Contents

Concepts 1

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 Time

TeSSLa has a model of continuous Time, where timestamps $t \in \mathbb{T}$ are used to represent a certain point in Time and $\mathbb{T}$ has to be isomorphic to $\mathbb{R}$.

1.1.2 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 $E_i \cup E_o \cup E_n = E$.

1.1.3 Streams

Streams are a collection of Events with specific 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 Event-streams, 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 denotes 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 $\bot$ when queried for its value at a specific time, which denotes, that no event happened at that time. Based on 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 $\langle e_1, \ldots e_n \rangle$ 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 $\bot$.

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. The progress of a Stream can be obtained with $p(\sigma) = t \in T$.

Internal and output Streams can be queried for the Node that produced them with $\eta(\sigma) = n \in N$.

1.1.4 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 consumed by 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 evaluated 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.

1.1.5 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. The set of all Nodes is called N. The Function of a Node $n \in N$ is written as f_n.

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 1.1.5)

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.

1.1.6 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 (Equivalence of evaluation Engines). *Two evaluation Engines are called equivalent, if they produce the same relationship between inputs and outputs for all possible inputs.*

Definition 2 (Equivalence of evaluation Engines for an input). *Two evaluation Engines are called equivalent for an input, if they produce the same output for that input.*

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.

1.1.7 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 Set of a Sequence of the inputs (which are internal or input Streams) and the output (an internal or an output Stream). The Set of all valid Node States $\widetilde{N} = (\Sigma_i \cup \Sigma_n)^* \times (\Sigma_n \cup \Sigma_o)$. Based on this, the Set of all valid global States is $S = \widetilde{N}^*$. A global State can be queried like $S_1(n) = \widetilde{n}_i$ to yield the State of the Node n and also by its index $S_1(i),\ i \in \mathbb{N}$ to get the Node State at that index.

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 output

1.1.8 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

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

where the Node which caused the Transition is $\eta(\varsigma(e_i))$. For brevity we define $\eta : T \to N$ with $\eta((e, e') = \eta(\varsigma(e_i)), \ e_i \in e'$, which yields the Node that is modeled by the Transition.

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.

Definition 3 (Application of a Transition on a State). *Given a global State S_0 and a Transition $\tau_1 = (e, e') = (\{e_1, e_2, \ldots, e_i\}, \{e'_1, e'_2, \ldots, e'_i\})$, when we* apply τ_1 to S_0 *we get a new global State S_1 with*
$\forall \widetilde{n}_i = S_0(i)$

- *if $\eta(\widetilde{n}_i) \neq \eta(\tau_1) \to S_1(i) = \widetilde{n}_i$ (No Node States changes except the one from the Node of the Transition)*

- *else let $(\{\sigma_1, \ldots, \sigma_n\}, \sigma_o) = n_i$. Than is $S_1(i) = (\widetilde{\sigma}, \sigma'_o)$ with*

    - *$\sigma'_o = \sigma_o + +e'$ with $++$ defined as appending all Events in e' to σ_o ordered by their timestamp*

    - *$\widetilde{\sigma} = \{\sigma'_1, \sigma'_2, \ldots, \sigma'_n\}$ with $\sigma'_l, l \in [1, n] : \sigma_l + \widetilde{e}$ where $\widetilde{e} \in e \wedge \eta(\varsigma(\widetilde{e})) = \eta(\sigma_l)$ and $+$ is the known append operator for sequences.*

This means, that the new global State is built with the old global State by altering only the Node State of the Node identified by the Events in the second element of the Transition. The Node State is altered by appending all Events in the second Element of the Transition to the output ordered by their timestamp and by adding

each Event in the first element of the Transition to the input corresponding to the Stream of that Event.

1.1.9 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 length of a Run can be retrieved with $l(R_1) = d \in \mathbb{N}$. A Run can be queried by it's index to return the element at that index: $R_1(i) = (\tau_i, S_i), \ i \in [0, l(R_1)]$.

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 4 (Equivalence of Runs). *Two Runs are called equivalent if they have an equal State at their last position.*

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

Lemma 1 (Equivalence of Runs over Transitions). *If the Set of Transitions of two Runs are equal, the Runs are equivalent.*

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 $\widetilde{n}_1, \widetilde{n}_2$ be the Node States of one such Node in both Engines. If the two Node 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, therefore its the two Node States are equal in both Engines. $\square$

Definition 5 (Distance of Runs). *The Distance δ of a Run R_1 to a Run R_2 is a pair $\delta(R_1, R_2) = (x, y)$, where x is the index before the first position where the two Runs differ and y is the number of Steps between the index of the first difference and the position where R_2 takes the Transition that R_1 took after step x. The Distance of Runs is ordered element-wise: $(x, y) < (x', y') \leftrightarrow ((x > x') \vee (x = x' \wedge y < y))$. Therefore two Runs with length d are equal, if their Distance is $d, 0$, which is the minimal Distance two Runs of length d can have at all.*

Example 1. *Let*

$$R_1 = \langle (\lambda, S_0), (\tau_1, S_1), (\tau_2, S_2), (\tau_3, S_3), (\tau_4, S_4), (\tau_5, S_5), (\tau_6, S_6) \rangle$$

$$R_2 = \langle (\lambda, S_0), (\tau_1, S_1), (\tau_2, S_2), (\tau_5, S_3'), (\tau_4, S_4'), (\tau_6, S_5'), (\tau_3, S_6') \rangle$$

$$R_3 = \langle (\lambda, S_0), (\tau_1, S_1), (\tau_2, S_2), (\tau_3, S_3), (\tau_5, S_4''), (\tau_4, S_5''), (\tau_6, S_6'') \rangle$$

Than is

$$\delta_{1,2} = \delta(R_1, R_2) = (3, 3), \; \delta_{1,3} = \delta(R_1, R_3) = (4, 1)$$

$$\delta_{2,1} = \delta(R_2, R_1) = (3, 2), \; \delta_{2,3} = \delta(R_2, R_3) = (3, 1)$$

$$\delta_{3,1} = \delta(R_3, R_1) = (4, 1), \; \delta_{3,2} = \delta(R_3, R_2) = (3, 2)$$

And $\delta_{1,3} < \delta_{2,3} < \delta_{2,1} < \delta_{2,1}$.

Definition 6 (Enabledness of a Node). *A Node with the input Streams $\tilde{\sigma}$ in an evaluation Engine is called* enabled *at a Step s of a Run R_1 of that Engine, when it satisfies all of the following conditions:*

- *All consumed Events are either internal Events that were produced at an earlier step or external Events that are present:*

    - *$\forall \sigma_x \in \tilde{\sigma} \cap \Sigma_n : \exists j \in [1, s] : R_1(j) = (\tau_j, S_j)$ with*

        * *$(e, e') = \tau_j \wedge e_x \in e'$ with $\varsigma(e_x) = \sigma_x$*

    - *and $\forall \sigma_i \in \tilde{\sigma} \cap \Sigma_i : \exists e_i$ with $\varsigma(e_i) = \sigma_i$*

- *and weren't consumed earlier*

    - *$\forall e_x$ (the same e_x as in the steps above)*

        * *$\nexists h \in [j, s] : (\tilde{e}, \tilde{e'}) = \tau_h \wedge (\tau_h, S_h) = R_1(h) \wedge e_x \in \tilde{e}$*

Definition 7 (Valid Run). *A Run R_1 is called* valid *if*

- *$\forall i \in [0, l(R_1)] : R_1(i) = (\tau_i, S_i) \wedge \eta(\tau_i)$ is enabled at step i*

- *and S_i is built by applying τ_i to S_{i-1}*

Definition 8 (Independence of Nodes). *A Node A is called* independent *of Node B in an evaluation Engine, if it is no descendant of that Node*

Definition 9 (Independence of Transitions). *A Transition τ_1 is called independent of another Transitions τ_2, if $\eta(\tau_1))$ is independent of $\eta(\tau_2))$.*

Lemma 2 (Exchange of independent Transitions)**.** *If a Transition τ_1 is independent of a Transition τ_2, than for all Runs of the evaluation Engine that produces the Runs the following holds:*

$$R_1 = \langle (\lambda, S_0), \ldots, (\tau_2, s_i), (\tau_1, s_j), \ldots, (\tau_l, s_l) \rangle \textit{is a valid Run}$$
$$\rightarrow R_2 = \langle (\lambda, S_0), \ldots, (\tau_1, s_i'), (\tau_2, s_j'), \ldots, (\tau_l, s_l') \rangle \textit{is a valid Run}$$

Proof. Because $A = \eta(\tau_1)$ is no descendant of $B = \eta(\tau_2)$, the Stream σ with $\eta(\sigma) = \eta(\tau_2)$ can be no input of A. Therefore the enabledness of A can't be changed by τ_2 as you can see from the Definition of enabledness. So A has to be enabled before τ_2 was taken by R_1, else it couldn't be enabled afterwards. Therefore R_2 also fullfills the requirements for a valid Run. $\qquad\square$

Lemma 3 (Influence of independent Nodes)**.** *When a Node A is independent of a Node B, than it has no influence on the enabledness of Node A if Node B is scheduled before or after it.*

Proof. Follows directly from Lemma 2 $\qquad\square$

Lemma 4 (Duration of Enabledness)**.** *A Node which is enabled stays enabled at least until it is scheduled. Formaly: If a Node n is enabled at step i in a Run R_1 it will stay enabled at least until Step $j > j$ with $R_1(j) = (\tau, s), \eta(\tau) = n$. Note that it doesn't have to be enabled after step j, because there could have been multiple Events buffered on it's inputs.*

Proof. TODO $\qquad\square$

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 Functions. Without timing Functions, all Nodes work only on values or the presence of Events. This leads to behaviours that can be easily reason about, as seen in the next sections.

All Systems to evaluate TeSSLa specifications we will look at are based on the described structure in Section 1.1.6. While there may be other approaches to evaluation, a DAG based approach seems to fit most naturaly and focusing on one structure makes comparing Systems easier.

Each evaluation Engine will work in steps, where each step is synonymous with an index in the Run of the System. Therefore at each Step one Node is scheduled to perform it's operation, represented as the Transition in the Run. The Transition will encode one of the following three Things that can happen:

- 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 all of its input queues, can produce a new output.

Evaluation Engines are free in the way they are scheduling their Nodes, only limited by causality (no Event can be consumed before it's produced). In the following evaluation Engines are classified by their scheduling approaches.

1.2.1 Synchronous Evaluation Engines

The first class of evaluation Engines are synchronous ones. They are characterized by a specific, fixed schedule. The scheduling algorithm is as follows: Choose a reversed topological order of the Nodes once and assign numbers to them in that order. Then always schedule the enabled Node with lowest Number.

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 10 (Equality of evaluation Engines). *An evaluation Engine is called* valid, *if it is equal to a synchronous evaluation Engine.*

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 present and can be consumed by the input Node A. When a Node is chosen to compute by the scheduler, only Node A is enabled, therefore it is scheduled. The right system is the representation of the next step: Node A has consumed the external event and produced
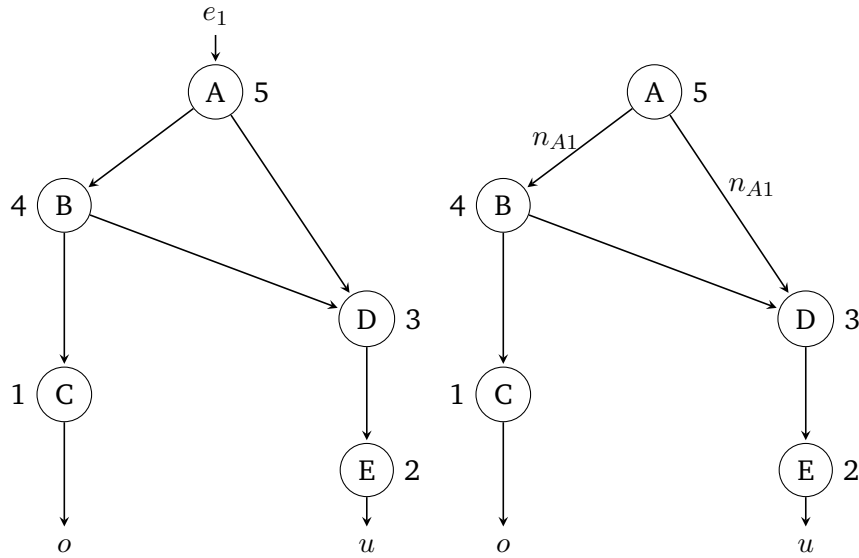
Fig. 1.1: Visualization of a simple asynchronous system with a reversed topological order.

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:

$$\langle (\lambda, S_0), ((\{n_{A1}\}, \{n_{B1}\}), S_1), ((\{n_{B1}\}, \{o_1\}), S_2),$$
$$((\{n_{A1}, n_{B1}\}, \{n_{D1}\}), S_3), ((\{n_{D1}\}, \{u_1\}), S_4)\rangle$$

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 enabled 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 5 (Equality of Engines for one Input). *Two evaluation Engines are equal for an input, if their Runs are equal for that input.*

Proof. Since two Runs are equal if they have the same last State, and all Events which were produced are stored in the State, during both Runs the same output Events had to be generated or else their State would differ. □

As defined by Definition 10 any evaluation Engine has to be equal to a synchronous to be valid.

The Equality is shown in two steps: First in Section 1.3.1 it is shown, that all possible synchronous Engines for a specification are equal, so there is only one valid 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 for a specification 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. Also both Runs will obviously have the same length (both Engines are the same DAG, so they have the same number of Nodes), let that length be l.

To proof the equality of both Engines we have to proof the equality of their Runs. To show the equality we will show that there is always another Run, which is equal to the second one, that has a smaller Distance from the first one. If such a closer Run always exist, we will show that the Run with Distance $l, 0$ to the first Run, which has to be the first Run itself, is also an equal Run to the second Run.

Theorem 1 (Equality of different synchronous evaluation Engines). *Two synchronous evaluation Engines for a specification with different schedules are equal.*

Proof. 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.

If the two Runs aren't equal, they must have a Distance which is bigger than $(l, 0)$. Let $[R_2]$ be the Set of all Runs that are equal to R_2. All of those Runs will also have a Distance from R_1 which is bigger than $(l, 0)$. Select one Run $R_2' \in [R_2]$ which has a minimal Distance from R_1. Let $(d, k) = \delta(R_1, R_2')$.

This means that at Step d the Run R_2' has taken a different Transition than Run R_1. Let the Transitions the Runs have taken be τ_1 for R_1 and τ_2 for R_2'. Run R_2' will take Transition τ_1 at step $d + k$ (as per the definition of the Distance). Obviously the two Transitions have to be independent of each other, else they couldn't have been taken in different order by the two Runs.

If $k > 1$ there will be a Transition $\tau_2' \neq \tau_1$ which is taken by the Run R_2' at Step $d + (k - 1)$. While this Transition τ_2' must also be taken in the first Run as per Lemma 4, it's not possible, that it was taken before τ_1, beause than the two runs wouldn't have been the same upto the point where τ_1 was taken. Therefore τ_1 has to be independent of τ_2', and because τ_2' was scheduled by the second Run before τ_1 both Transitions are independent of each other.

As of Lemma 3 which one of them is taken first can't change the enabledness of the Node of the second Transition. Therefore there is a Run R_2'' which is exactly the same to R_2', except that the Transitions τ_1, τ_2' are scheduled the other way around. Figure 1.2 visualizes how changing the order of the two Transitions can't change the global State of the Engine after both were taken. Therefore the Runs R_2' and R_2'' are equal and the Distance to R_1 is $d, k - 1$, which contradicts the initial statement that R_1' was a Run with a minimal Distance. This means that there is an equal Run to R_2 which has at most the Distance $(d, 1)$.

If $k = 1$ the Transition τ_2' from the previous case is equal to τ_2. Based on the same reasoning there also exist a Run R_2'' which is exactly the same to R_2' except that the order of τ_2 and τ_1 is changed, and which is also equal to R_2' and to R_2. This Run has the Distance $(d, 0)$ from R_1. This obviously doesn't make sense: The first element of the Distance is the last step where both Runs are equal, the second element describes how many Steps afterwards the differing Transition was taken. But if it was taken right in the Step after the last equal Step, there is no difference at that position, so the Difference of R_1 and R_2'' can be at most $(d + 1, x), x \in \mathbb{N}_{>0}$. This also contradicts our initial Statement that R_2' was the Run with the smallest Distance from R_1 which is equal to R_2.

Combined we can now say, that there is no lower bound on the Distance of equal Runs of R_2 to R_1, therefore the Run with the Distance $(l, 0)$ also has to be equal to R_2.
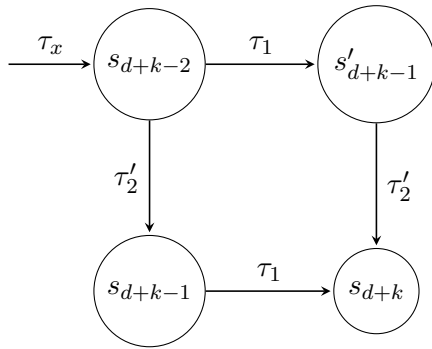
Fig. 1.2: Commutativity Diagramm of Node scheduling

□

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

List of Theorems