Implementation and Semantics of 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

Implementation and Semantics of 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
Implementation and Semantics of 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 a field of computer sience that is widely researched and has multiple approaches. 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, external executions of monitors can't directly modify the program. 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 ([D'A+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 proofen that the actual approach of this runtime, a functional, actor based, asynchronous system, will generate the same observations on input traces as an ideal 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.

To test the software based runtime, different specifications will be tested on multiple traces, some of which are generated by actual 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

Chapter 2

Chapter 3

Chapter ??

Chapter ??

Chapter ??

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 [D'A+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 a independet process. The implementation of the TeSSLa runtime in this implementation follows a similar approach, by being 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, therefore working like Copilot, or by actually instrumenting the code to generate the traces, which alters the semantics of the program.

2.3 RMoR

2.4 Driver Trace

System 3

3.1 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.

3.1.1 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: $(v_1, t_1), (v_2, t_2), \ldots$. Values 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 cam also be applied to Signals.

3.1.2 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, a timestamp and the strea, it is 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.

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.

3.1.3 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 $add(S_D, S_D) \to S_D$: 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).

3.1.4 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.

3.2 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 stated, as seen in the next sections.

3.2.1 Ideal synchronous evaluation

An ideal system I for a specification T is one that consumes an input event e_x and immediately emits appropriate output Events $o_{1,1}, o_{1,2}, \dots, o_{1,x}$ with $x \in \mathbb{N}_{\geq 0}$ and

Timestamp:	t_0	t_1	t_2	t_3
Input:		e_1	e_2	e_3
State:	I_0	I_1	I_2	I_3
Outputs:		$(o_{1,1}, \ldots, o_{1,x})$	$(o_{2,1}, \ldots, o_{2,y})$	$(o_{3,1}, \ldots, o_{3,z})$

Fig. 3.1: Example computation of 3 Inputs by an ideal implementation

changes it's internal state S to save the fact that the event e_x was received and optionally other properties generated by the evaluation of the spec that are needed by later computations.

Obviously this approach can not be implemented in practice, but it is a good base to consider equalitys of different approaches. The Ideal 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 ideal system, or else the different approach is not seen as a valid system for the specification.

The behaviour for a single input can be described by two functions:

$$\Phi : S \times E \to S$$
$$\Theta : S \times E \to O^*$$

The behaviour for multiple inputs is the composition of the functions.

The trace for the implementation I for a Stream of input events from E^* is given by the output Stream from O^* that is generated by the formulas.

The processing of a series of 3 input events of an ideal system can be visualized like shown in Figure 3.1.

3.2.2 Asynchronous evaluation

An asynchronous system A for a specification T is an abstraction of an actual implementation of a runtime. The actual Implementation consists of multiple Nodes, where each one implements one formula of the TeSSLa specification. Each Node is run as it's own process and newly generated events are passed as messages to Nodes that work on them.

The asynchronous system doesn't consider a number of implementation details of the actual implementation, eġthe needed computation duration of a Node: Each Node takes exactly one step to produce a new output based on inputs.

The asynchronous system has a more complex behaviour than the ideal system because Nodes have to wait for other Nodes that are before them and there are multiple Nodes that can produce outputs.

In contrast to I the asynchronous system takes multiple steps to produce an output from an input. During a step multiple things can happen at once:

- A new, external 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 its input queue 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.

To reason about the steps A takes while performing a computation, we have to define a schedule of the computations of the nodes. This schedule has to be fair, otherwise it's possible that no output would ever be created, because the output nodes will never be scheduled. Such a schedule is given by the following steps:

- Number the nodes in reversed topological order
- schedule the node with the lowest number that has an input event buffered

As shown in Section **??** this is a fair schedule.

Figure 3.2 shows two DAG representations of an asynchronous system where the Nodes A to D 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

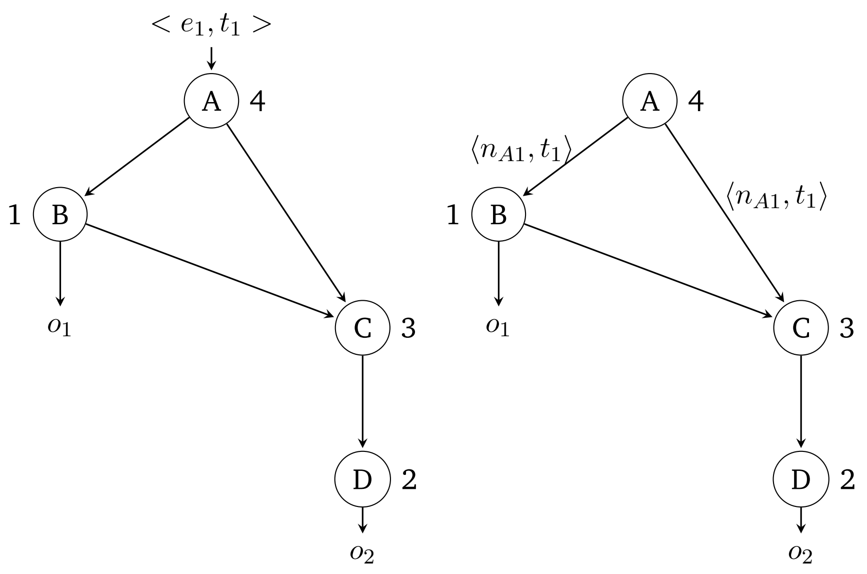

Fig. 3.2: Visualization of a simple asynchronous system with a reversed topological order.

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 proppagated to all it's children: Node B and C. 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 C ahs 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 emitted as the output event $\langle o_1, t_1 \rangle$. Therefore the trace of consumed external and produced internal and output events by the system with this specific schedule could be visualized as:

$$\langle I_1, t_1 \rangle \langle n_{A1}, t_1 \rangle \langle n_{B1}, t_1 \rangle \langle o_{1,1}, t_1 \rangle \langle n_{C1}, t_1 \rangle \langle n_{D1}, t_1 \rangle \langle o_{2,1}, t_1 \rangle$$

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.

3.3 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 3.2.1 the System A has to produce an equal relationship of inputs and outputs as I to be a valid implementation of the specification.

The equality is shown in two steps: First it is shown, that the computation of A with a fixed, reversed topological schedule is equal to I in Section 3.3.1. Afterwards in Section 3.3.2 it is shown that A with any fair schedule is equal to A with the fixed, reversed topological schedule, and therefore equal to I.

3.3.1 Asynchronous system with topological schedule and ideal system

When given a series of input events, A won't emit the outputs in the same order as I, but the reversed topological schedule will assure that all outputs that can be produced after consuming a specific input will be produced before the next Input is consumed as shown in the next sextion. Therefore, to make the relationship between inputs and outputs the same as the one from I, only the outputs between each consumed input event have to be reordered. Stated otherwise: If I and A consume exactly one input, they both will produce exactly the same outputs, only in different order.

The following paragraphs will give a more precise notion of the equality between the systems.

To proof the equality of both systems we have to proof the equality of the traces A produces, while computing the outputs, to the traces I generates. Let $e_1, e_2, \ldots, e_x$ be the input events both implementations receive. The trace of I was shown in Section 3.2.1 and the traces of A were shown in Section 3.2.2.

Let's first reason about the easy case, were only one external event is received: In this case I has only two states, I_0, I_1 and will only produces outputs once: $o_{1,1}, \ldots, o_{1,x}$. A will walk through multiple states, bound by the number of functions in the specification it's implementing. Because the specification is a DAG, especially has no circles, eventually A has to finish it's computation, let the number of steps be k. Let $h_{\min}$ be the minimal height of the DAG (the height of the leave with the fewest steps to the Source). The first output can only be created after $h_{\min}$ steps and after step k all outputs have to be created, so that $o_{1,1}, \ldots, o_{1,x}$ were emitted. Let the States of A during the computation be $A_0, A_{1,0}, A_{1,1}, \ldots, A_{1,k}$ where A_0 is the initial state.

When taking the first step and $A_{1,0}$ is reached, the event e_1 is consumed by a source node N_A, which produces the internal event $\langle n_{A,1}, t_1 \rangle$ and distributes it to it's children. Note that there is no alternative to this bahivour, because there were no prior events and therefore no internal Node has an event to process on it's input queue, therefore the Source consuming the external event is the only node that can, and therefore must, compute anything. Afterwards all Nodes that are direct children of the source that consumed the external event will have one input event buffered and are able to perform their computation in the next step.

At least until step $h_{\min}$ every step one or more Nodes, that are no outputs, will perform a computation, therefore pushing internal events closer to the output nodes. Somewhere between step $h_{\min}$ and k all internal events will reach an output node and produce an output.

A more complex case is when multiple external events are received.

Because the scheduling of nodes of A is based on the reversed topological order, A will only consume one new external event and then will schedule internal nodes until all internal events have reached an output node, only than the next input will be consumed by a source node. I will step through a series of States $I_0, I_1, \dots, I_x$, A on the other hand will step through a Series of states for each state that I takes: $A_0, A_{1,0}, A_{1,1}, \dots, A_{1,j1}, A_{2,0}, \dots, A_{x,j2}$ and generate internal events along those steps. Based on this behaviour lets define when a State of A is equal to one of I: The first states A_0 and I_0 are always assumed to be equal, because no output can be observed and therefore one can't observe a difference. A State $A_{i,j}$ with $j > 0$ is called equal to a State I_i iff:

- The previous state $A_{i,j-1}$ was equal to I_i
- and either
  - No new output was generated at state $A_{i,j}$
  - or if a new output o is created it is equal to one of the outputs of I_i that wasn't generated before

A State $A_{i,0}$ is called equal to a state I_i iff

- The previous state $A_{i-1,x}$ was equal to I_{i-1}
- the input e_i was consumed at the step
- and the States $A_{i-1,0}$ to $Ai - 1, x$ together produced the same output like I_{i-1}

Naively speaking the System A moves through states $A_{i,0}, \ldots, A_{i,x}$ while it produces the same outputs as I produced when it reached state I_i, and as soon as the last output was produced consumes a new event and thereby takes state $A_{i+1,0}$.

Based on this, a series of states $\vec{A}$ is called equal to a series of States $\vec{I}$ if each state in $\vec{A}$ is equal to a state in $\vec{I}$. The System A is equal to the system I in regard to a series of inputs $\vec{e}$ iff all possible series of states it could take to process $\vec{e}$ are equal to $\vec{I}$.

3.3.2 Equality of asynchronous systems with different 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 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

[D’A+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, 3).

[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. 3, 4).

List of Figures

List of Tables

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