An asynchronous evaluation engine for stream based specifications

Alexander Schramm

November 20, 2016
Version: My First Draft

University of Luebeck

UNIVERSITÄT ZU LÜBECK

Institute For Software Engineering and Programming Languages

isp

Master Thesis

An asynchronous evaluation engine for stream based specifications

Alexander Schramm

1. Reviewer Prof. Dr. Martin Leucker
Institute For Software Engineering and Programming Languages
University of Luebeck

2. Reviewer Who Knöws
We will see
University of Luebeck

Supervisors Cesar Sanchez

November 20, 2016

Alexander Schramm
An asynchronous evaluation engine for stream based specifications
Master Thesis, November 20, 2016
Reviewers: Prof. Dr. Martin Leucker and Who Knöws
Supervisors: Cesar Sanchez

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

Program 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 Verifications 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, like LTL formulas or in specification languages that are specifically developed for runtime verification. One field of RV is looking at verifying behviour of streams of data, specifying relationships of values on those streams. Examples for this are RMoR, Lola and TeSSLa, which we will look at more closely in Section 2.

The language TeSSLa aims to make it easy to specify behaviour of streams. To gain this it introduces a number of language features and syntax sugar to expressively describe the conditions a stream should fulfill. The evaluation of TeSSLa specifications is done in two steps: first the specification is compiled by a compiler written at the ISP of the University of Luebeck. 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 (see RMoR) 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 ([DAn+05]).

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

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 an 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 and more.

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

1.2 Results

1.3 Thesis Structure

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

Chapter 2

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

Chapter 3

Moving towards the implementation of the new system, in this chapter practical concepts and systems that are used for implementing and evaluating the runtime are shown. It is explained how they are used and which alternatives exist.

Chapter 4

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

Chapter ??

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

2 Related Work

As Runtime Verification is a widely researched field there are many different approaches towards monitoring programs. TeSSLa itself and the implemented runtime builds on concepts and results of many of them. In the following section some of them are highlighted to give a better understanding of choices made during this thesis.

2.1 LOLA

LOLA [DAn+05] may arguably have the biggest influence on TeSSLa and the theoretical work of this thesis. LOLA defines a very small core language to describe streams as the result of combinations of other streams of events. In contrast to TeSSLa it is defined in regards of a discrete timing model.

Lola defines a notion of efficiently monitorable properties and an approach to monitor these properties.

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 applies to streams.

2.2 Copilot

The realtime runtime monitor system Copilot was introduced in [Pik+10]. 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. [Pik+10]

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.

2.3 RMoR

2.4 Driver Trace

2.5 Debie

2.6 MaC

2.7 RiTHM

3 System

3.1 Evaluation Engine

3.1.1 Erlang and Elixir

todo: BEAM, Actors/Thread, multiplattform (nerves project)

3.1.2 Implementation

todo: Timing model: reason why events have to carry timestamps in contrast to interweaved monitors

3.2 Trace Generation

3.2.1 TraceBench

3.2.2 Aspect oriented programming

3.2.3 CIL

3.2.4 Google XRay

3.2.5 GCC instrument functions

3.2.6 Sampling

3.2.7 LLVM/clang AST matchers

4 Concepts

4.1 Definitions

In the following definitions are given to reason about semantics of implementations of a TeSSLa runtime. A TeSSLa specification gives a number of transformations over input streams and a subset of the generated streams as outputs. Streams can be queried for the value they hold at a specific time.

4.1.1 Streams

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, which hold a value and a timestamp: $(v_1, t_1), (v_2, t_2), \ldots$. Signals can be described by a sequence of changes, which hold a value and a timestamp: $(v_1, t_1), (v_2, t_2), \ldots$. This shows that the only difference between Signal and EventStreams is given, when we query the stream for it's value: A Signal will always have a value, an EventStream may return $\perp$, which denotes, that no event happened at that time. Because the similarity of Signals and EventStreams in the following we will mainly reason about EventStreams, but most things can also be applied to Signals.

Futhermore Signals and EventStreams hold the timestamp to which they have progressed, which can be equal or greater than the timestamp of the last Event happened on them.

4.1.2 Events

Streams consists of Events (or changes, which can be modeled the same as events). There are three Types of Events: Input, Output and Internal events.

Let E be the Set of valid input events (E for external) and $e \in E$, where each event carries a value, which can be *nothing*, a timestamp and the stream it's perceived on (e.g. a function call of a specific function). Further let O be the Set of valid output events and $o \in O$, which have the same properties than input events, while their channel is specified by the TeSSLa specification.

Internal events are mostly an implementation detail, which denotes steps of computation inside the runtime: Let the Set of valid internal events be N. Internal events also carry a value and a timestamp, but their stream is implicitly given by the node that produces the event.

4.1.3 Functions

A TeSSLa specification consists of functions which manipulates streams and generate new 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 directly be given to another function (function composition).

4.1.4 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.

Because Functions in TeSSLa specifications itself depend on other Functions, and these dependencies have to be circle free, the specification can be represented as a DAG and the Nodes are also organized as a DAG. Each Node has a State that can hold arbitrary data used for computation. One part of the State is the History of a Node, which holds all Events received from it's predecessors, called it's inputs, and all produced Events of the Node, called it's output.

Nodes use a FIFO queue, provided by the Erlang plattform, to process new received Events in multiple steps:

1. Add the new Event to the inputs
2. Check if a new output Events can be produced (see Section 4.1.4)
3. If so, compute all timestamps, where new Events might be computed and
    a) Compute the Events, add them to the History as new outputs
    b) Distribute the updated output to all successors
4. Else wait for another input

Determination of processable Events

Based on the asynchronous nature of Nodes, Events from different channels can be received out of order. E.g. if a Node C is a child of Node A and B, it can receive Events from Node A at timestamps t_1, t_2, t_3, t_4 before receiving an event with timestamp t_1 from Node B. Therefore a Node can not compute it's output upto a timestamp unless it has informations from all predecessors that they did progress to that timestamp. When Node C receives the first four Events from Node A, it will only add them to it's inputs but won't compute an output. When it finally receives the first Event from Node B it can compute all Events upto t_1. To do so it will compute *change timestamps*: The union of all timestamps where an Event occured on any input between the timestamp of the last generated output and the minimal progress of all inputs. To see why this is necessary lets assume that Node C will receive a new Event from Node B with timestamp t_4: All inputs have progressed to t_4, but on the stream from Node A there are changes between t_1 (where the last output was generated) and t_4, therefore the *change timestamps* are t_2, t_3, t_4 and the Node will have to compute it's output based on the values of the streams at that timestamps.

4.1.5 TeSSLa Evaluation Model

Based on the format of TeSSLa specifications a specific structure for evaluation is given: A DAG that consists of Nodes, where the roots are the ones consuming external Events and the leaves are the ones producing outputs. The Nodes in the DAG are called *ready* when they have at least one input buffered for all of their predecessors, meaning they are able to perform a computation and produce a new output. In later sections the equality of different schedules of Nodes are shown.

4.1.6 State

All TeSSLa runtimes have to have a State, which encodes information necessary to compute further Events. The State of a whole Evaluation Engine is made up of the States of it's Nodes. The State of a Node is made up of arbitrary Information needed to perform later computations, the events it generated and the queue of new inputs.

4.1.7 Transitions

Transitions describe the Steps between two States an evaluation Engine takes: The consumption of an Event from all inputs of a Node and the production and propagation of a new output from that Node.

4.1.8 Run

A Run of an Evaluation Engine is a Series of States and Transitions. It is a representation of the steps the Engine takes to evaluate a specification over streams. The run $s_0\theta_1 s_1$ means, that the Engine was in it's initial State, took the transition θ_1 and thereby reached the state s_1.

4.2 Behaviour of different evaluation strategies without timing functions

For a first step we specify and compare behaviours of different approaches to evaluate TeSSLa specifications without timing formulas, meaning that only functions, which manipulate values or the presence of events, but not the timestamp of them, are used. This leads to behaviours that can be easily reason about, as seen in the next sections.

All Evaluation Engines compute Events in steps. Each step a Node is scheduled to perform it's specific Task, therefore one of the following things can happen in each step.

- An input Event can be consumed by a source in the DAG, which generates an internal event that is propagated to it's children
- An internal Node which has at least one new input buffered on all of its input queues can perform it's computation and generate a new internal event, which is propagated to the children of that node, which therefore can compute in the next step.
- An output node, which has at least one new input buffered on its input queue, can produce a new output.

The difference between Evaluation Engines is the way they are scheduling the Nodes. In the following we will define different ways of scheduling and proove equality between them.

4.2.1 Synchronous evaluation

An synchronous System I for a specification T is one that has a fixed schedule build like this: Number all Nodes in a reversed topological order once, then always schedule the ready Node with the lowest number. Obiously for many DAGs there is no unique reversed topological order, therefore one has to be chosen. This schedule ensures that no new events are consumed by any predecessor of a Node that is ready itself, therefore Events are *pushed* through the DAG towards an output Node as fast as possible.

The synchronous system can be considered as the *Source of Truth*: The Relationship between inputs and outputs it generates is assumed to be the right evaluation for a given TeSSLa specification. All other approaches must generate a relationship between inputs and outputs that can be transformed into the one from the synchronous system, or else the different approach is not seen as a valid system for the specification.

Figure 4.1 visualizes a synchronous System. It shows two DAG representations of an evaluation Engine where the Nodes A to E are labeled in a reversed topological order and o_1 and o_2 represents the output channels with that name. The left System is in it's initial State and an input event $\langle e_1, t_1 \rangle$ is ready to be consumed. When a Node is chosen to compute by the scheduler, only Node A is ready, 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 $\langle n_A, t_1 \rangle$ which is propagated to all it's children: Node B and D. In the next step Node B would be scheduled, because it has the lowest number of any node that can compute (actually it's the only node that can compute at all, because D has to wait for the event from B). After B was scheduled, it would have produced the internal Event $\langle n_B, t_1 \rangle$ which would then be distributed to Nodes C and D. Therefore the trace of consumed external and produced internal and output events by the system with this specific schedule could be visualized as:

$$\langle e_1, t_1 \rangle \langle n_{A1}, t_1 \rangle \langle n_{B1}, t_1 \rangle \langle n_{C1}, t_1 \rangle \langle o_{1,1}, t_1 \rangle \langle n_{D1}, t_1 \rangle \langle n_{E1}, t_1 \rangle \langle o_{2,1}, t_1 \rangle$$

If there were more than one input event, at this point Node A would be scheduled again, consume the next external event and the following nodes would be scheduled in the same order as before, extending the trace in an obvious way.

4.2.2 Asynchronous evaluation

An asynchronous system A for a specification T is a Evaluation Engine with a fair, but not fixed schedule.

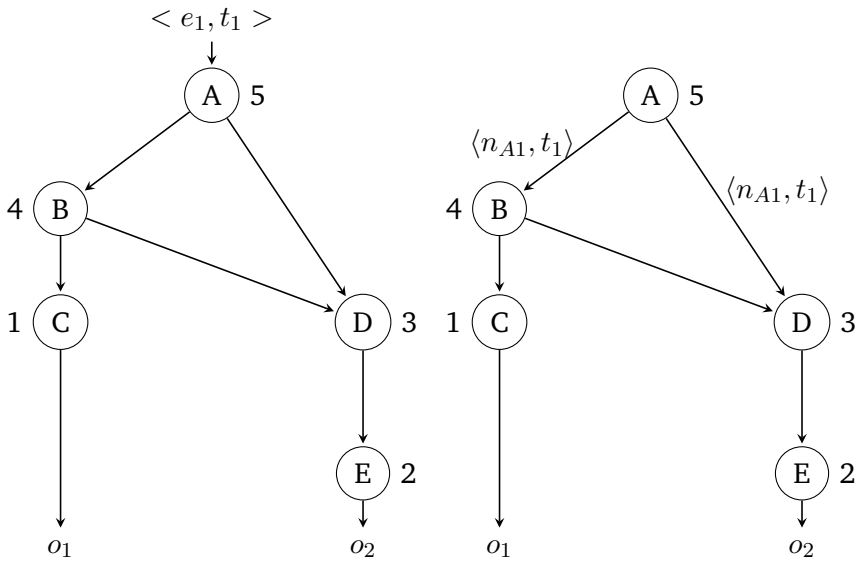

Fig. 4.1.: Visualization of a simple asynchronous system with a reversed topological order.

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

4.3 Equalitys of different Systems without timing functions

Based on the described behaviours of the approaches we now can proof the equality of them.

As already stated in Section 4.2.1 a synchronous evaluation engine is regarded as the source of truth, therefore all other kinds of evaluation engine have to be equal to one.

The equality is shown in two steps: First in Section 4.3.1it is shown, that all possible synchronous Systems for a specification are equal, so there is only one true evaluation. Afterwards in Section 4.3.2 it is shown that any asynchronous evaluation engine is equal to a synchronous one.

4.3.1 Equality of synchronous Systems

When given a series of input events, two synchronous evaluation Engines with different schedules will generate different runs, but both will produce all outputs that

can be produced after consuming one specific input before the next Input is consumed as reasoned in Section 4.2.1.

To proof the equality of both systems we have to proof the equality of their runs. To do this we will show that any two runs of two synchronous system can be iteratively reordered without violation of causality until they are equal.

Let $\vec{e} = (e_1, e_2, \ldots, e_x)$ be the input events both implementations receive. Furthermore let R_1, R_2 be the runs of the two Systems for a given TeSSLa specification.

Because each TeSSLa specification contains only a finite amount of functions and works on finite traces, the runs also have to be finite. When two Systems have a different schedule, their Runs will be different at a finite number of positions. Let R' be the finite prefix of both runs that are equal (This will be at least s_0, but possible more) and i_d the index of the first difference. This means that at Step i the second evaluation engine has taken a different transition, meaning a different Node was scheduled at that point, than the first evaluation engine.

Let N_1 be the Node scheduled by the first evaluation Engine and N_2 by the second. Because N_1 was scheduled by the first evaluation engine, it also has to be ready in the second engine, and because it wasn't scheduled by it, it has to still be ready after that step. The same holds for N_2 in the first evaluation engine. After step i_d both system might take a finite number of different transitions, but at some point the first System has to schedule Node N_2 and the second system Node N_1, because there are only a finite number of Nodes with a lower number and a Node can only become *not ready* by performing it's computation. Let $i_{N_2} > i_d$ be the index of the step where the first System schedules the Node N_2. Let N_b be the set of all Nodes which were scheduled between i_d and i_{N_2}. None of theese Nodes can be a children of N_2, because otherwise that Node couldn't be scheduled before N_2, because it would have had to wait for an Event from N_2. Now let $N_c \in N_b$ be the Node that was scheduled at step $i_{N_2} - 1$ and $r_s = (\langle \tau_{i_{N_2}-2}, s_{i_{N_2}-2} \rangle, \langle \tau_{i_{N_2}-1}, s_{i_{N_2}-1} \rangle \langle \tau_{i_{N_2}}, s_{i_{N_2}} \rangle)$ the suffix of the run from the first system from two steps before N_2 was scheduled upto the point where it was scheduled.

Figure 4.2 shows how changing the order of the Nodes N_c, N_2 has no influence on the state after both have executed. This is, because none of the two Nodes can be a children of the other, therefore only the state of other Nodes can change when one of them computes (by adding their generated events to their input queue). If both Nodes have the same child, it will receive the inputs in different order, but because they're happening on different channels it doesn't matter.

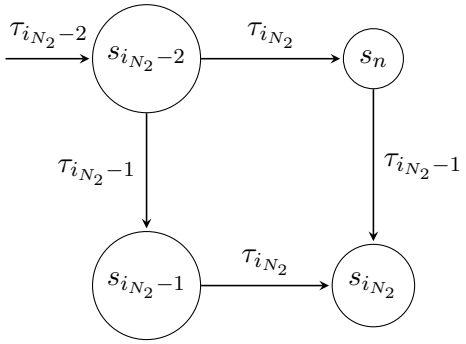

Fig. 4.2.: Commutativity Diagramm of Node scheduling

4.3.2 Equality of synchronous and asynchronous Systems

When the Nodes of A aren't scheduled in reversed topological order, the system can consume Inputs before producing all outputs based on the last consumed input. Therefore the reordering of outputs and inputs (and internal events) has to be performed over the whole trace, not only between Input events. Idea: each step is a commutation of two internal events in regard to the rev top order. => show commutativity of traces (note: only valid commutations, no two events, where one depends one the other, can be commuted, this is ensured by the scheduling of nodes that have input buffered)

Bibliography

[DAn+05] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, et al. „LOLA: Runtime monitoring of synchronous systems". In: *Proceedings of the International Workshop on Temporal Representation and Reasoning* (2005), pp. 166–175 (cit. on pp. 2, 5).

[Pik+10] Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. „Copilot: A hard real-time runtime monitor". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6418 LNCS.Rv (2010), pp. 345–359 (cit. on pp. 5, 6).

List of Figures

List of Tables

Example Appendix

A

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

A.1 Appendix Section 1

This is the second paragraph. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Alpha	Beta	Gamma
0	1	2
3	4	5

Tab. A.1.: This is a caption text.

A.2 Appendix Section 2

And after the second paragraph follows the third paragraph. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information

about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Alpha	Beta	Gamma
0	1	2
3	4	5

Tab. A.2.: This is a caption text.

After this fourth paragraph, we start a new paragraph sequence. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Colophon

This thesis was typeset with LaTeX 2_ε. It uses the *Clean Thesis* style developed by Ricardo Langner. The design of the *Clean Thesis* style is inspired by user guide documents from Apple Inc.

Download the *Clean Thesis* style at `http://cleanthesis.der-ric.de/`.

Declaration

You can put your declaration here, to declare that you have completed your work solely and only with the help of the references you mentioned.

Luebeck, November 20, 2016

Alexander Schramm