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

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 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. Examples for this are RMoR ([Hav08]), Lola ([DAn+05]) and others ([Zhe+15; Pik+10; MB15]), 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 (e.g. [Hav08]) 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 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 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.

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

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

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

1.2 Results

1.3 Thesis Structure

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

Chapter 2

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

Chapter 3

Moving towards the implementation of the new 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

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

As Runtime Verification is a widely researched field there are many different approaches towards monitoring programs. As stated in [Hav08] most approaches are geared towards Java, while many critical systems are written in C. Because our implementation of a runtime is independent of the environment of the monitored program, this opens up the possibility to investigate this somewhat neglected field. Therefore ways to generate traces from C programs are highlighted in this chapter.

TeSSLa itself and the implemented runtime builds on concepts and results of many other Runtime Verification techniques. In the following sections some of them are highlighted to give a better understanding of choices made during this thesis.

2.1 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 [MB15] a way to monitor distributed programs is given. To do this distributed monitors, which have to communicate with one another, are implemented.

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

2.2 LOLA

LOLA [DAn+05] has big 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 on other streams. 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 applied to streams.

2.3 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.

2.4 RMoR

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

Specifications are given as a textual representation of state machines. The specifications are then interweaved into the programm using CIL [Nec+02]. Specifications work on the level of function calls and state properties like *write may never be called before open was called*. Because Software Developers are often working at the same abstraction level (in contrast to e.g. assembler or machine instructions), they can define specifications without having to learn new concepts. For the TeSSLa runtime support for traces at the same abstraction level (function calls, variable reads and writes) is present and used in most of the tests in Section **??**.

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

2.5 MaC

3 System

Besides the theoretical basics presented in Section 2 the TeSSLa runtime of this thesis is built upon a number of technologies. To better understand decisions made during the implementation this Chapter will give an overview of them and show why they were choosen.

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

3.1 TeSSLa Runtime

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

3.1.1 Erlang and Elixir

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 Debie

3.2.2 TraceBench

3.2.3 Aspect oriented programming

3.2.4 CIL

3.2.5 Google XRay

3.2.6 GCC instrument functions

3.2.7 Sampling

3.2.8 LLVM/clang AST matchers

4 Concepts

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 Streams and a subset of the generated streams as outputs.

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. Signals can be described by a sequence of changes, where a change notes 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 $\perp$, 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.

Formally a stream σ is defined as a Sequence of Events, the Set of all Streams Σ is defined as all possible finite sequences of Events $\Sigma = \{\sigma | \sigma \in E^*\}$ An input Stream σ_i is a stream consisting of only input events, the set of all input Streams is $\Sigma_i = \{\sigma_i | \sigma_i \in E_i^*\}$. Output and internal Streams are defined analogous.

Furthermore Streams hold the timestamp to which they have progressed, which can be equal or greater than the timestamp of the last Event happened on them.

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.

The Set of all Events is denoted as E. Each event carries a value, which can be *nothing* or a value of a Type (types are formally defined in the TeSSLa speicification

but aren't important for this thesis), a timestamp and the channel it's perceived on (e.g. a function call of a specific function or the name of an output stream).

$E_i \subset E$ is the Set of all input events, their channel corresponds to a specific trace. $E_o \subset E$ is the Set of all output events, their channel 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 channel of internal events is implicitly given by the node that produces the stream of the Event. Note that E_i, E_o, E_n are pairwise disjoint and $E_i \cup E_o \cup E_n = E$.

4.1.3 Functions

A TeSSLa specification consists of functions. Functions generate new Streams by applying operation on other stream. 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).

Functions can be divided into three categories: pure, unpure and timing. Pure Functions, also called stateless, are evaluated only on the values their inputs have at the timestamp they are evaluated, therefore they don't have to *remember* anything about earlier events. Unpure, or stateful, Functions are evaluatet with additional, stored information, which they can mutate. E.g. a Function *eventCount* has to *remember* how many events already happened on it's input stream and increment that counter on every new event. Timing Functions are evaluated not only on the value of Events but also on their timestamp and can also manipulate it: While non timing functions will consume events at a specific timestamp and emit an event with that timestamp, timing functions can emit Events with a different timestamp.

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

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. Each Node has a set of inputs, which are either input or internal Streams, and one output, which is either an internal or an output Stream.

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 Event 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 Streams can be received out of order. E.g. if a Node C is a child of Node A and B, it can receive Events from Node A at timestamps t_1, t_2, t_3, t_4 before receiving an event with timestamp t_1 from Node B. Therefore a Node can not compute it's output upto a timestamp unless it has informations from all predecessors that they did progress to that timestamp. When Node C receives the first four Events from Node A, it will only add them to it's 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 Engine

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

A Node in the DAG is called *ready* when it has at least one event buffered on all of it's inputs, meaning it is able to perform a computation and produce a new output.

A Node in the DAG is called independent of another Node, if it is no descendant of that Node. This means the Node doesn't depend on the Events emitted by the other Node.

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

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

4.1.6 State

All TeSSLa evaluation Engines have to hold a State, which encodes information necessary to continue the evaluation. The State of a whole Evaluation Engine is made up of the States of it's Nodes.

Each Node has a State that can hold arbitrary data used for computation. One part of the State of every Node is the History of a Node, which holds all Events received from it's parents, called it's inputs, and all produced Events of the Node, called it's output. Other information stored in the State may be the number of Events seen by an *eventCountNode* which is incremented everytime the Node consumes a new input.

Formally the State of a Node is a tuple of some arbitrary information, the inputs (a set of internal and input Streams) and the output (an internal or an output Stream). Everytime a Node is scheduled it changes it State by:

- Updating it's inputs
- updating it's output
- optionally updating its arbitrary information

4.1.7 Transitions

A Transition describes what happens when the evaluation engine switches State by performing a Step: The consumption of one Event from each input of a Node and the computation of a new output from that Node. Formally a Transition maps a sequence of Events to one Event. The empty transition, meaning no input was consumed or produced, is labeled with λ.

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

4.2 Behaviour of different schedules without timing functions

For a first step we specify and compare behaviours of different approaches to evaluate TeSSLa specifications without timing 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. At each step a Node is scheduled to perform it's specific operation, 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

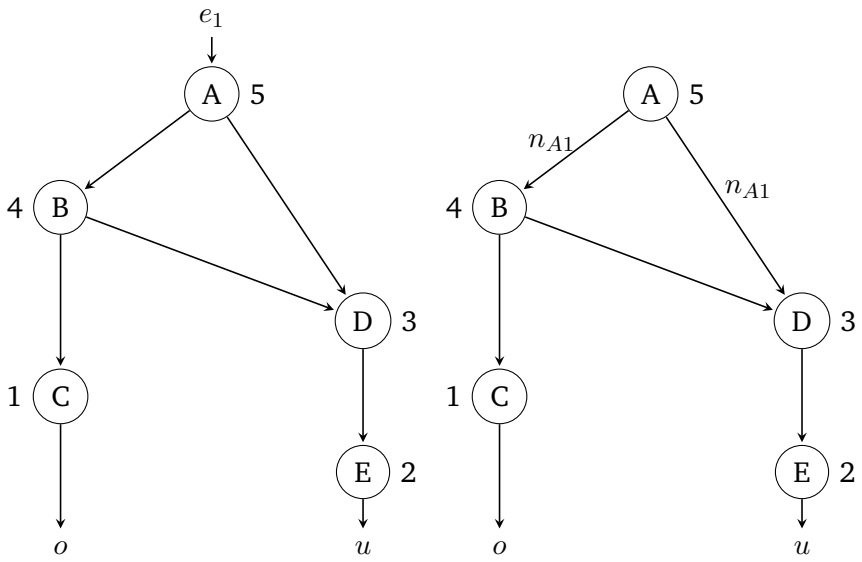

Fig. 4.1.: Visualization of a simple asynchronous system with a reversed topological order.

- 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.

Evaluation Engines are free in the way they are scheduling their Nodes. In the following we will classify Evaluation Engines by their scheduling approaches.

4.2.1 Synchronous Evaluation Engines

An synchronous Evaluation Engine is one that has a fixed schedule, build like this: Number all Nodes in a reversed topological order, then always schedule the Node with the lowest number that is ready. Obviously for many DAGs there is no unique reversed topological order, therefore one can be chosen by the evaluation engine.

This schedule ensures that no Node is scheduled which has a successor that can be scheduled, therefore Events are *pushed* through the DAG towards an output Node as fast as possible.

A synchronous Evaluation Engine is considered as the *Source of Truth*: Any other Evaluation Engine has to be equal to one, else it is not a valid evaluation engine.

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 and u represents the output channels with that name. The left System

is in it's initial State and an input event e_1 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 n_{A1} which is propagated to all it's children: Node B and D. In the next step Node B would be scheduled, because it has the lowest number of any node that can compute (actually it's the only node that can compute at all, because D has to wait for the event from B). After B was scheduled, it would have produced the internal Event n_{B1} which would then be distributed to Nodes C and D.

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

$$\langle(\lambda, s_0), (\langle n_{A1}\rangle \to n_{B1}, s_1), (\langle n_{B1}\rangle \to o_1, s_2),$$
$$(\langle n_{A1}, n_{B1}\rangle \to n_{D1}, s_3), (\langle n_{D1}\rangle \to u_1, s_4)\rangle$$

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

4.2.2 Asynchronous evaluation

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

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

4.3 Equalitys of different schedules without timing functions

Based on the described behaviours of the approaches we now can proof the equality of different Schedules for an Engine. Two Engines for a TeSSLa specification are equal, if they produce the same outputs for the same inputs. Because the consumed inputs and produced outputs can be reconstructed from a Run, we can show equality by showing that Runs are equal.

Two Runs are assumed to be equal if the State of their last elements are equal. Because all generated Events are saved in the State of an Engine, if two Runs have the same State in their last element they did produce the same outputs.

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 Engines 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 Engines 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 have different Runs. But both will produce all outputs that can be produced after consuming one specific input before the next Input is consumed as reasoned in Section 4.2.1.

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 changing the State of the last Element of the Run until they are the same Run. When two Runs can be reordered in this way they had to be equal from the beginning, because the last state was never altered.

I think this could be stated more clearly, but it's good enough for now

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 synchronous Engines 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 (λ, s_0), but possible more) and i_d the index of the first difference. This means that at Step i_d 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 the one scheduled by the second. Because N_1 was scheduled by the first evaluation Engine and upto i_d the two Engines took the same steps 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.

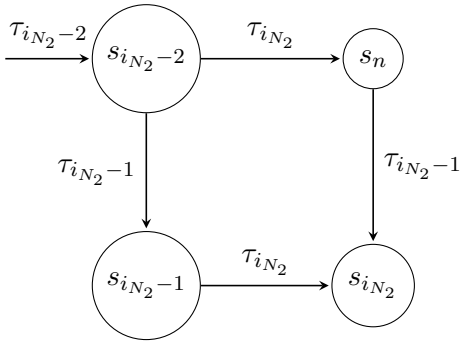

Fig. 4.2.: Commutativity Diagramm of Node scheduling

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}. All of theese Nodes have to be independent of N_2, because otherwise they couldn't be scheduled before N_2. On the other hand N_2 has to be independent of all Nodes in N_b because otherwise it couldn't have been scheduled before them in the second evaluation Engine.

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 infix of the Run of the first Engine, starting two steps before N_2 was scheduled upto the point where it was scheduled.

Figure 4.2 visualizes how changing the order of the Nodes N_c, N_2 has no influence on the global state after both have executed. This follows from the fact, that both Nodes are independent of each other as shown before. This means that the computation of one of them can't change the State of the other, thereby having absolutely no consequence on the computation of the other. If both Nodes have mutual children, that children will receive their inputs in different order, but because the events are on different channels it doesn't matter. Therefore it doesn't matter which of the both Nodes is scheduled first: the state of the system will be different only at one position, but afterwards they both will have the same State again. If one of the Nodes generate an output event, that event will be emitted one step earlier or later, which is also not relevant for the global State.

The praragraph is a bit handwavy, needs more theory

Here the induction step should happen, right?

4.3.2 Equality of synchronous and asynchronous schedules

When the Nodes of A aren't scheduled in reversed topological order, the system can consume Inputs before producing all outputs based on the last consumed Input. Therefore the reordering of runs has to be performed over wider parts of the run.

4.4 Behaviour with Timing functions

4.5 Equalitys with Timing functions

sectionParallel computation

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. 1, 2, 5).

[Hav08] Klaus Havelund. „Runtime Verification of C Programs". In: (2008) (cit. on pp. 1, 5).

[MB15] Menna Mostafa and Borzoo Bonakdarpour. „Decentralized Runtime Verification of LTL Specifications in Distributed Systems". In: *2015 IEEE International Parallel and Distributed Processing Symposium* (2015), pp. 494–503 (cit. on pp. 1, 5).

[Nec+02] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. „CIL: Intermediate language and tools for analysis and transformation of C programs". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 2304 (2002), pp. 213–228 (cit. on p. 7).

[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. 1, 6).

[Zhe+15] Xi Zheng, Christine Julien, Rodion Podorozhny, and Franck Cassez. „BraceAssertion: Runtime verification of cyber-physical systems". In: *Proceedings - 2015 IEEE 12th International Conference on Mobile Ad Hoc and Sensor Systems, MASS 2015* (2015), pp. 298–306 (cit. on p. 1).

List of Figures

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