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 System

1.1 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.

1.2 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

1.3 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.

Therefore it's behaviour for a single input can be described by two functions:

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

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.

1.4 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 propagated 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

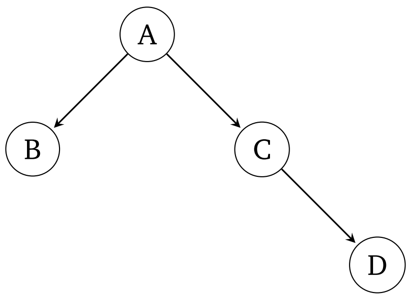

Fig. 1.1: Visualization of a simple asynchronous system

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:		$(o_{1,1},\dots,o_{1,x})$	$(o_{2,1},\dots,o_{2,y})$	$(o_{3,1},\dots,o_{3,z})$

Fig. 1.2: Example computation of 3 Inputs by an ideal implementation

- schedule the node with the lowest number that has an input event buffered

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

1.5 Equality of Semantics

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 event holds the timestamp it was generated, they can be reordered so that the output will be exactly the same as the one from I

1.5.1 Equality of States

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 $e_1, e_2, \dots, e_x$ 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. The processing of a series of 3 input events can be visualized like shown in Figure 1.2.

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: $o_{1,1}, \dots, o_{1,x}$. 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 $o_{1,1}, \dots, o_{1,x}$ were emitted. Let the States of A during the computation be $A_0, A_{1,0}, A_{1,1}, \dots, A_{1,k}$ 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 have 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. To make the reasoning easier lets first assume, that the next input event is only consumed, when all outputs based on the previous event were produced (therefore being kind of synchronous). 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: $A_0, A_{1,0}, A_{1,1}, \dots, A_{1,j1}, A_{2,0}, \dots, A_{x,j2}$. 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 $\vec{I}$.

Problem for multiple input events: an output event from I_2 could be produced by A before all output events of I_1 were produced

List of Figures

List of Tables

Colophon

This thesis was typeset with $\LaTeX\,2_\varepsilon$. 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