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. A specific 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 **??**.

The language TeSSLa aims to make it easy to specify behaviour of streams. To gain this it introduces a number of language features and sintax sugare 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. 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.

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

Chapter 2

Chapter ??

Chapter ??

Chapter ??

2 System

2.1 Definitions

There are three Types of Events: Input, Output and Internal.

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 channel 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 only used by the asynchronous system: Let the Set of valid internal events be N. Internal events also carry a value and a timestamp, but their channel is implicitly given by the node that produces the event.

2.2 Semantics of TeSSLa functions

Think of semantics of thinks like add, mrv, etc. Can these be defined independent of async/synchronous or for each. For synchronous they are partly defined in the TeSSLa spec

2.3 Semantics of an ideal, synchronous evaluation Engine

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

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. 2.1: Example computation of 3 Inputs by an ideal implementation

Therefore it's 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 semantics 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 can be visualized like shown in Figure 2.1.

2.4 Semantics of an asynchronous evaluation Engine

An asynchronous implementation A for a specification T has more complex characteristics: It's state is defined by the product of the States of it's nodes, where each node represents a primitive operation in the specification and the nodes are organized as a DAG.The output is specified by the concatination of outputs of the nodes that are marked as output nodes in the specification.

In contrast to I the asynchronous implementation 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, generate a new internal event, which is propa-

gated to the children of that node, which therefore could 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 be never scheduled. Such a schedule is given by the following algorithm:

- 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 2.2 shows two DAG representations of an asynchronous system where the Nodes A to D 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 $\langle e_1, t_1 \rangle$ 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 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, a different reversed topological order could habe labeled B with 3, C with 2 and D with 1, then D would have been scheduled instead of 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:

$$\langle I_1, t_1 \rangle \langle n_{A1}, t_1 \rangle \langle n_{B1}, t_1 \rangle \langle o_1, t_1 \rangle \langle n_{C1}, t_1 \rangle \langle n_{D1}, t_1 \rangle \langle u_1, t_1 \rangle$$

If there were more than one input events, 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 a obious way.

TODO Proove equality between different topological orders TODO also sthàbout multiple nodes computing at once == (one node at a time + fairness)

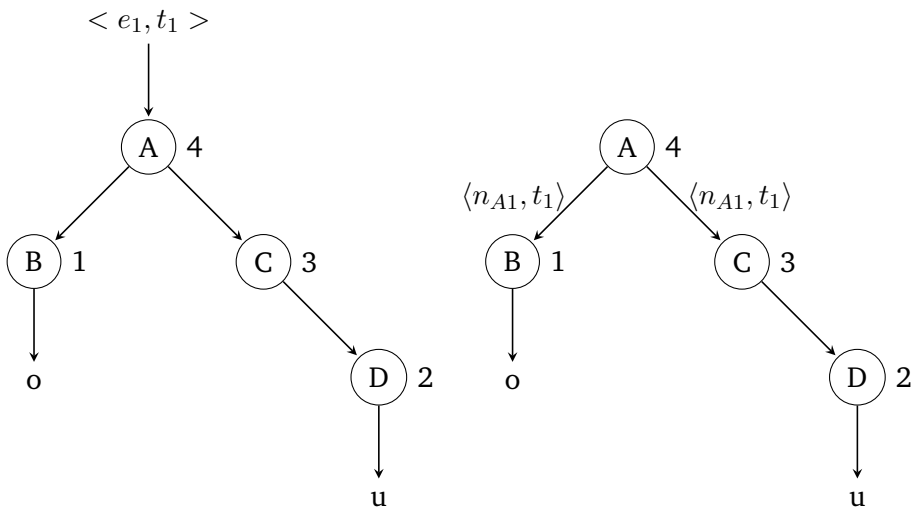

Fig. 2.2: Visualization of a simple asynchronous system with a reversed topological order.

2.5 Equality of Semantics for a topological schedule

A and S are considered equal if for a Series of input Events they produce the same output Events, or stated otherwise: The equality is defined only over the traces they consume and produce. A won't emit the outputs in the right order, but because each events carry the timestamp they were created at, the relationship between consumed input events and produced outputs based on that event can be restored. 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 states A will take, while producing the outputs, to the States I takes. Let $e_1, e_2, \ldots, e_x$ be the input events both implementations receive. The State of I will change every time a new input is received and will produce all outputs that could be computed.

Due to the asynchronous nature of A there is no direct mapping from the states of I to the states of A, because A will compute outputs in a non deterministic order. Therefore the equality has to be shown inductive over the possible States A could reach while computing the outputs.

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 and no other node can compute. 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 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 $I_0, I_1, \ldots, I_x$, A on the other hand will step through a Series of states for each state that I takes: $A_0, A_{1,0}, A_{1,1}, \ldots, A_{1,j1}, A_{2,0}, \ldots, A_{x,j2}$ 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 $\vec{I}$.

2.6 Semantics between different schedules for an asynchronous system

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)

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