Contents

1 Concepts

In this Chapter different ways to evaluate TeSSLa specifications are given and their equivalence is shown. To do so in Section 1.1 building blocks for evaluation approaches are defined, which are then used in later Sections to define behaviour of them and show their equivalence.

1.1 Definitions

While the TeSSLa specification itself defines a set of semantics, for this Thesis we will slightly alter some of it and add some new definitions based on them. This is necessary to reason about the specifics how the evaluation engine is built (Note that TeSSLa doesn't define an operational semantic, therefore we will define our own) and how it behaves.

1.1.1 Events

Events are the atomic unit of information that all computations are based on. 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 stream it's perceived on (e.g. a function call of a specific function or the name of an output stream).

The value of an event can be queried with the function υ, its timestamp with π and its Stream with ς.

$E_i \subset E$ is the Set of all input events, their Stream corresponds to a specific trace. $E_o \subset E$ is the Set of all output events, their Stream 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 Stream 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 $E_i \cup E_o \cup E_n = E$.

1.1.2 Streams

Stream are a collection of Events with some added characteristics. While Events are the atomic unit of information, Streams represent the sequence of related Events over time.

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 can be represented as a Sequence of Events $\langle e_1, \dots e_n \rangle$ where $\pi(e_i) < \pi(e_{i+1}) \forall i < n \in \mathbb{N}$. 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 only of 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. To get the Event of a Stream σ at a timestamp t it can be queried like a function: $\sigma(t) = e$ with $\pi(e) = t$. When working with Signals, the function will return the latest Event that happened at or before t while an EventStream may return $\perp$.

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.

1.1.3 Functions

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

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 over the whole input stream, meaning they can look at all Events that happened on its inputs before the time of evaluation and also at all it's previous output events. 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 1.2. In Section **??** the conclusions of earlier sections will be extended to include timing functions.

1.1.4 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 1.1.4)
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.

1.1.5 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 Engine 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.

Definition 1. *Two evaluation Engines are called equal, if they produce the same Relationship between inputs and outputs.*

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.

1.1.6 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 which holds the History of its input Streams and its output Stream. 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.

To distinguish between the two types of States, we will call the State of the whole Engine the *global State* and the State of a Node the *Node State*.

Formally the State of a Node is a tuple of the inputs (a Set of internal and input Streams) and the output (an internal or an output Stream). The Set of all valid States of a Node is $N = (\Sigma_i \cup \Sigma_n)^* \times (\Sigma_n \cup \Sigma_o)$. Based on this, the Set of all valid States is $S = N^*$. A global State can be queried like $S_1(i) = n_i$ to yield the State of the Node at that position.

Everytime a Node is scheduled it changes it State by:

- Updating it's inputs (Adding a new Event to the Input Streams)
- Generating a new Event if possible and add it to it's outputs

1.1.7 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 optional generation of one or more new output Events of that Node. To look at it in another way: A Transition is the computation of a Node, therefore when we say 'Node A is scheduled' we mean that a Transition is taken which models the computation of that Node.

Formally Transitions are a Relation between two Sets of Events. E.g. the Transition $\tau = (\{e_1, e_2\}, \{e_3\})$ specifies that two Events were consumed by a Node (thereby adding them to its inputs) and one Event was produced based on them (which is added to the output). The Set of all Transitions is

$$T = \{(e, e') | e \subseteq (E_i \cup E_n), e \neq \emptyset, e' \subseteq (E_n \cup E_o), \forall e_i, e_j \in e' : \varsigma(e_i) = \varsigma(e_j)\}$$

where the Node which caused the Transition is the one creating the Stream $\varsigma(e_i)$.

The empty Transition, meaning no input was consumed and no output produced, is labeled with λ. Note that all Transitions, except the empty one, have to consume at least one Event (therefore no Events can be created from nowhere) and that it's possible that no Event was produced based on the consumed Events (think of a *FilterNode*). Furthermore it's theoretically possible to create multiple Events in one Transition. This makes only sense in the context of Timing Nodes, because else the generated Events would have the same timestamp, which is forbidden by the definition of Streams. With Timing Nodes one could for example implement an *EchoNode*, which duplicates an input after a specified amount of Time.

1.1.8 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 input 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.

Definition 2. *Two runs are called equal if they have the same State at their last position.*

Because the State can be built from the Transitions that were taken, equality can also be defined over Transitions.

Lemma 1. *If the Set of Transitions of two Runs are equal, the Runs are equal.*

Proof. Let R_1, R_2 be the Runs of two Engines with the same Set of Transitions T'. Let S_1 be the final State of the Run R_1 and S_2 of R_2. If the two global States weren't equal, there would have to be at least one i with $S_1(i) \neq S_2(i)$, meaning the same Node has to have a different State in both Engines. Let n_1, n_2 be the States of one such Node in both Engines. If the two States are different, one of them has to contain at least one Event on an input or output that isn't on the same Stream in the other State. Let that Event be e_d. To be added to the State, there has to be a Transition $\tau = (e, e')$ with $e_d \in e \vee e_d \in e'$. This Transition has to be in T', which means it was taken by both Engines, therefore e_d is in the History of the Node in both Runs. □

1.2 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.
- An internal Node, which has at least one new input buffered on all of its input queues, can perform its 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 Evaluation Engines are classified by their scheduling approaches.

1.2.1 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.

Definition 3. *An evaluation Engine is called* valid *for a specification, if it is equal to a synchronous evaluation Engine for that specification.*

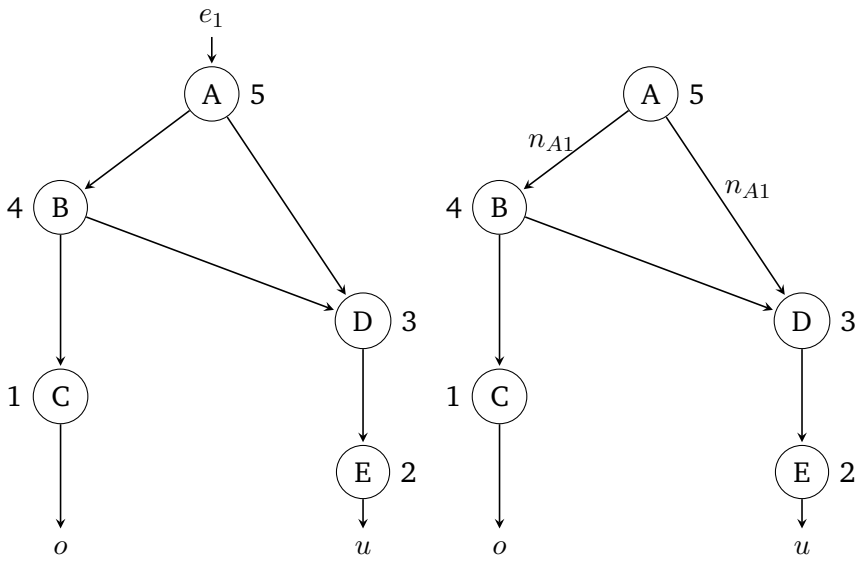

Fig. 1.1: Visualization of a simple asynchronous system with a reversed topological order.

As shown in Section 1.3 all synchronous evaluation Engines are equal. Therefore to proof that an evaluation Engine is valid for a specification, one has to show, that it is equal to any synchronous one.

Figure 1.1 visualizes a synchronous evaluation Engine. 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 Streams 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.

1.2.2 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.

more later

1.3 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 the same evaluation Engine for a TeSSLa specification.

Lemma 2. *Two evaluation Engines are equal, if their 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 1.2.1 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 1.3.1it is shown, that all possible synchronous Engines for a specification are equal, so there is only one true Evaluation. Afterwards in Section 1.3.2 it is shown that any asynchronous evaluation Engine is equal to a synchronous one.

1.3.1 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 1.2.1.

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 Engines receive. Furthermore let R_1, R_2 be the Runs of the two Engines 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 is 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 the two Engines scheduled different Nodes.

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.

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.

The praragraph is a bit handwavy, needs more formal definitions

Figure 1.2 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 Streams it doesn't matter. Therefore it doesn't matter which of the

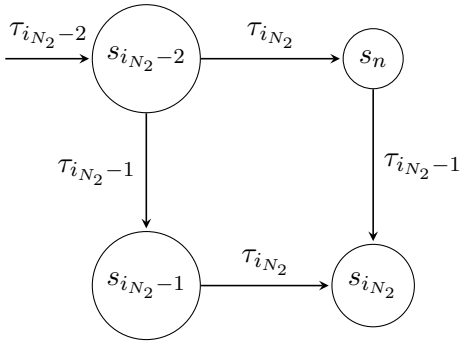

Fig. 1.2: Commutativity Diagramm of Node scheduling

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.

To summarize: The order of computation of the Node N_2 and the one scheduled before it doesn't change the State of the Evaluation Engine after both have run. Therefore their order can be changed, which generates a new Run which is equal to the old one.

This reordering can be repeated until N_2 is scheduled right after N_1, because as shown earlier N_2 has to be independent of all Nodes between the two. Now N_1 and N_2 also have to be independent, else one of them couldn't have been scheduled before the other. Therefore the two can be also reordered without changing the final State. After this reordering, both Runs will have a final prefix R'', which length has to be at least one more than R'.

Now the reordering can be repeated for the next index where the first difference between the runs happen. Everytime the reordering for one index finishes the finite, equal prefix will has at least one more position, therefore at one Point both runs are equal, because the Prefix is the run itself.

1.3.2 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.

1.4 Behaviour with Timing functions

1.5 Equalitys with Timing functions

1.6 Parallel computation

List of Figures

List of Tables