.04 MB | 59.36 MB |
| 11 | 51.25 MB | 54.08 MB | 57.74 MB | 62.32 MB |
| 12 | 47.60 MB | 52.06 MB | 53.92 MB | 65.00 MB |
| 13 | 50.15 MB | 52.54 MB | 57.23 MB | 64.99 MB |
| 14 | 51.03 MB | 52.09 MB | 59.37 MB | 64.33 MB |
| 15 | 52.56 MB | 62.94 MB | 53.98 MB | 65.32 MB |
| 16 | 50.79 MB | 50.54 MB | 58.80 MB | 67.25 MB |
| 17 | 61.64 MB | 49.85 MB | 56.02 MB | 58.18 MB |
| 18 | 50.69 MB | 51.54 MB | 60.00 MB | 59.93 MB |
| 19 | 49.59 MB | 52.21 MB | 61.75 MB | 82.53 MB |
| 20 | 50.04 MB | 51.91 MB | 67.09 MB | 72.43 MB |

Tab. 8.9: RAM usage of multiple runs of TesslaServer V2 with different number of input events

| Run # | 500 | 1000 | 5000 | 1000 |
|---|---|---|---|---|

### 8.1.4 Execution Time in Regard to Number of Nodes

**Tab. 8.10:** Execution time of multiple runs of TesslaServer V1 in regard to different amount of nodes in a specification

| Run # | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|
| 1 | 0.997 s | 1.295 s | 2.893 s | 3.780 s | 7.166 s |
| 2 | 0.875 s | 1.280 s | 2.903 s | 4.139 s | 7.071 s |
| 3 | 1.269 s | 1.277 s | 2.837 s | 4.055 s | 7.083 s |
| 5 | 0.888 s | 1.262 s | 3.693 s | 3.811 s | 6.937 s |
| 6 | 0.917 s | 1.330 s | 4.930 s | 4.150 s | 7.075 s |
| 7 | 0.881 s | 1.419 s | 3.761 s | 3.826 s | 7.327 s |
| 8 | 0.959 s | 1.398 s | 3.115 s | 3.835 s | 7.158 s |
| 9 | 0.973 s | 1.294 s | 2.944 s | 4.093 s | 7.102 s |
| 10 | 0.980 s | 1.257 s | 3.143 s | 4.208 s | 6.981 s |
| 11 | 0.918 s | 1.285 s | 2.788 s | 4.092 s | 7.081 s |
| 12 | 0.939 s | 1.329 s | 2.914 s | 4.033 s | 7.198 s |
| 13 | 0.965 s | 1.225 s | 2.850 s | 3.815 s | 7.135 s |
| 14 | 0.871 s | 1.320 s | 3.003 s | 3.817 s | 6.943 s |
| 15 | 0.898 s | 1.336 s | 2.922 s | 3.905 s | 6.901 s |
| 16 | 0.897 s | 1.357 s | 3.212 s | 3.968 s | 7.153 s |
| 17 | 1.073 s | 1.401 s | 3.057 s | 4.081 s | 7.10 s |
| 18 | 1.182 s | 1.342 s | 2.759 s | 3.826 s | 7.311 s |
| 19 | 1.019 s | 1.288 s | 2.945 s | 3.991 s | 7.064 s |
| 20 | 0.928 s | 1.314 s | 3.304 s | 5.343 s | 7.161 s |

**Tab. 8.11:** Execution time of multiple runs of TesslaServer V1.5 in regard to different amount of nodes in a specification

| Run # | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|
| 1 | 0.474 s | 0.602 s | 0.811 s | 1.353 s | 2.134 s |
| 2 | 0.476 s | 0.605 s | 0.813 s | 1.410 s | 2.431 s |
| 3 | 0.487 s | 0.577 s | 0.884 s | 1.331 s | 2.015 s |
| 5 | 0.502 s | 0.606 s | 0.933 s | 1.273 s | 2.138 s |
| 6 | 0.511 s | 0.615 s | 0.930 s | 1.231 s | 2.287 s |
| 7 | 0.518 s | 0.577 s | 0.813 s | 1.269 s | 2.638 s |
| 8 | 0.527 s | 0.603 s | 0.898 s | 1.306 s | 2.90 s |
| 9 | 0.540 s | 0.596 s | 0.946 s | 1.195 s | 2.874 s |
| 10 | 0.511 s | 0.618 s | 0.834 s | 1.319 s | 3.80 s |
| 11 | 0.494 s | 0.624 s | 0.796 s | 1.260 s | 2.842 s |
| 12 | 0.561 s | 0.601 s | 0.893 s | 1.497 s | 3.275 s |
| 13 | 0.487 s | 0.619 s | 0.843 s | 1.256 s | 2.898 s |
| 14 | 0.541 s | 0.582 s | 0.868 s | 1.203 s | 2.951 s |
| 15 | 0.482 s | 0.627 s | 0.80 s | 1.257 s | 3.118 s |
| 16 | 0.487 s | 0.604 s | 0.861 s | 1.347 s | 4.078 s |
| 17 | 0.476 s | 0.591 s | 0.862 s | 1.213 s | 7.687 s |
| 18 | 0.574 s | 0.596 s | 0.846 s | 1.284 s | 5.113 s |
| 19 | 0.555 s | 0.590 s | 0.824 s | 1.277 s | 3.892 s |
| 20 | 0.507 s | 0.619 s | 0.879 s | 1.275 s | 3.991 s |

Tab. 8.12: Execution time of multiple runs of TesslaServer V2 in regard to different amount of nodes in a specification

| Run # | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|

## 8.2 Ringbuffer Code

```
#include <stdio.h>
#include <stdatomic.h>
#include <pthread.h>
#include <stdlib.h>
#include <sys/time.h>

int BUF_SIZE = 5;

char* ring_buffer;
char* _Atomic read_ptr;
char* write_ptr;

void init_ring_buffer() {
  ring_buffer = malloc(BUF_SIZE);
  read_ptr = ring_buffer;
  write_ptr = ring_buffer;

}

char* buffer_next(char* ptr) {
  return (char*) ring_buffer + (((ptr - ring_buffer) + 1)
      % BUF_SIZE);
}

void process(char* data){
  struct timespec ts;
  ts.tv_sec = 0;
  ts.tv_nsec = 1000000;
  nanosleep(&ts, NULL);
}

void producer_main() {
  char* next_write_ptr;
  while (!feof(stdin)){
    next_write_ptr = buffer_next(write_ptr);
    if (next_write_ptr != read_ptr) {
      *write_ptr = getc(stdin);
      write_ptr = next_write_ptr;
```

```
39        struct timespec ts;
          ts.tv_sec = 0;
          ts.tv_nsec = 100000000ULL * rand() / RAND_MAX ;
42        nanosleep(&ts, NULL);
        }
      }
45 }

   void* consumer_main(void* thread_id) {
48   int tid = (int) thread_id;
     char* current_rptr;
     char* current_wptr;
51   char data = 0;
     char* next_read_ptr;
     char* local_read_ptr;
54
     while (1) {
       local_read_ptr = read_ptr;
57
       if (local_read_ptr != write_ptr) {
         data = *local_read_ptr;
60       next_read_ptr = buffer_next(local_read_ptr);
         if (atomic_compare_exchange_weak(&read_ptr, &
            local_read_ptr, next_read_ptr)) {
           process(&data);
63       }
         struct timespec ts;
         ts.tv_sec = 0;
66       ts.tv_nsec = 100000000ULL * rand() / RAND_MAX ;
         nanosleep(&ts, NULL);
       }
69   }
   }

72 void create_consumers(int num_consumers) {
     pthread_t threads[num_consumers];
     int thread_ids[num_consumers];
75   int rc;
     for(int t = 0; t < num_consumers; t++){
       thread_ids[t] = t;
78     rc = pthread_create(&threads[t], NULL, consumer_main,
           (void* ) t);
```

```
      if (rc){
        pthread_exit(NULL);
81    }
    }
  }
84
  int main() {
    init_ring_buffer();
87
    create_consumers(2);

90  producer_main();

    return 0;
93 }
```

**Listing 8.1:** Code of the ringbuffer example programm

## 8.3 Instrumentation Benchmark Data

**Tab. 8.13:** Execution time of multiple runs of an uninstrumented example C program compiled with different optimization levels

| Run # | O3 | O2 | O1 | O0 |
|---|---|---|---|---|
| 1 | 0.1161 s | 0.1191 s | 0.2948 s | 0.6530 s |
| 2 | 0.1157 s | 0.1156 s | 0.2738 s | 0.6632 s |
| 3 | 0.1167 s | 0.1148 s | 0.2762 s | 0.6660 s |
| 5 | 0.1141 s | 0.1238 s | 0.2678 s | 0.6449 s |
| 6 | 0.1164 s | 0.1141 s | 0.2763 s | 0.6779 s |
| 7 | 0.1149 s | 0.1187 s | 0.2759 s | 0.6569 s |
| 8 | 0.1200 s | 0.1117 s | 0.2793 s | 0.6452 s |
| 9 | 0.1100 s | 0.1159 s | 0.2894 s | 0.6577 s |
| 10 | 0.1103 s | 0.1151 s | 0.2816 s | 0.6565 s |
| 11 | 0.1138 s | 0.1219 s | 0.2807 s | 0.6649 s |
| 12 | 0.1163 s | 0.1176 s | 0.2723 s | 0.6588 s |
| 13 | 0.1170 s | 0.1168 s | 0.2781 s | 0.6618 s |
| 14 | 0.1127 s | 0.1200 s | 0.2799 s | 0.6571 s |
| 15 | 0.1124 s | 0.1124 s | 0.2763 s | 0.6544 s |
| 16 | 0.1231 s | 0.1198 s | 0.2752 s | 0.6740 s |
| 17 | 0.1203 s | 0.1191 s | 0.2750 s | 0.6417 s |
| 18 | 0.1163 s | 0.1229 s | 0.2662 s | 0.6602 s |
| 19 | 0.1186 s | 0.1143 s | 0.2685 s | 0.6608 s |
| 20 | 0.1182 s | 0.1160 s | 0.2935 s | 0.6540 s |
| 21 | 0.1166 s | 0.1128 s | 0.2743 s | 0.6557 s |
| 22 | 0.1196 s | 0.1137 s | 0.2758 s | 0.6767 s |
| 23 | 0.1156 s | 0.1164 s | 0.2886 s | 0.6637 s |
| 25 | 0.1203 s | 0.1166 s | 0.2829 s | 0.6816 s |
| 26 | 0.1173 s | 0.1203 s | 0.2849 s | 0.6397 s |
| 27 | 0.1160 s | 0.1158 s | 0.2814 s | 0.6493 s |
| 28 | 0.1161 s | 0.1233 s | 0.2841 s | 0.6558 s |
| 29 | 0.1086 s | 0.1245 s | 0.2797 s | 0.6555 s |
| 30 | 0.1138 s | 0.1158 s | 0.2746 s | 0.6605 s |
| 31 | 0.1159 s | 0.1194 s | 0.2767 s | 0.6677 s |
| 32 | 0.1231 s | 0.1197 s | 0.2818 s | 0.6489 s |
| 33 | 0.1154 s | 0.1125 s | 0.2842 s | 0.6527 s |
| 34 | 0.1147 s | 0.1216 s | 0.2762 s | 0.6488 s |
| 35 | 0.1205 s | 0.1136 s | 0.2732 s | 0.6657 s |
| 36 | 0.1184 s | 0.1135 s | 0.2743 s | 0.6611 s |
| 37 | 0.1149 s | 0.1160 s | 0.2773 s | 0.6637 s |
| 38 | 0.1183 s | 0.1157 s | 0.2722 s | 0.6638 s |
| 39 | 0.1138 s | 0.1105 s | 0.2787 s | 0.6738 s |
| 40 | 0.1113 s | 0.1144 s | 0.2707 s | 0.6705 s |

Tab. 8.14: Execution time of multiple runs of an instrumented example C program compiled with different optimization levels

| Run # | O3 | O2 | O1 | O0 |
|---|---|---|---|---|
| 1 | 3.3911 s | 3.2436 s | 3.4216 s | 3.8694 s |
| 2 | 3.5288 s | 3.6410 s | 3.3192 s | 3.6850 s |
| 3 | 3.3494 s | 3.4974 s | 3.3641 s | 3.6085 s |
| 5 | 3.5640 s | 3.4832 s | 3.5960 s | 3.7213 s |
| 6 | 3.3368 s | 3.4711 s | 3.4086 s | 3.6712 s |
| 7 | 3.5097 s | 3.5463 s | 3.4823 s | 3.6867 s |
| 8 | 3.3817 s | 3.3919 s | 3.4265 s | 3.8107 s |
| 9 | 3.6170 s | 3.4889 s | 3.5831 s | 3.6996 s |
| 10 | 3.3764 s | 3.3402 s | 3.4814 s | 3.7145 s |
| 11 | 3.4867 s | 3.7752 s | 3.5054 s | 3.7900 s |
| 12 | 3.3393 s | 3.3320 s | 3.6760 s | 3.7082 s |
| 13 | 3.2769 s | 3.3553 s | 3.7271 s | 3.6436 s |
| 14 | 3.5930 s | 3.3282 s | 3.4532 s | 3.6124 s |
| 15 | 3.5464 s | 3.5488 s | 3.5590 s | 3.7060 s |
| 16 | 3.6068 s | 3.5292 s | 3.4408 s | 3.6440 s |
| 17 | 3.3833 s | 3.3312 s | 3.6440 s | 3.6952 s |
| 18 | 4.6447 s | 3.2691 s | 3.6565 s | 3.7749 s |
| 19 | 3.9256 s | 3.3252 s | 3.5309 s | 3.6850 s |
| 20 | 3.3639 s | 3.3914 s | 3.6782 s | 3.6187 s |
| 21 | 3.6016 s | 3.3473 s | 4.5602 s | 3.6672 s |
| 22 | 3.2825 s | 3.3359 s | 4.2268 s | 3.5982 s |
| 23 | 3.1969 s | 3.3892 s | 4.3686 s | 3.6702 s |
| 25 | 3.3095 s | 3.3870 s | 3.5713 s | 3.7751 s |
| 26 | 3.4130 s | 3.4193 s | 3.6797 s | 3.6604 s |
| 27 | 3.5319 s | 3.3532 s | 3.5412 s | 3.6823 s |
| 28 | 3.3935 s | 3.3253 s | 3.5808 s | 3.6570 s |
| 29 | 3.3294 s | 3.3051 s | 3.4674 s | 3.6164 s |
| 30 | 3.4536 s | 3.3509 s | 3.5215 s | 3.6644 s |
| 31 | 3.4952 s | 3.4388 s | 3.8695 s | 3.7006 s |
| 32 | 3.4600 s | 3.3684 s | 4.1289 s | 3.6317 s |
| 33 | 3.5761 s | 3.4953 s | 4.5513 s | 3.6137 s |
| 34 | 3.5816 s | 3.4220 s | 4.4331 s | 3.7160 s |
| 35 | 3.6319 s | 4.3748 s | 3.6797 s | 3.6431 s |
| 36 | 3.4037 s | 3.7254 s | 3.5498 s | 3.6337 s |
| 37 | 3.5317 s | 3.3875 s | 3.9773 s | 4.0382 s |
| 38 | 3.6149 s | 3.2542 s | 3.6003 s | 3.6492 s |
| 39 | 3.3683 s | 3.1844 s | 3.6274 s | 3.6503 s |
| 40 | 3.3448 s | 3.3463 s | 3.6728 s | 3.6539 s |

Tab. 8.15: Execution time of multiple runs of an instrumented program with different percentages of instrumented function calls

| Run # | 10% | 5% | 2% | 1% | 0.5% | 0.1% | 0.01% | 0.001% |
|---|---|---|---|---|---|---|---|---|
| 1 | 32.2744 s | 16.5746 s | 6.8182 s | 3.4653 s | 1.9216 s | 0.5291 s | 0.3014 s | 0.1262 s |
| 2 | 32.0180 s | 17.2402 s | 6.7273 s | 4.7988 s | 1.8523 s | 0.5143 s | 0.2784 s | 0.1243 s |
| 3 | 32.5687 s | 16.5926 s | 6.9581 s | 4.0173 s | 1.7571 s | 0.4944 s | 0.2864 s | 0.1245 s |
| 5 | 31.7277 s | 16.4390 s | 6.5665 s | 3.9327 s | 1.7359 s | 0.4897 s | 0.2750 s | 0.1209 s |
| 6 | 33.2172 s | 16.2865 s | 6.6697 s | 4.3147 s | 1.7712 s | 0.4748 s | 0.2754 s | 0.1266 s |
| 7 | 32.0802 s | 16.4187 s | 6.6856 s | 3.9003 s | 1.7668 s | 0.4553 s | 0.2718 s | 0.1314 s |
| 8 | 36.0954 s | 16.7155 s | 6.6875 s | 3.8426 s | 1.8202 s | 0.4432 s | 0.2712 s | 0.1301 s |
| 9 | 36.9455 s | 16.7840 s | 6.5936 s | 4.2499 s | 1.7469 s | 0.4662 s | 0.2723 s | 0.1200 s |
| 10 | 33.0339 s | 20.1691 s | 7.4481 s | 3.4560 s | 1.8394 s | 0.4528 s | 0.3151 s | 0.1175 s |
| 11 | 35.2251 s | 19.8842 s | 7.2807 s | 3.8017 s | 1.7728 s | 0.4460 s | 0.3352 s | 0.1189 s |
| 12 | 31.8794 s | 20.2660 s | 7.0673 s | 5.2281 s | 1.6975 s | 0.4340 s | 0.2739 s | 0.1209 s |
| 13 | 32.9260 s | 19.2307 s | 6.7634 s | 4.3974 s | 1.7026 s | 0.4441 s | 0.2708 s | 0.1195 s |
| 14 | 31.9317 s | 17.8926 s | 6.7118 s | 3.7419 s | 1.7544 s | 0.4361 s | 0.2752 s | 0.1216 s |
| 15 | 32.6378 s | 17.5177 s | 7.7485 s | 3.3878 s | 1.8489 s | 0.5027 s | 0.2767 s | 0.1240 s |
| 16 | 32.1711 s | 16.3863 s | 6.7146 s | 3.3716 s | 2.0388 s | 0.4380 s | 0.2893 s | 0.1266 s |
| 17 | 32.4985 s | 16.4408 s | 6.6772 s | 4.1358 s | 1.8717 s | 0.4349 s | 0.2669 s | 0.1210 s |
| 18 | 32.1387 s | 16.4152 s | 7.1151 s | 3.2681 s | 1.8253 s | 0.4491 s | 0.2719 s | 0.1217 s |
| 19 | 32.0004 s | 16.1371 s | 6.7711 s | 3.6237 s | 1.7483 s | 0.4274 s | 0.2678 s | 0.1215 s |
| 20 | 32.5547 s | 16.4341 s | 7.2219 s | 3.7627 s | 1.7674 s | 0.4555 s | 0.2658 s | 0.1222 s |
| 21 | 32.1219 s | 16.1507 s | 6.7387 s | 3.4525 s | 1.7188 s | 0.4372 s | 0.2741 s | 0.1280 s |
| 22 | 32.7006 s | 16.2854 s | 6.5879 s | 3.4259 s | 1.7273 s | 0.4447 s | 0.2731 s | 0.1207 s |
| 23 | 32.0705 s | 15.9990 s | 6.6927 s | 3.6898 s | 1.7741 s | 0.4446 s | 0.2728 s | 0.1271 s |

Tab. 8.15: Execution time of multiple runs of an instrumented program with different percentages of instrumented function calls

| Run # | 10% | 5% | 2% | 1% | 0.5% | 0.1% | 0.01% | 0.001% |
|---|---|---|---|---|---|---|---|---|
| 25 | 32.5558 s | 16.2882 s | 6.7336 s | 3.2830 s | 1.8821 s | 0.4399 s | 0.2744 s | 0.1245 s |
| 26 | 35.2615 s | 17.6253 s | 6.5753 s | 3.3661 s | 1.8344 s | 0.4657 s | 0.2745 s | 0.1279 s |
| 27 | 46.2991 s | 16.6608 s | 6.6603 s | 3.3115 s | 1.7554 s | 0.4431 s | 0.2678 s | 0.1266 s |
| 28 | 34.3145 s | 16.7270 s | 6.7223 s | 3.5270 s | 1.7145 s | 0.4346 s | 0.2728 s | 0.1251 s |
| 29 | 33.3674 s | 15.9586 s | 6.7363 s | 3.3917 s | 1.7318 s | 0.4426 s | 0.2735 s | 0.1257 s |
| 30 | 35.3249 s | 15.8946 s | 7.1817 s | 3.2812 s | 1.7364 s | 0.4255 s | 0.2741 s | 0.1274 s |
| 31 | 37.5802 s | 17.0438 s | 8.0833 s | 3.9514 s | 1.7507 s | 0.4614 s | 0.2701 s | 0.1250 s |
| 32 | 35.6964 s | 17.9818 s | 6.6254 s | 3.4312 s | 1.8008 s | 0.4516 s | 0.2693 s | 0.1183 s |
| 33 | 32.3928 s | 16.2117 s | 6.8224 s | 3.4366 s | 1.7850 s | 0.4451 s | 0.2717 s | 0.1215 s |
| 34 | 34.6837 s | 16.1150 s | 8.0794 s | 3.4316 s | 1.7732 s | 0.4375 s | 0.2807 s | 0.1304 s |
| 35 | 32.9116 s | 16.6887 s | 7.7993 s | 3.6139 s | 1.7912 s | 0.4468 s | 0.2682 s | 0.1248 s |
| 36 | 32.2931 s | 17.4771 s | 6.6512 s | 3.4691 s | 1.9558 s | 0.4335 s | 0.2717 s | 0.1269 s |
| 37 | 32.5576 s | 16.2437 s | 6.6823 s | 3.3764 s | 1.8431 s | 0.4469 s | 0.2719 s | 0.1273 s |
| 38 | 32.4156 s | 16.1199 s | 6.6837 s | 3.3485 s | 1.8451 s | 0.4372 s | 0.2792 s | 0.1234 s |
| 39 | 32.7322 s | 16.1902 s | 6.6598 s | 3.5290 s | 1.8274 s | 0.4583 s | 0.2744 s | 0.1252 s |
| 40 | 32.7395 s | 16.4886 s | 6.7993 s | 3.3594 s | 1.8212 s | 0.4294 s | 0.2709 s | 0.1233 s |

## 8.4 Example TeSSLa Specifications

```
define values: Signal<Int> :=
  variable_values("buffer.c:write_ptr")
define writes: Events<Int> := changeOf(values)
define processed: Events<Unit> :=
  function_calls("buffer.c:process")

define bufLevel: Signal<Int> :=
  sub(eventCount(writes), eventCount(processed))

define error: Signal<Boolean> :=
  signalNot(and(
    leq(literal(0),bufLevel), leq(bufLevel,literal(5))
  )
)
```

**Listing 8.2:** TeSSLa specification that checks if the size of a buffer is always between zero and five. It counts the number of events happened on two input streams. The first stream denotes changes of the variable *write_ptr*, the second calls of the function *process*. An error is encountered when more data is processed than is present or if more than five items are written onto the buffer.

```
define values: Signal<Int> :=
  variable_values("buffer.c:write_ptr")
define new_data: Events<Int> := changeOf(values)
define data_processed: Events<Unit> :=
  function_calls("buffer.c:process")

define delayed_new_data: Events<Unit> :=
  delayEventByTime(new_data, 1000000)

define consumed_in_past: Signal<Boolean> :=
  inPast(1000000, data_processed)

define error: Events<Boolean> :=
  eventNot(sample(consumed_in_past, delayed_new_data))
```

**Listing 8.3:** TeSSLa specification describing a performance constraint. The specification describes that after each change of the *write_ptr* variable there has to be a call to the *process* function after at most 1 millisecond. Since TesslaServer only implements the past time functions of TeSSLa the specification is a bit more complex than it would be when using funvtions that can depend on future values.

# List of Figures

# List of Tables

# Glossary

**Lola** A specification language and algorithms for the online and offline monitoring of synchronous systems including circuits and embedded systems. 1, 6, 7, 51

**Rcat** Requirements CApture Tool. 8

**Rmor** Requirement Monitoring and Recovery. 1, 8

**TeSSLa** A temporal, stream based specification Language. 1, 2, 5–8, 11, 19–22, 27, 30, 32, 38, 43, 49, 51, 53, 55, 58, 61, 62, 65, 67–69, 71, 89–91

**API** Application Programming Interface. 51, 53, 57, 59

**BEAM** Bogdan/Björn's Erlang Abstract Machine. 49, 50

**DAG** Directed Acyclic Graph. 21, 38–40, 42, 47, 51, 52

**FIFO** First In First Out. 21

**IR** Intermediate Representation. 59

**ISP** Institute for Software Engineering and Programming Languages. 1

**JSON** JavaScript Object Notation. 52

**LLVM** Low Level Virtual Machine. 2, 58, 59

**LTL** Linear Temporal Logic. 51

**OTP** Open Telecom Platform. 49, 53

**RAM** random access memory. 56, 63–65, 91

**RV** Runtime Verification. 1, 2, 7, 8

**TL** Temporal Logic. 1, 6

**VM** Virtual Machine. 49, 50

# List of Theorems