UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

Synthesis of Stream-based Monitors on FPGAs

Synthese von strombasierten Monitoren auf FPGAs

Masterarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Thiemo Bucciarelli

ausgegeben und betreut von
Prof. Dr. Martin Leucker

mit Unterstützung von
Malte Schmitz
Daniel Thoma

Lübeck, den 1. März 2020

Erklärung

Ich erkläre hiermit an Eides statt, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

(Thiemo Bucciarelli)
Lübeck, den 1. März 2020

Abstract In this thesis, a method to synthesise stream-based monitors on FPGAs is presented. For this, a translation from specifications written in the language *TeSSLa* to hardware descriptions is defined, which can then be synthesised by use of CAD tools on an FPGA. Additionally, there are multiple optimisation phases motivated and defined, with the purpose of improving the performance or resource consumption on the hardware. This is implemented as a Scala program which takes a *TeSSLa* specification and produces Verilog code which represents this specification in hardware. This implementation is then used to evaluate the performance and resource usage of the introduced concept and optimisation phases.

Kurzfassung In dieser Arbeit wird eine Methode zur Synthese von Strom-basierten Monitoren auf FPGAs eingeführt und implementiert. Hierfür wird eine Übersetzung von Spezifikationen in der Sprache *TeSSLa* zu Hardwarebeschreibungen definiert, welche dann mittels CAD tools auf FPGAs synthetisiert werden kann. Zusätzlich dazu werden mehrere Optimierungsphasen motiviert und definiert, mit dem Ziel die Performanz oder Ressourcenverbrauch auf der Hardware zu optimieren. Dies ist implementiert als ein Scala Programm, welches eine *TeSSLa* Spezifikation einliest und Verilog Code generiert welcher diese Spezifikation in Hardware darstellt. Diese Implementierung wird dann verwendet um die Performanz und den Ressourcenverbrauch des eingeführten Konzepts und der Optimierungsphasen zu evaluieren.

Contents

1. Introduction

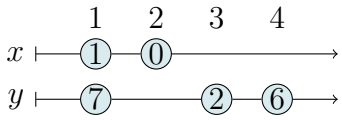
Nowadays, electronic systems can be found in many areas around us. Often, like in avionics or railway applications, malfunctions can have severe outcomes and may even endanger human lives. Hence a system — depending on the risk it may pose to its environment — requires a certain level of safety to assure correct behaviour in critical situations. One crucial aspect of this is the software verification whose objective is to ensure the specified functionality of the system. The common verification tools can be split into two categories:

Static analysis focuses on the use of formal methods to analyse the system and assure correctness of it (or parts of it). Prominent tools for this are Hoare logic or Model Checking. With increasing size and complexity of systems — for example concurrency or distributed systems — formal methods increase drastically in their required workload, often making them unviable to prove correctness of the whole system.

Dynamic analysis extracts its information from runs of the system. This allows for a better scalability, at the expense of it not being an actual proof of correctness, as it is usually not possible to cover all possibly existing runs. One category of this is runtime verification. The basic idea of runtime verification is to specifically evaluate a single run of the system in question, during runtime. This is usually accomplished by describing the required properties by using a logical formula, which can then be translated into a monitor. Since runtime verification takes place during the execution of the system, it can also be used to react to malfunctions accordingly and attempt to lead the system back to a valid state, which is called recovery.

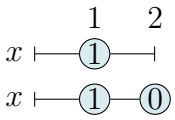
One possibility to define such monitors is by the use of specification languages, which allow to describe a system on a high level and thus allow to directly define the monitor itself.

TeSSLa (*Te*mporal *Stream*-based *S*pecification *L*anguage) [CHL+18] is a specification language using stream-processing. Here, a stream is a timed sequence of discrete events, where each event has a timestamp and a value, as shown in the following example, showing two streams x which has events at timestamps 1 and 2, and y with events at timestamps 1, 3 and 4:



Events on a stream are shown as circles with their associated value, the timestamp is written above. Note that the timestamps are not limited to be natural numbers, they may also be from a continuous domain like $\mathbb{R}$. Here and in following examples however, natural number timestamps are used for the sake of simplicity. As you can see here, events on streams are not limited to simple boolean values but can be of more complex data domains, thus also allowing not only to represent logical assertions but also more complex computations.

Streams denote a timed sequence of events, which, in case of an online analysis, may not be yet known in their entirety at a specific point in time. Taking the previous example, the stream x might only be known up to timestamp 2, meaning it is not yet known if the stream has a value on timestamp 3 or not. This is called the progress of a stream. Additionally, this progress can be inclusive or exclusive, which in our example denotes the difference whether or not it is known for timestamp 2 itself if there is a value or not. The following figure shows two versions of x, the first one having an exclusive progress of 2, the second one with an inclusive progress of 2, where the bar on their right side marks the progress of the stream:



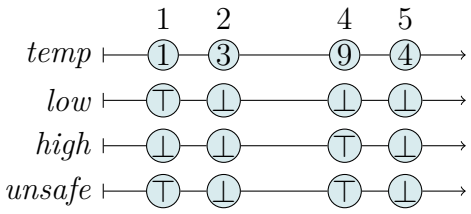
As such a stream is also required to be totally ordered in their timestamps, the progress for each stream is effectively denoted by the most recent timestamp of that stream.

Each definition in *TeSSLa* is effectively a transformation from such a stream to another stream. In the following, you can see an example[1] for a *TeSSLa* specification:

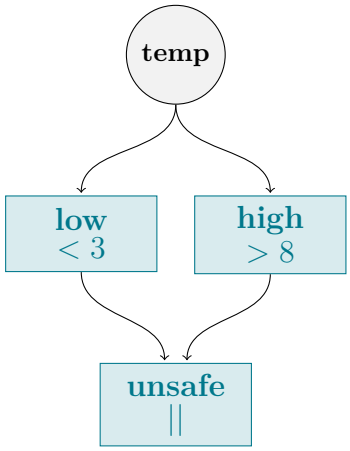
```
in temp: Events[Int]

def low := temp < 3
def high := temp > 8
def unsafe := low || high

out *
```

[1] taken from: `https://www.tessla.io/`

Here, an input stream *temp* is expected, which is then checked to be in a safe range between 3 and 8. An evaluation of this specification could then look as follows:



The evaluation of those streams is event-based, which means that the occurrence of a new event on stream *temp* triggers the computation of a new event for its dependent streams. Furthermore, the already existing events can not be affected by that newly occurring event. One further important aspect to note is that such a stream-based approach does not define any sequential execution order, but instead models the data-flow between operations. Such a data-flow oriented language is inherently parallel, as can be seen in the data-flow diagram of the previous example:



Here you can see that the streams *low* and *high* are not dependent of each other and can therefore be computed in parallel. Note that the parallelism described here does not refer to data parallelism, but instruction level parallelism, meaning that multiple different operations can be executed in parallel on the same data. This property of *TeSSLa* allows to make use of parallelism significantly easier compared to imperative languages. As previously described, each stream is a timed sequence, containing their timestamps and progress explicitly. Therefore *TeSSLa* does not require synchronisation of streams on a global level, but only locally. This allows for asynchronous streams which are also independent in their data-flow to be computed in parallel even when progressing at different frequencies, thus *TeSSLa* is also viable for asynchronous or decentralised systems. [LSS+18, LSS+19a]

While a purely software based solution can make use of the parallel properties of *TeSSLa*, it can only do so on a high level, as for example by using thread- or task-level parallelism by using multi-core or multi-processor systems. However, this requires the use of features like scheduling of threads and communication or synchronisation between tasks, entailing a significant amount of overhead, which may very well exceed the gain from executing it in parallel. Especially when considering a very fine-grained parallelism such an approach would most likely not yield any satisfying results. Additionally, use cases like performing online analysis on debug data of processors as discussed in [DDG+18], require a high-performance solution to be able to process events with the required frequency.

A hardware solution would allow for an optimal use of *TeSSLa*s inherent parallelism. Furthermore, the previously described feature of *TeSSLa* not using global synchronisation also allows to use pipelining. Referring to the previous example again, this means that while *unsafe* is processing an event of a certain timestamp, *low* and *high* can already process the following event. Another advantage of a hardware-based approach is that it would also prove beneficial to use cases where real-time constraints apply, as ensuring such constraints is significantly more difficult in a software solution, since its execution time is a lot more fluctuant. This is due to the fact that the execution time of a software solution is affected by many factors, like scheduling or caching.

As described in [CHL+18], as long as *TeSSLa* operates only on streams with bounded, constant-sized values, each operator only needs a constant amount of memory as it does not require to store the entire stream, but only needs to store one single data value at most. This is important for an implementation in hardware, as this permits to implement the operators with constant space usage and thus only using registers, without having to rely on random access memories. This allows for a more performant implementation of *TeSSLa* specifications on hardware compared to using random access memory, as accessing random access memory takes significantly longer than accessing a register.

However, compared to a software-based solution, an implementation in hardware comes at the expense of flexibility. A popular compromise for this is the use of FPGAs, which allow to define hardware in a reconfigurable way, such that some flexibility is being retained.

The goal of this thesis is to describe an automated synthesis of arbitrary *TeSSLa* specifications to FPGAs. Specifically, this is accomplished by defining *TeSSLa* operators as hardware modules which are responsible for computing the according events. Furthermore, we introduce queues, which handle storing and dispatching of events. This effectively forms a separation of modules into functionality and storing modules. This separation of concerns is essential for later optimisations. Additionally, a communication interface and protocol for communicating between modules

is introduced. Afterwards, through analysing properties and inferring information on *TeSSLa* specifications, multiple different optimisation phases are defined, with aiming to improve the performance and area used on the hardware. In particular:

- Some streams can be known to only have events at exactly the same timestamps. This allows to perform optimisations with the goal of reducing synchronisation overhead between those streams, as they are effectively already synchronised on their logical time.
- Depending on the used operators and potential additional metadata, it is possible to infer required and sufficient bit widths for values. This may allow to reduce the amount of resources required on the hardware.
- Definition of a hardware circuit always comes with timing constraints, as a signal can not travel arbitrary distances within a single clock cycle. As queues are used to store events in between stream operations, their placement has a direct impact on the path lengths and thus timing of signals. Additionally, if an event is led through multiple paths where one path requires more clock cycles than the others, this path would act as a bottleneck. Therefore, the deduction of a useful depth for queues is crucial as well.

The discussed translations and optimisations have been implemented in the context of this thesis. The implementation allows to automatically generate hardware descriptions of *TeSSLa* specifications for a specific hardware setup, which are then benchmarked in respect to their throughput and resource usage. Additionally, a second, slightly different setup was realised which allows an accurate measurement of the effect of the optimisation phases on the overall performance. Those measures are then used to evaluate the concept and to conclude future work.

1.1. Related Work

In [DDG+18] and [DGH+17], a online analysis for trace data in embedded systems is described which defines a configurable, low-level interpreter for *TeSSLa* specifications on FPGAs. This approach allows reconfiguring the hardware for different specifications without repeating the synthesis process. The approach taken in this thesis however aims to directly synthesise *TeSSLa* specifications on hardware, in order to maximise performance and optimally use parallel properties of *TeSSLa*.

There have been approaches to synthesise monitors from logics like STL (*Signal Temporal Logic*) [MN04] onto FPGAs as well. [JBG+15][SJN+17] However, those approaches are fundamentally different as they don't use the concept of event stream processing.

1.2. Outline

The thesis is divided into the following main chapters:

Chapter 2 formally introduces the specification language *TeSSLa* with its syntax and semantics. Furthermore, different approaches to designing hardware are analysed and the here taken approach is motivated. Additionally, FPGAs, high-level synthesis and HDLs are introduced.

Chapter 3 describes the general idea on how to represent a *TeSSLa* specification as hardware. This is accomplished by splitting the operators into separate modules which can then be combined into one main module describing the initial specification. A significant focus lays on the definition of the atomic modules, the introduction and usage of queues, and the description of the communication between modules.

Chapter 4 focuses on the optimisation of the concept by introducing three phases: *Stream merging*, *bit width inference* and *queue placement*.

Chapter 5 then highlights the implementation of the previously defined concept. This also includes an introduction of the used tools and languages, a visualised overview of the compiler pipeline, the test framework used and a full example on how the compiler can be used.

Chapter 6 evaluates the described implementation in respect to performance and area used, by introducing appropriate benchmarking setups and then evaluating different specifications for those aspects. A high focus here is on the evaluation of the optimisation phases, thus mostly comparing an optimised specification with its non-optimised counterpart.

2. Basics

In this chapter, the language *TeSSLa* is formally introduced with its syntax, semantics and properties relevant for this thesis. Specifically, recursive definitions in *TeSSLa* are being discussed as those are of high relevance for the further chapters of this thesis, for example assuring that there are no combinatorial loops occurring or when defining the different optimisation phases. Additionally, logic synthesis and high-level synthesis are being introduced and different approaches are being discussed.

2.1. TeSSLa

After shortly introducing *TeSSLa* and its concepts in the previous chapter, it will be formally introduced here with its syntax and semantics. Specifically, the subset of *TeSSLa* without the operator *delay* is used here. Note that the syntax used in the previous chapter differs from the one introduced here, as that was the syntax of the already existing *TeSSLa* front-end[1], which allows to define *TeSSLa* specifications with definitions of input and output streams, types, annotations and macros. This front-end will also be used in Chapter 5 and Chapter 6. Examples depicted in the following chapters use either of both syntax.

The following definitions are adapted from [CHL+18], and figures visualising streams are adapted from [ST20].

2.1.1. Syntax

A *TeSSLa* specification φ is defined in [CHL+18] as a set of definitions of the form $x := e$ with $x \in \mathbb{V}$ and $\mathbb{V}$ being a finite set of variable symbols, where

$$e ::= \textbf{nil} \mid \textbf{unit} \mid x \mid \textbf{lift}(f)(e,\ldots,e) \mid \textbf{time}(e) \mid \textbf{last}(e,e)$$

Additionally, the notion of *input variables* for every variable not occurring on the left-hand side and *flat* for a specification without nested expressions are adapted as well.

[1] `https://www.tessla.io/`

2.1.2. Semantics

The *TeSSLa* semantics are defined in [CHL+18] over an abstract time domain $\mathbb{T}$ as follows:

Definition 2.1 (Time Domain [CHL+18])**.** A time domain is a totally ordered semiring $(\mathbb{T}, 0, 1, +, \cdot, \leq)$ that is not negative, i.e. $\forall_{t \in \mathbb{T}} 0 \leq t$.

Additionally, such a time domain can then also be extended to $\mathbb{T}_\infty = \mathbb{T} \cup \{\infty\}$ with $\forall_{t \in \mathbb{T}} t < \infty$.

Definition 2.2 (Event stream [CHL+18])**.** An event stream is defined over a time domain $\mathbb{T}$ and a data domain $\mathbb{D}$ as a finite or infinite sequence

$$s = a_0 a_1 \cdots \in \mathcal{S}_{\mathbb{D}} = (\mathbb{T} \cdot \mathbb{D})^\omega \cup (\mathbb{T} \cdot \mathbb{D})^+ \cup (\mathbb{T} \cdot \mathbb{D})^* \cdot (\mathbb{T}_\infty \cup \mathbb{T} \cdot \{\bot\})$$

where $a_{2i} < a_{2(i+1)}$ for all i with $0 < 2(i+1) < |s|$.

Intuitively, an event stream therefore describes an alternating sequence of timestamps and values, where the timestamps are strictly monotonic. Therefore, this explicitly denotes up to which timestamp a stream is known, which is called the progress. This progress can be inclusive or exclusive, depending on whether or not the value for the current timestamp is known. As an example:

$$s_1 = \mathbf{1}\ 1\ \mathbf{4}\ \bot$$
$$s_2 = \mathbf{1}\ 1\ \mathbf{4}$$
$$s_3 = \mathbf{1}\ 1\ \boldsymbol{\infty}$$

Here, the timestamps are marked in bold for easier readability. All three streams shown here have a value 1 at timestamp 1. The stream s_1 has an inclusive progress of 4, as it is already known not to have a value at timestamp 4, contrary to s_2 which has an exclusive progress of 4. The stream s_3 has an infinite progress, denoting that it is fully known and does not have any values after timestamp 1.

Also, as described in [CHL+18], an event stream s can alternatively be seen as a function $s \in \mathbb{T} \to \mathbb{D} \cup \{\bot, ?\}$ where $s(t) = d$ if the stream s at time t has value d, $s(t) = \bot$ if it has no value at that timestamp, and $s(t) = ?$ if it is unknown.

Definition 2.3 (*TeSSLa* semantics [CHL+18])**.** For a specification φ of stream definitions $y_i := e_i$, every e_i can be interpreted as a function from input streams $s_1, \ldots, s_k \in \mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_k}$ and output streams $s'_1, \ldots, s'_n \in \mathcal{S}_{\mathbb{D}'_1} \times \cdots \times \mathcal{S}_{\mathbb{D}'_n}$ by composition of primitive functions, such that for fixed input streams $s_i, \ldots, s_k$

$$[\![e_i]\!]_{s_i,\ldots,s_k} \in \mathcal{S}_{\mathbb{D}'_1} \times \cdots \times \mathcal{S}_{\mathbb{D}'_n} \to \mathcal{S}_{\mathbb{D}'_i}$$

which can then be composed into a function over all definitions as follows:

$$[\![e_1, \ldots, e_n]\!]_{s_i,\ldots,s_k} \in \mathcal{S}_{\mathbb{D}'_1} \times \cdots \times \mathcal{S}_{\mathbb{D}'_n} \to \mathcal{S}_{\mathbb{D}'_1} \times \cdots \times \mathcal{S}_{\mathbb{D}'_n}$$

The semantics of the specification φ can then be denoted as the least fixed-point of this function:

$$[\![\varphi]\!] \in \mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_k} \to \mathcal{S}_{\mathbb{D}'_1} \times \cdots \times \mathcal{S}_{\mathbb{D}'_n}$$
$$[\![\varphi]\!](s_1, \ldots, s_k) = \mu([\![e_1, \ldots, e_n]\!]_{s_1,\ldots,s_k})$$

The computation of the fixed-point here is required as stream definitions in *TeSSLa* can also be recursive. This will be shown by use of an example after introducing the primitives. In the following, the primitive functions making up $[\![e_i]\!]_{s_i,\ldots,s_k}$ are listed as defined in [CHL+18]:

nil is a constant denoting the completely known stream without any events

$$[\![\mathbf{nil}]\!] = \infty \in \mathcal{S}_{\mathbb{D}}$$

unit is a constant for the stream with only a single event at timestamp zero

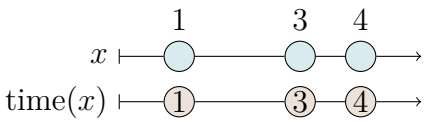
$$[\![\mathbf{unit}]\!] = 0\Box\infty \in \mathcal{S}_{\mathbb{U}}$$

where $\mathbb{U} = \{\Box\}$ is used as the unit type, which is used for streams with only a single value.

time returns the stream of timestamps of its encapsulating stream e. The semantics of the **time** operator is defined as $[\![\mathbf{time}(e)]\!] = \text{time}([\![e]\!])$ where time is a function $\in \mathcal{S}_{\mathbb{D}} \to \mathcal{S}_{\mathbb{T}}$ with $\text{time}(s) = s'$ such that

$$\forall_t s'(t) = t \Leftrightarrow s(t) \in \mathbb{D} \qquad \forall_t s'(t) = \bot \Leftrightarrow s(t) = \bot$$

The following example shows a stream x which has events at the timestamps $1, 3$ and 4 and the $\text{time}(x)$:



lift is used to lift an n-ary function f on values to streams. In the following, the notation $A_1 \times \cdots \times A_n \rightarrowtail B$ is used to describe the set of functions where all A_i and B have been extended by $\bot$.

The semantics of a binary **lift** is defined as $[\![\textbf{lift}(f)(e_1, e_2)]\!] = \text{lift}_2(f)([\![e_1]\!], [\![e_2]\!])$ with $\text{lift}_2 \in (\mathbb{D}_1 \times \mathbb{D}_2 \rightarrowtail \mathbb{D}') \rightarrow (\mathcal{S}_{\mathbb{D}_2} \times \mathcal{S}_{\mathbb{D}_2} \rightarrow \mathcal{S}_{\mathbb{D}'})$ with $\text{lift}_2(f)(s, s') = s''$ such that

$$\forall_{t,d \in \mathbb{D}'} s''(t) = d \Leftrightarrow (s(t) \in \mathbb{D}_1 \vee s'(t) \in \mathbb{D}_2) \wedge \text{known}(t) \wedge f(s(t), s'(t)) = d$$

$$\forall_t s''(t) = \bot \Leftrightarrow (s(t) = \bot \wedge s'(t) = \bot) \vee \text{known}(t) \wedge f(s(t), s'(t)) = \bot$$

where $\text{known}(t) := s(t) \neq ? \wedge s'(t) \neq ?$ denotes if the stream is known up to the given timestamp.

The definition of binary lift can then be extended to define n-ary lifts for $n > 2$ as described in [CHL+18].

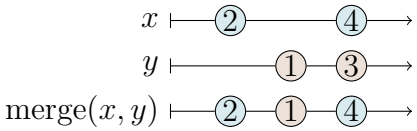
Also, a unary lift is defined as $[\![\textbf{lift}(f)(e)]\!] = \text{lift}_1(f)([\![e]\!])$ where $\text{lift}_1 \in (\mathbb{D} \rightarrowtail \mathbb{D}') \rightarrow (\mathcal{S}_{\mathbb{D}} \rightarrow \mathcal{S}_{\mathbb{D}'})$ and $\text{lift}_1(f)(s) = s'$ such that

$$\forall_{t,d \in \mathbb{D}'} s'(t) = d \Leftrightarrow s(t) \in \mathbb{D} \wedge f(s(t)) = d$$
$$\forall_t s'(t) = \bot \Leftrightarrow s(t) = \bot \vee s(t) \in \mathbb{D} \wedge f(s(t)) = \bot$$

An example usage for **lift** is shown in the following example. Here, an operator *merge*, which merges two streams into one, is defined as follows:

$$\text{merge}(x, y) = \text{lift}(\text{f})(x, y)$$
$$\text{f} : \mathbb{D}_\bot \times \mathbb{D}_\bot \rightarrow \mathbb{D}_\bot$$
$$\text{f}(a, b) = \begin{cases} b & \text{if } a = \bot \\ a & \text{else} \end{cases}$$

Applying this stream operation on two example streams x and y could then look like follows:



last takes two streams as input and returns the last known value (exclusively) of the first stream at the timestamps of the second. The semantics of **last** are defined as $[\![\textbf{last}\ (e_1, e_2)]\!] = \text{last}([\![e_1]\!], [\![e_2]\!])$ where $\text{last} \in \mathcal{S}_{\mathbb{D}} \times \mathcal{S}_{\mathbb{D}'} \rightarrow \mathcal{S}_{\mathbb{D}}$ and $\text{last}(s, s') = s''$ such that

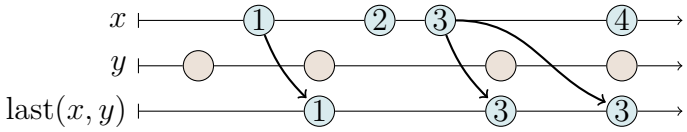
$$\forall_{t,d \in \mathbb{D}} s''(t) = d \Leftrightarrow s'(t) \in \mathbb{D}' \wedge \exists_{t' < t} s(t') = d \wedge \text{noData}(t', t)$$
$$\forall_t s''(t) = \bot \Leftrightarrow s'(t) = \bot \wedge \text{defined}(t) \vee \forall_{t' < t} s(t') = \bot$$

with

$$\text{noData}(t, t') := \forall_{t'' | t < t'' < t'} s(t'') = \bot$$

$$\text{defined}(t) := \forall_{t'<t} s''(t') \neq ?$$

The following example shows the result of **last** applied to the two streams x and y:



With **last** allowing to access previous values, and using **lift** to lift arbitrary functions on values to streams, this set of operators already provides a sufficient expressiveness for many use cases. As discussed in [CHL+18], *TeSSLa* without the *delay* operator as handled here, is able to describe every function $f \in \mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_k} \to \mathcal{S}_{\mathbb{D}'_1} \times \cdots \times \mathcal{S}_{\mathbb{D}'_n}$, as long that function is monotonic (order preserving), continuous (preserving the supremum), timestamp conservative (not introducing new timestamps) and future independent (outputs are only dependent on current and previous events).

2.1.3. Recursive Definitions

Streams in *TeSSLa* can also be recursively defined. However, some recursive definitions, like for example $x = x$ do not have a unique fixed point. Therefore, the notion of a well-formed specification was introduced in [CHL+18], declaring that a specification is well-formed if every cycle in the dependency graph contains at least one *delayed* labelled edge, and that edges to the first argument of a **last** operator are labelled with *delayed*. Furthermore, it is stated that the least fixed point of such a specification is also the only fixed point.

For the rest of this thesis, specifications used are always considered to be well-formed, which means that a recursive definition is only possible using the first argument of the **last** operator.

Considering the following example specification:

$$count = merge(last(count, x) + 1, 0)$$

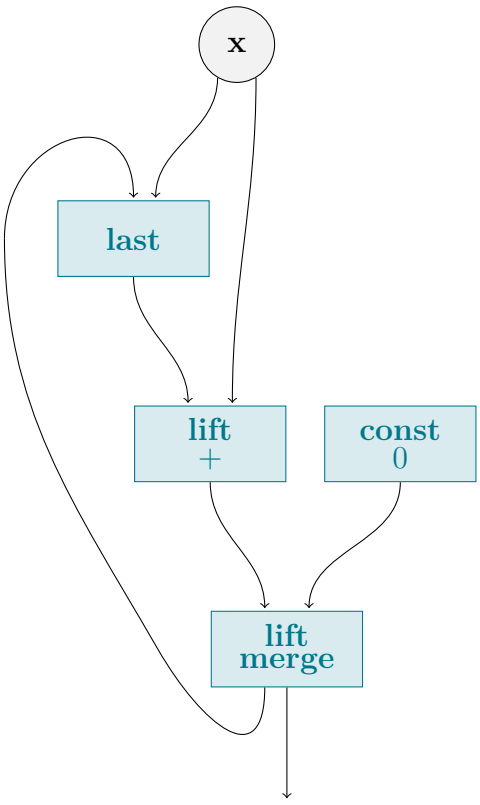
Here you can see that the stream *count* is defined recursively by referring to its own previous value. This value is then increased by one, and the use of the previously introduced *merge* allows to merge it with a constant 0. This stream therefore implements a counter, keeping count of the amount of events occurred on x. Note that, for the sake of simplicity, the 0 is here assumed to be lifted to a stream $\mathbf{0}\ 0\ \boldsymbol{\infty}$, which

can be obtained by use of **lift** with **unit**. Additionally, for this and future examples, arithmetic operations like + are interpreted as $\mathit{lift}(\mathit{total}(+))(x, y)$ where

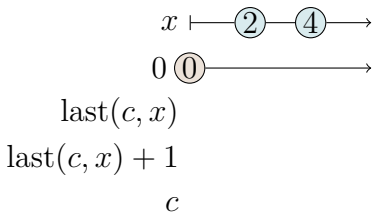
$$\mathit{total}(f)(a,b) = \begin{cases} f(a,b) & \text{if } a \neq \bot \wedge b \neq \bot \\ \bot & \mathit{else} \end{cases}$$

which applies the function f to the inputs only if both are defined, requiring both inputs to occur on the same timestamp. Another possibility would be to interpret such operators with signal semantics, by applying it to the last known values of each stream. This would not require the events of both streams to be at the exact same timestamp, which is a more natural interpretation of adding two streams. Such a *signal lift* operator can be implemented by using a combination of **lift** with **last**. However, this is omitted here as it is of no direct relevance for the concepts described in the following chapters. The implementation supports **lift** with signal semantics as well, as discussed in Section 5.3.

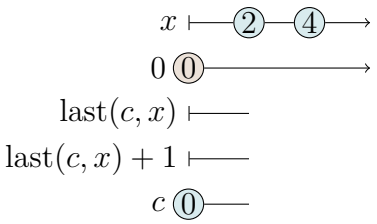
The dependency graph of this specification looks as follows:



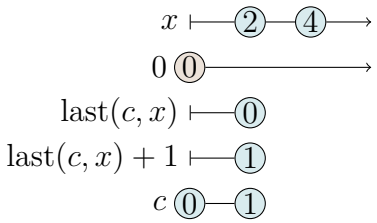
Such a recursive definition can be evaluated by sending the progress and values through the dependency graph until a fixed point is reached. The evaluation process



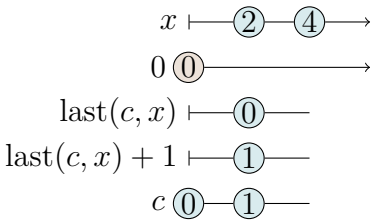
(a) Initial streams



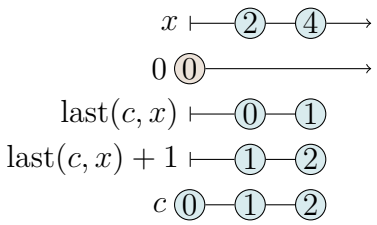
(b) Progression up to t_1



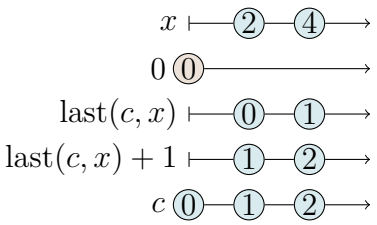
(c) Fixed-point



(d) Progression up to t_2.



(e) Fixed point



(f) Final result

Figure 2.1.: Example evaluation for *counter*.

is visualised in Figure 2.1 with the example of a stream $x = 1\ 2\ 3\ 4\ \infty$. The occurrence of an timestamp 1 on x allows $\mathbf{last}(c, x)$ to progress up to timestamp 1 (exclusive), since at that point it is known that no earlier events occur on x. Specifically, for timestamp 0, $\mathbf{last}(c, x)$ evaluates to $\bot$, allows the dependent streams to be computed as well, and then finally $count(0) = 0$. At this point, *count*, being the left-hand side input of **last**, only has a progress of 0, while x has a progress of 1. The semantics of **last**, in particular $x(t) = \bot \wedge \text{defined}(t) \Rightarrow \forall_t\ \mathbf{last}(c, x)(t) = \bot$, cause the stream to increase its progress by propagating it through the cycle up to 1. This is shown in 2.1a.

After the progress has been propagated through, the value for **last** at 1 can be computed, and with this also its dependent streams (2.1b). Similarly, the progress 3 at x then causes that progress to be propagated through the cycle again, and $c(3)$ can be evaluated.

2.2. Synthesis

As the goal of this thesis is to define a translation from *TeSSLa* specifications to FPGAs, a rough overview on FPGAs is given, followed by different approaches on how such a synthesis could be performed.

In hardware design, synthesis in general describes the process of transforming a higher level description of a system into an actual electronic circuit representing it. There are multiple different levels of abstraction, where *logic synthesis* and *high-level synthesis* are probably the most prominent.

An FPGA (*field-programmable gate array*) is an integrated circuit which consists of an array of reconfigurable logic blocks. Those configurable logic blocks (*CLB*) usually contain multiple look-up tables (*LUT*) which can be used to implement logic functions, and other components such as flip-flops. Those CLBs are then interconnected through routing channels, which are configurable through switch blocks at each intersection. This allows to define how data should be routed through the logic blocks. Additionally, there are specialised I/O blocks for external connections. FPGAs can be programmed by use of a bit-stream, which is usually automatically generated by a CAD tool, allowing description of the hardware on a higher level, which is then synthesised to a bitstream for the FPGA. However, those tools mostly only support a very specific set of languages, mainly *Verilog* and *VHDL*.

2.2.1. Logic Synthesis and HDLs

In logic synthesis, the input is a description of the hardware on register-transfer level (*RTL*), describing the circuit to be implemented. The logic synthesis then is the transformation of this circuit definition into a definition on gate-level, describing a net of logic gates implementing the described functionality. Languages used to describe hardware on this level are called hardware description languages (*HDL*), where well-known examples are *Verilog* and *VHDL*.

A significant difference between HDLs and programming languages are that HDLs contain a notion of time. The behaviour of a system is thus defined over time, which is essential for defining a hardware circuit. Also, since inherent concurrency is one of the main features of hardware, HDLs are usually data-flow instead of control-flow oriented and provide tooling specifically designed to support concurrent implementations. The design is usually split into modules with defined input and output ports and some attached behaviour.

2.2.2. High-Level Synthesis

High-level synthesis, or also called behavioural synthesis aims to design the system in a more abstract way, focusing more on the architecture and interfaces than on implementation details. This is especially useful for architectural exploration on more complex designs, as it allows quick prototyping, simulation and validation of the design compared to using an HDL. One framework often used in this context is *SystemC*, which is a library for *C++*.

2.2.3. Chisel

Chisel [BVR+12] is a domain specific language integrated as a framework into the functional programming language Scala. Contrary to SystemC however, Chisel is not used to describe the system on a behavioural level, but instead directly describes the logic and wiring similar to VHDL or Verilog. The advantage of Chisel over other HDLs is that, as a framework for Scala, it allows definition of hardware as meta programming while having access to all language features of Scala as well, such as higher order functions or type parameters. This allows defining hardware in a flexible and easy manner, while still maintaining control over implementation details like in conventional HDLs. The Chisel code written in Scala can then be compiled into Verilog code, which is useful as Verilog is widely supported by synthesis tools. Therefore, Chisel can be seen as a framework to generate parametrised Verilog code within Scala through meta programming.

A significant drawback of defining an entire system using high-level synthesis is that the increased amount of abstraction also shrouds the automatically generated implementation, which may lead to unsatisfactory results as discussed in [htt]. As synthesising individual *TeSSLa* operators does not come with major architectural challenges and as it would be favourable to maintain control over implementation details, there is no incentive to use high-level synthesis for our purpose. Additionally, defining the resulting HDL code through meta programming would be useful, as the goal is to translate arbitrary specifications and thus many aspects of the translation requiring to be parametrised on a meta level. Therefore, Chisel has been chosen for the implementation in Chapter 5. Hence, for the following chapters, modules are being described in an HDL-like pseudo-code.

3. General structure

The *TeSSLa* dependency graph already provides a data-flow oriented view on the specification. Thus, an intuitive way to model a specification into hardware would be to define each stream operation as a module, and their dependencies as communication channels. This way, the data-flow diagram could nearly directly be translated into the module graph, by replacing every operator in the graph with the according module. However, there are four main tasks to take into account here:

- The communication between the submodules needs to be defined, such that the progression of the streams represented by each module can be properly defined.
- The atomic modules need to be defined.
- The output of one module may be read by multiple other modules. This needs to be considered as there is no global synchronisation, therefore the targets may read the outgoing event asynchronously. This should be supported without blocking the source module.
- A hardware description also poses timing constraints to the design, as a signal can not traverse arbitrary distances within a single clock cycle. Depending on the complexity of the specification it may therefore be required to buffer values in between to be able to resume their processing in the next clock cycle.

3.1. Communication

The main requirement posed to the communication between modules is that there needs to be some form of handshake or agreement between the communication partners, signalling that the transfer has been completed. This is necessary since there is no global synchronisation, and modules it cannot directly be foreseen when a module is actually outputting a new event or ready to process the next incoming event. A very popular choice for this purpose is the ready-valid interface, which is described in the following.

3.1.1. Ready-Valid Interface

The ready-valid interface extends each outgoing data port d by another outgoing flag v and an ingoing flag r. Those two ports are intended to signal whether the data laying at d of the source is valid (v), and if the target is ready (r) to read the value. The use of those two flags allows the definition of simple two-way handshake protocols. This type of communication is also used in the AXI interface[axi17].

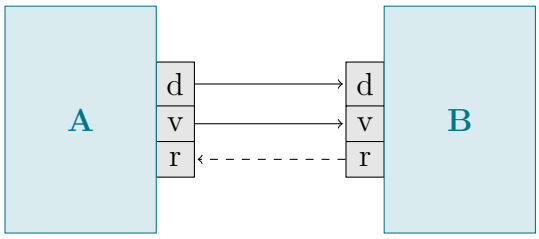


Figure 3.1.: Ready-Valid Interface

Figure 3.1 illustrates a simple example usage of the ready-valid interface. In this case, a data value should be transferred from module A to B. The communication channel features three wires:

- The *data* wire d, which is used to transfer the data from A to B
- The *valid* flag v, signalling B that the data on d is valid
- The *ready* flag r, signalling A that B is ready to process the incoming data

However, in the context of *TeSSLa* each stream has timestamps and values to transmit, which is not covered so far. In the following, different options of how to implement this are being discussed. As the properties and structure of a *TeSSLa* stream play an important role in evaluating those options, remember that a *TeSSLa* stream is an alternating sequence of timestamps with values. Additionally, note that the notion of progress defines up to which timestamp the stream is currently known.

1. One option would then be to transfer timestamps and values as a tuple over one channel. However, keeping the evaluation of recursive definitions in mind, where the progress was effectively decoupled of the value by being propagated through the cycle until a fixed point is reached, shows that handling timestamps and values as associated tuples would be problematic.

2. Another possibility would be to transfer timestamps and values over one single data channel d. This would stay close to the formal definition of a stream, by transferring timestamps and values in an alternating manner, thus also not requiring any further flag to identify which of both the current value is. However, values and timestamps will usually not be of the same data type,

thus requiring different bit lengths and interpretations for them. This would complicate the implementation as type casts and padding of values would be required.

3. Timestamps and values could also be transferred over separate channels, while sharing the r, v flags and using a third flag to differentiate between timestamps and values. This is significantly easier to implement as they are handled separately, however the drawback here is that the channel requires a higher bit width compared to the previous option.

Additionally, depending on how the input is being generated it may very well be that a timestamp and its associated value are both present at the same time. Being able to properly handle this would naturally increase the performance of the resulting design as well. However, for the sake of simplicity and consistency to the formal definition of streams and the *TeSSLa* operators, this is not directly considered yet. Therefore, the third option is chosen for this implementation, while the extension to support presence of both inputs at the same time is mentioned in the outlook in Chapter 7. This solution also allows to perform a progress update simply by sending multiple timestamps after each other. As an example, if a stream x would at a certain point in time be $x = 0\square 5\square 6\bot$, the last transmitted data was the timestamp 6. Updating the streams progress to $x = 0\square 5\square 7\bot$ would then result in sending the timestamp 7.

The here chosen interface is visualised in Figure 3.2, where

- The wire t contains the current timestamp value
- The signal $t?$ signals if the current value is a timestamp.

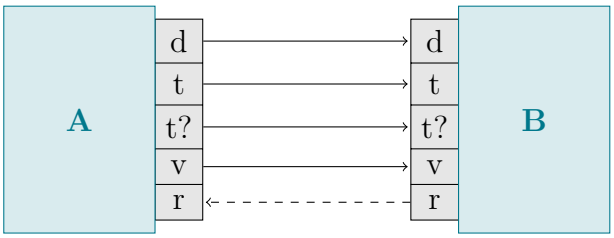


Figure 3.2.: Ready-Valid Protocol with a timestamp wire

The interface so far could also have been realised by using AXI instead. However, one very important limitation to AXI is that the ready-valid handshake has to be performed for every data transfer. This is an issue since the progress of a stream, denoted by its timestamp, can be transmitted even if the r flag of the target module is not set. As an example, given the stream $x = \mathbf{013}$ with an exclusive progress of 3. The occurrence of timestamp 3 at this point also provides the information that there will be no event on timestamp 2 occurring. While this information is

being processed, it may be that the stream progresses further to $x = \mathbf{017}$, signalling that there will be no events up to timestamp 7. Therefore, this is providing *more* information than the previous value, meaning that instead of finishing to process the previous one and then starting to process this new input, the new input can just be taken over directly. This optimisation requires allow timestamps to be overridden, which can not be accomplished with AXI.

Furthermore, the implementation of modules communicating with such a two-way interface needs to ensure that no combinatorial loops can occur, which would happen if v and r are defined in a way that they r is generated by a combinatorial circuit from v and vice versa, which is visualised in Figure 3.3. In other implementations

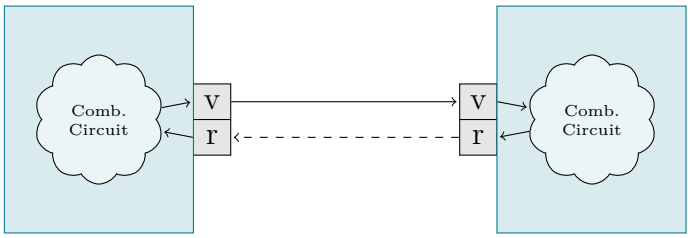


Figure 3.3.: A combinatorial loop using r and v.

like AXI [axi17], this is solved by only allowing v to be combinatorially generated from r, but not vice-versa. The same concept is also adapted here, although by allowing only r to be generated from v. This then results in the following guidelines for the communication between modules:

- Neither r nor v can be unset after being set, until the data transfer is complete
- The transfer is assumed to be performed at the rising edge of the next clock cycle when r and v are both set.
- The output of the target module sets the value of t according to the progress of the stream, even without r being set.
- v is not combinatorially generated from r
- The data transferred is consistent with the formal definition of *TeSSLa* streams, in particular the timestamps being strictly increasing and values are not overriding each other.

The following modules comply with those rules, and it is assumed that all modules they communicate with do so too.

3.2. Atomic Modules

After the discussing the communication, the next step is to define modules representing the *TeSSLa* operators. As described previously, the idea is to have each primitive stream operation of *TeSSLa* replaced by a module which represents that operator while retaining their semantics. Specifically, the primitive operations introduced in Subsection 2.1.1 are **time**, **lift** and **last**. In the following sections, a module definition will be derived from their respective semantics for each of those primitives. Those modules are described and explained step by step; The full pseudo-code for each module can then be seen in Chapter A.

Time

Intuitively speaking, the semantics of **time** introduced previously defines it as a unary function which returns a stream with the same timestamps as the input stream, but with the values replaced with the timestamp itself. As a unary operator, the respective module only requires a single input and output port. Additionally, as the input and output are both streams, they are using the ready-valid interface extended by timestamps which was introduced previously:

```
port in {...}
port out {...}
```

Furthermore, the module needs to store the last seen timestamp to be able to emit it as a value when a value occurs at the input, requiring a register to store it in. If there is a timestamp laying at the input, then this is obviously the most recent timestamp, otherwise the value in the register is still the most recent one.

```
register timeReg = 0

wire time = if in.valid && in.isTimestamp
            then in.timestamp else timeReg
timeReg := time
```

As **time** only manipulates the value without filtering or adding elements to the stream, the `ready`, `valid` and `isTimestamp` signals of its ports can just be forwarded. The `value` as well as `timestamp` signals of the output are simply the most recent timestamp `time`. This results in the full module shown in Listing A.1.

Lift

Intuitively, **lift** applies the function f on the current values of its two input streams. There are multiple things to consider here:

- At least one of the two inputs has a value (there are no events generated)
- The values have to be of the same timestamp. This requires synchronisation between both input streams on their timestamps.
- The progress of both streams needs to be known up to this timestamp included, since otherwise the value may still change.
- As the domain and range of f are extended by $\perp$, f can also filter out values depending on the input.

Apart of the two input streams, **lift** also takes the function f as parameter. As this is an arbitrary function and its implementation therefore varies depending on the context, it is beneficial to separate the function from the rest of the **lift** module. This can be accomplished by defining another port for the function, which has outgoing wires for both input values and an ingoing wire for the result of the function. This then allows to attach functions onto existing **lift** modules dynamically. For simplicity, it is assumed that the function computes within one clock cycle, thus not requiring any two-way communication. This results in the following IO definition:

```
port a {...}
port b {...}
port op {
  in aValid := ...
  in a := ...
  in bValid := ...
  in b := ...
  out outValid :=
  out out :=
}
port out {...}
```

As previously noted, the semantics of **lift** require a synchronisation between both operands on their logical time, since it is required that both streams are known up to the current timestamp. This can be realized by keeping one stream from progressing until the second one caught up to it with its progress, by setting the ready flags accordingly. First off, to keep both inputs synchronised, assert that the `ready` signal of an input should only be set if both inputs are currently valid:

```
a.valid && b.valid
```

Furthermore, either one of the following conditions has to hold true, which are defined here for `a`, whereas the conditions for `b` are analogue:

Progress: Since **lift** requires both operands to be known up to the current timestamp, the progress of **lift** is the lowest progress of its inputs. As the progress is also emitted as output, the `ready` signal of the output has to be set as well. Additionally, to allow the input with the lower progression to catch up, keep the other input from progressing until their timestamp values are the same. This results in the following constraint for `a`:

```
wire progress = a.isTimestamp && b.isTimestamp
wire progressA = a.timestamp <= b.timestamp
progress && progressA && out.ready
```

Value: If the input is a value `!a.isTimestamp`, it can be forwarded to the operator `op`. Asserting that the `valid` flag of that input is set is not required anymore, as both inputs are already synchronised by asserting that both are valid. Under the previously made restriction that the calculation of `op` finished within one clock cycle, there are two possible scenarios:

- The function call produced a resulting value, so `op.outValid` is set. In this case, since there is a value to emit and the input is not stored in any register, the output needs to be ready as well, otherwise the value would be lost.
- The function call filtered the value out, meaning that `op.outValid` is false. Here, no output is produced, therefore the input can simply be consumed.

This results then in the following assertion:

```
!a.isTimestamp && (!op.outValid || out.ready)
```

In conclusion, the circuit for the `ready` flag of `a` looks as follows:

```
ready := hasInput && (progress && progressA && out.ready
      || !a.isTimestamp && (!op.outValid || out.ready))
```

The input values can then simply be forwarded to the `op` port:

```
port op {
  aValid := !a.isTimestamp
  a := a.value
  bValid := !b.isTimestamp
  b := b.value
}
```

As previously noted, the progress of **lift** is the lowest progress of both inputs:

```
if progressA then a.timestamp else b.timestamp
```

The resulting value is then the result of applying the function f to the values of both input streams at the current timestamp. Since the synchronization was already taken care of and under the assumption that `op` calculates the function f, the wire `op.out` contains the value of the resulting stream for this timestamp.

The `valid` flag should then be set if both inputs are synchronized and there is either a progress or a value to emit. This results in the following definition for the `out` port:

```
port out {
  valid := hasInput && (progress || op.outValid)
  isTimestamp := progress
  value := op.out
  timestamp := if progressA then a.timestamp else b.timestamp
}
```

The full pseudo code of **lift** can be seen in Listing A.2.

Unary Lift

A unary lift could also be defined by using the binary lift. However, this is undesirable as a separate definition of a unary lift is significantly less complex, as shown in the semantics of *TeSSLa* in Subsection 2.1.2. In particular, in the case of lifting a unary function, there is no need for synchronisation between operands. For a hardware module, this means that no comparison of timestamps is required, which results in less logic cells required for the module, dependent of the bit width of timestamps. This plays also a major role in the first optimisation phase shown in Section 4.1.

The unary lift module uses a single input port `a` and output port `out`, and similar to the binary lift module, a port `op` as interface for the synthesised function f, which uses the wires `a`, `out` and `outValid`. Note that contrary to the binary lift, the operator port does not require a wire to determine if the input is valid, as there is only one input.

The input can then be consumed if one of the two following cases apply:

- The output is ready, in which case the input can be processed.

- The output is not ready, but the value has been filtered out, meaning that the input was a value `!a.isTimestamp` and the operator result not being valid `!op.outValid`

This results in the following definition of the input:

```
port a {
  ready := out.ready ||
           !a.isTimestamp && !op.outValid
}
```

The input is simply forwarded to the operator port:

```
port op {
  a := a.value
}
```

The output is then valid if:

- The input is valid timestamp, marking a progress update
- The input is valid and the operator did not filter the value out, meaning that `op.outValid` holds

The timestamp is then forwarded from the input, and the value is taken from the operator result, concluding in the following definition for the output:

```
port out {
  valid := a.valid && (a.isTimestamp || op.outValid)
  isTimestamp := a.isTimestamp
  value := op.out
  timestamp := a.timestamp
}
```

Again, the full pseudo code for this module can be seen in Listing A.3.

Last

As a reminder, the semantics of **last** are defined as $[\![\textbf{last}\ (e_1, e_2)]\!] = \text{last}([\![e_1]\!], [\![e_2]\!])$ where $\text{last} \in \mathcal{S}_{\mathbb{D}} \times \mathcal{S}_{\mathbb{D}'} \to \mathcal{S}_{\mathbb{D}}$ and $\text{last}(s, s') = s''$ such that

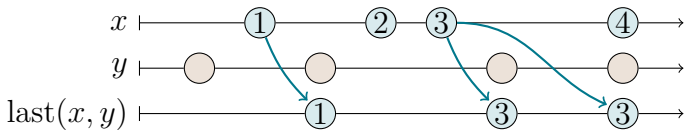
$$\forall_{t,d \in \mathbb{D}} s''(t) = d \Leftrightarrow s'(t) \in \mathbb{D}' \wedge \exists_{t'<t} s(t') = d \wedge \text{noData}(t', t) \tag{3.1}$$

$$\forall_t s''(t) = \bot \Leftrightarrow s'(t) = \bot \wedge \text{defined}(t) \vee \forall_{t'<t} s(t') = \bot \tag{3.2}$$

with

$$\text{noData}(t, t') := \forall_{t''|t<t''<t'} s(t'') = \bot$$
$$\text{defined}(t) := \forall_{t'<t} s''(t') \neq ?$$

Intuitively, the **last** operator returns the *last known* values of the first stream at the timestamps of the second one. For the following deductions, it is important to be aware of the edge case where both streams have an event at the same timestamp, as shown here at the occurrence of value 4 on x:



As you can see, the *last known* value is evaluated excluding the current timestamp. This is relevant as in this case, the value 3 has to be emitted, and the new value 4 has to be stored.

Since **last** is a binary operator, it requires two input ports (here called `value` and `trigger`, since the first stream provides the values for the result while the second stream triggers the output of events) as well as one output port. Hence the IO of the **last** module looks as follows:

```
port trigger {...}
port value {...}
port out {...}
```

The terms $s(t') = d$ and $\text{noData}(t', t)$ in (3.1) imply that **last** needs to know the last seen value (since there are no values between the timestamp t' and t, the value d is effectively the most recent value), thus requiring a register to store that value.

Additionally, (3.1) requires a comparison of timestamps t and t'. However, contrary to **lift** the timestamps do not require to be synchronised, hence instead of blocking both inputs, the timestamps of each input should rather be stored in a register. This allows the streams to still progress independently — however, with limitations introduced later — from each other.

As defined in (3.2), if the value stream s does not have any value yet up to a timestamp t, the result is undefined as well. For this, it is necessary to know if a value already occurred on `value`. Thus, a flag telling if the previously declared value register is initialised is needed.

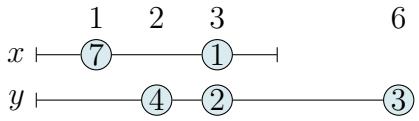
In total, this results in the following registers and initialization values:

```
register initReg = false
register valueReg
register triggerTimeReg = 0
register valueTimeReg = 0
```

The registers `valueTimeReg` and `triggerTimeReg` store the most recent timestamps for their respective input, (which is analogue to the register in Section 3.2):

```
wire valueTime = if value.valid && value.isTimestamp
then value.timestamp else valueTimeReg
valueTimeReg := valueTime

wire triggerTime = if trigger.valid && trigger.isTimestamp
then trigger.timestamp else triggerTimeReg
triggerTimeReg := triggerTime
```

last requires in (3.1) that there is no data between timestamps t' and the triggered timestamp t, therefore only requiring the last occurred value to be stored. If an event on the `trigger` stream occurs, the most recent value of the `value` stream should be emitted. However, since both inputs progress independently from each other, the currently stored value is only be known to be the most recent value if the progress of `value` is equals or higher than the one of `trigger`. Otherwise, the stored value might be outdated. At this point, `trigger` needs to be prevented from progressing any further until `value` caught up, as previous trigger events are not stored. To make it clearer, consider the following example with two input streams x and y:



The bar on their right side marks the progress of the stream. If the module would now compute **last**(x, y), it would not yet be possible to emit a value for the trigger event on y at timestamp 6, since there might be another event occurring on x at timestamps 4 or 5, hence returning the currently last seen value 1 of x would be incorrect.

Additionally, if `trigger` has a value, the module will emit a value, thus it requires `out` to be ready. An exception to this is if the value register was not yet initialized, in which case there is no value to emit and the trigger is ignored. This results in the following definition for the `ready` port of `trigger`.

```
port trigger {
ready := if trigger.isTimestamp then out.ready
else (triggerTime <= valueTime)
&& (!initReg || out.ready)
}
```

The next step is to define the logic when to actually store the value in `valueReg`. As previously defined, the `trigger` stops progressing when an event occurs until `value` caught up. This means that it is known that no other trigger events occurred on the interval between both of their progresses. Therefore, if there is a new event on `value` occurring, it can simply be overridden. However, this is only the case if the progress of `value` is strictly smaller than the one of `trigger`. If both streams have the same progress, the previously described edge case may occur, where both have an event at the same timestamp. In that case, the previous value needs to be emitted *and* the new value needs to be stored in the register. This means that the value is only allowed to be overridden if the previous value was already emitted. Therefore, if their timestamps are the same and an event occurs on `value`, the value can not be stored in the register yet, until either `trigger` progresses further, or an event on `trigger` occurs and the previous value is emitted. However, occurrence of an event on `trigger` alone is also not sufficient. It has to be ensured that the value can also be processed in this clock cycle, meaning that the either the output has to be ready or the register was not yet initialised, in which case the value can be freely stored.

The `ready` signal of `value` is then only set if it is a timestamp (in which case it would be passed to the `valueTime` register) or if the value can be stored currently. This ensures that the value does not progress past the trigger, which could lead to losing events. Also, as `value` is written to a register and not directly affecting the output, its `ready` signal does not require the `out` port to be ready.

```
wire triggered = trigger.valid && !trigger.isTimestamp
wire mayStoreValue = triggerTime > valueTime
                     || (triggerTime == valueTime
                        && triggered
                        && (!initReg || out.ready))
wire storeValue = value.valid && !value.isTimestamp
                  && mayStoreValue

port value {
  ready := value.isTimestamp || mayStoreValue
}
```

The register `valueReg` and flag `initReg` can then be set in dependence of `storeValue`:

```
initReg := initReg || storeValue
valueReg := if storeValue then value.value else valueReg
```

The final step is the definition of the `out` port. The timestamp to be emitted can simply be forwarded from the `trigger` input, as **last** can only emit values if a trigger value occured, and the `trigger` input not progressing past an occurrence of a trigger value. Hence the progress of **last** effectively is the progress of `trigger`.

The output is valid under two possible conditions:

- The trigger is a valid timestamp, meaning progress of the **last** module.
- The trigger is a valid data and the value register has been initialized, as well as the previously described constraint to their progress. This is the case where **last** emits a value.

```
port out {
  valid := trigger.valid && (trigger.isTimestamp
           || (triggerTime <= valueTime) && initReg)
  isTimestamp := trigger.isTimestamp
  value := valueReg
  timestamp := trigger.timestamp
}
```

3.2.1. Queues

In a *TeSSLa* specification, one stream a may be used in the definition of multiple other streams throughout the rest of the specification. As hardware, this means that the result of the module representing a has to be transferred to multiple different target modules. A simple way to approach this would be to wire the output of said stream to all target modules, and aggregating all `ready` flags of the targets to form the `ready` flag of a. However, this would effectively synchronise all outputs of a. Additionally, if one path would need significantly more clock cycles than the others, this solution would lead to stalling the other paths since they have to wait until all targets are ready, thus resulting in a bottleneck.

Concluding from this, it would be favourable to be able to buffer a certain amount of events, and dispatch them in an asynchronous way. A possible approach meeting those requirements would be to use a queue with a reading pointer for each output. Using multiple reading pointers allows asynchronous reading of the result by different target modules, and with an according depth it provides buffering of values to balance out differences in path lengths for the targets.

Definition 3.1 (Queue)**.** A queue of depth d and fan-out n is defined as

$$Q_{d,n} = (v, h, r_1, \ldots, r_n)$$

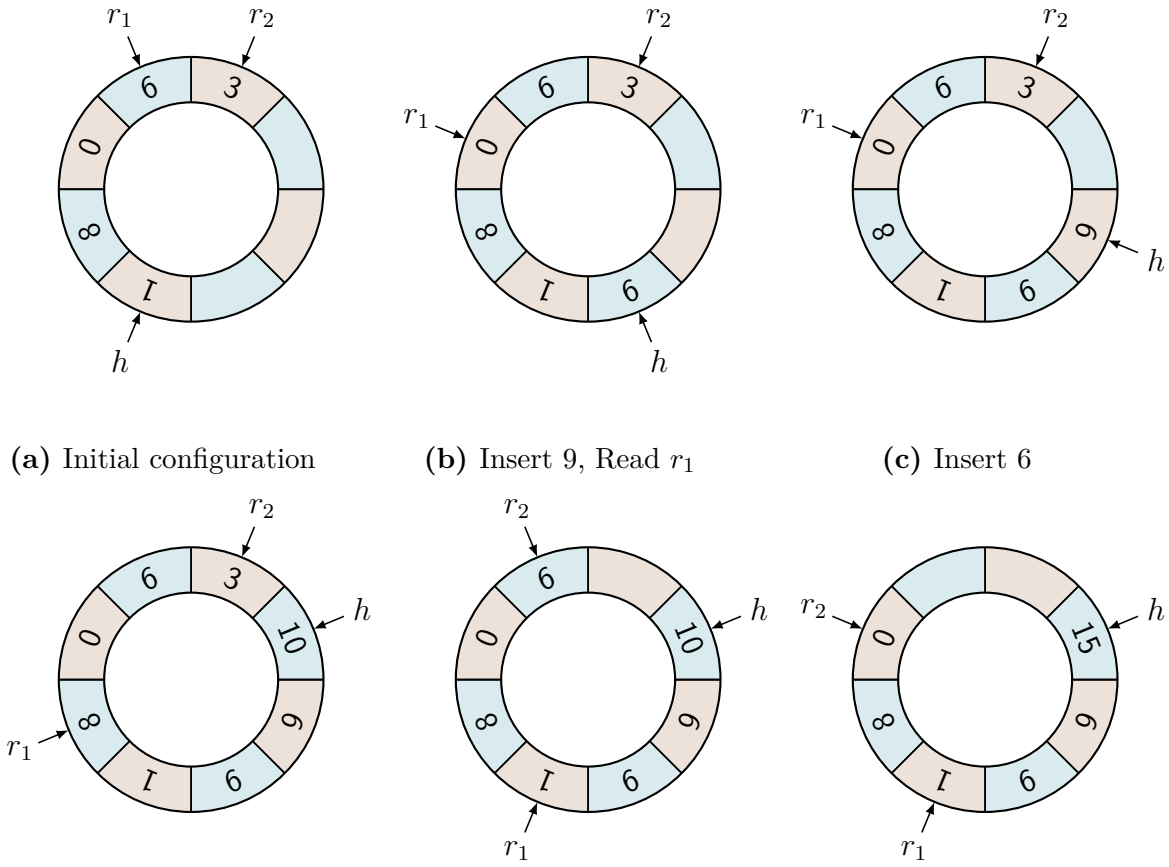
where v is an array of size $2d$, where the indices $0, 2, \ldots, 2d$ are used to store timestamps and the indices $1, 3, \ldots, 2d-1$ to store values. A head pointer $h \in \{0, \ldots, 2d-1\}$ pointing at the most recent value and $r_1, \ldots, r_n \in \{0, \ldots, 2d-1\}$ being the reading pointers.

The distinction of values and timestamps within the queue is necessary because timestamps are handled differently than values are. As long as the input of a queue is only updating its timestamp (progressing further without emitting a value), it is not needed to actually buffer every single timestamp occurring, as there are no values occurring in between those progressions. This was already discussed in Subsection 3.1.1. The opposite case of multiple valid data values occurring without any progression in between is semantically not possible and can therefore be ignored. This means that timestamps, contrary to values, are allowed to be overridden within the queue. This also leads to the conclusion that storing pairs of timestamps and values in the queue would be disadvantageous, as this would make it significantly more difficult to describe such a progress update.

The idea is to use the here defined data structure as a modified FIFO ring buffer with h pointing at the most recently written data, and for each output r_i pointing at the next data to be read by target i. A value can be considered as read if all r_i read said value. If the queue is full the inputs should be blocked such that no values are overridden. Additionally, since the decision whether or not to read a value in the previously defined modules is dependent on what kind of value is present, it is needed that the value can be *peeked* without it being removed from the queue. This feature of a FIFO is called *first word fall through* (FWFT). Overall, the here described structure differs from a usual FIFO ring buffer in the following aspects:

- Storing of two different types of data (timestamps and values), allowing overriding of timestamps
- Using multiple reading pointers to allow asynchronous reading.
- Using FWFT

In Figure 3.4, you can see an example run on a queue with depth 4 and two outputs with their associated reading pointers r_1 and r_2. The valid range of the queue is implicitly defined by h and the furthest reading pointer. Entries in blue coloured registers are assumed to be timestamps, while the other ones are data values. In this representation, the queue is processed in a counter clockwise manner.



(a) Initial configuration **(b)** Insert 9, Read r_1 **(c)** Insert 6

(d) Insert 10 **(e)** Read r_1, Read r_2 **(f)** Read r_2, Update head

Figure 3.4.: Example for $Q_{4,2}$.

As you can see in 3.4b, if a reading pointer other than the currently furthest pointer (here r_2) progresses, the valid range of the queue does not diminish as the value still awaits reading by other targets. After the value has been read by every participant in 3.4e, the register can be considered as free again.

In 3.4c and 3.4d there are values inserted up until the point that the queue is completely filled. At this point, the queue should signal its input that no more values can be accepted at this point.

Timestamps can override each other if no data values occur in between. This can be seen in 3.4f, where the last written input was the timestamp 10, which got overridden by a newer incoming timestamp 15.

Pseudocode

The here described module expects the depth to be of $d = 2^n, n \in \mathbb{N}$, since this allows the use of overflows to reset the pointers.

As a module, the previously described data structure requires a set of registers for timestamps and values respectively, as well as registers for the pointers h and r_i. Additionally, the scenario that all pointers are pointing at the same register can have two different meanings:

- The queue has one element and at least one of the reading participants still needs to read that value, or
- The queue is empty.

To differentiate between those two cases, a register where each bit is a flag to signal whether or not the queue for its associated participant is empty can be used. This leads to the following IO and register definition:

```
port in{...}
port out[n]{...}

register values[d]
register timestamps[d]
register head = 0
register tails[n]
register isEmpty = 2^n − 1
```

The registers `head` and `tails(i)` are expected to be of $\log_2(d)$ bits, and `isEmpty` to be of n bits.

First, it needs to be checked whether or not the queue is able to accept more inputs. As seen in Figure 3.4, there are two cases as to when the queue is ready: Either the queue is not full or the input would be a timestamp override. Deciding if the queue is full requires that for each reading participant, the reading pointer r_i differs from the next writing position $h + 1$, or r_i is at $h + 1$ but the value has already been read.

```
wire nextHead = head + 1
wire notFull = (isEmpty(0) || tail(0) != nextHead)
               && ...
               && (isEmpty(n-1) || tail(n-1) != nextHead)
wire headIsTimestamp = !head(0)
wire timestampOverride = headIsTimestamp && in.isTimestamp
wire ready = notFull || timestampOverride

in.ready := ready
```

If there is a valid input additionally to `ready` being set, it can be inserted into the queue. Inserting a value into the queue also requires an update of the pointer h, unless it was a timestamp override (see 3.4f).

```
wire writing = ready && in.valid
wire proceed = writing && !timestampOverride
wire writePos = if proceed then nextHead else head
head := writePos
```

To then write the value to a certain register, `writing` has to be set and the pointer `writingPos` needs to point at said register. Note that here, two separate sets of registers are used, therefore the pointer is split such that even and odd values point at the `values` and `timestamps` registers respectively.

```
values(i) := if (writing && writingPos == 2 * i + 1)
             then in.value else value(i)
timestamps(i) := if (writing && writingPos == 2 * i)
                 then in.timestamp else timestamps(i)
```

Next, each tail pointer r_i needs to be updated as well. There are 3 different cases to consider:

- The queue was empty for i in the previous clock cycle. Then it should be set to the current writing position.
- The output i is ready and r_i is different from the writing position. In this case r_i should be incremented by 1.

- The output i is not ready, or r_i is equal to the writing position (which is the case during timestamp overrides). Here, r_i should remain unchanged.

```
tails(i) := if isEmpty(i) then writePos else
              if out(i).ready && tails(i) != writePos then
                tails(i) + 1 else tails(i)
```

The final register to update is `isEmpty`. The queue can only be empty for i if there is no value inserted in the current clock cycle and:

- The queue was already empty for i in the previous clock cycle.
- The output i is ready and r_i is the same as h, meaning that there is only one value left to read.

```
isEmpty(i) := !writing && (isEmpty(i) ||
                           out(i).ready && tails(i) == head)
```

Finally, the wires of each output i can be defined:

- `valid` is set as long as the queue is not empty for i.
- `isTimestamp` is set if the pointer r_i is even, since it then points to a timestamp.
- `timestamp` and `value` are selected from the respective sets of registers at index $\lfloor \frac{r_i}{2} \rfloor$.

```
wire pos = tails(i) >> 1
out(i).valid := !isEmpty(i)
out(i).isTimestamp := !tail(i)(0)
out(i).value := values(pos)
out(i).timestamp := timestamps(pos)
```

Recursive Definitions

An important aspect to take into account when wanting to synthesise *TeSSLa* specifications on hardware are recursive definitions. In *TeSSLa*, streams can be recursively defined by using the **last** operator as shown in Subsection 2.1.3. With the concept of replacing every node in the data-flow diagram by a module, a cycle in the data-flow diagram would then also result in a cyclic dependency between modules. In such a case it has to be ensured that there is no combinatorial loop forming when synthesising those definitions on hardware. It can be assumed that a recursive definition will always contain at least one queue on its cyclic dependency. The reason for this is that a queue is required to dispatch events to multiple targets, meaning that if

the cycle does not contain a queue, the result of the recursive definition would never be used, making it obsolete, in which case the entire recursive definition could be removed.

Since a queue buffers the values in a register before transmitting them further, there is no possibility for a combinatorial loop to occur in a recursive definition.

Assuming that all atomic modules previously described represent the *TeSSLa* semantics of their associated operator correctly and that all specifications considered here are assumed to be well-formed, such a recursive definition then also computes the only fixed-point correctly.

3.3. Translation

All modules defined up to this point are sufficient to translate *TeSSLa* specifications into a hardware module. As *TeSSLa* is a data-flow oriented language its representation as hardware can be described as a transformation of the data-flow diagram, which is done here by use of an example.

Given the following specification, which contains the recursively defined stream `sum`, calculating the sum of all values on the input stream `x`:

```
in x: Events[Int]

def sum := merge(last(sum, x) + x, 0)
out sum
```

Listing 3.1: Example specification

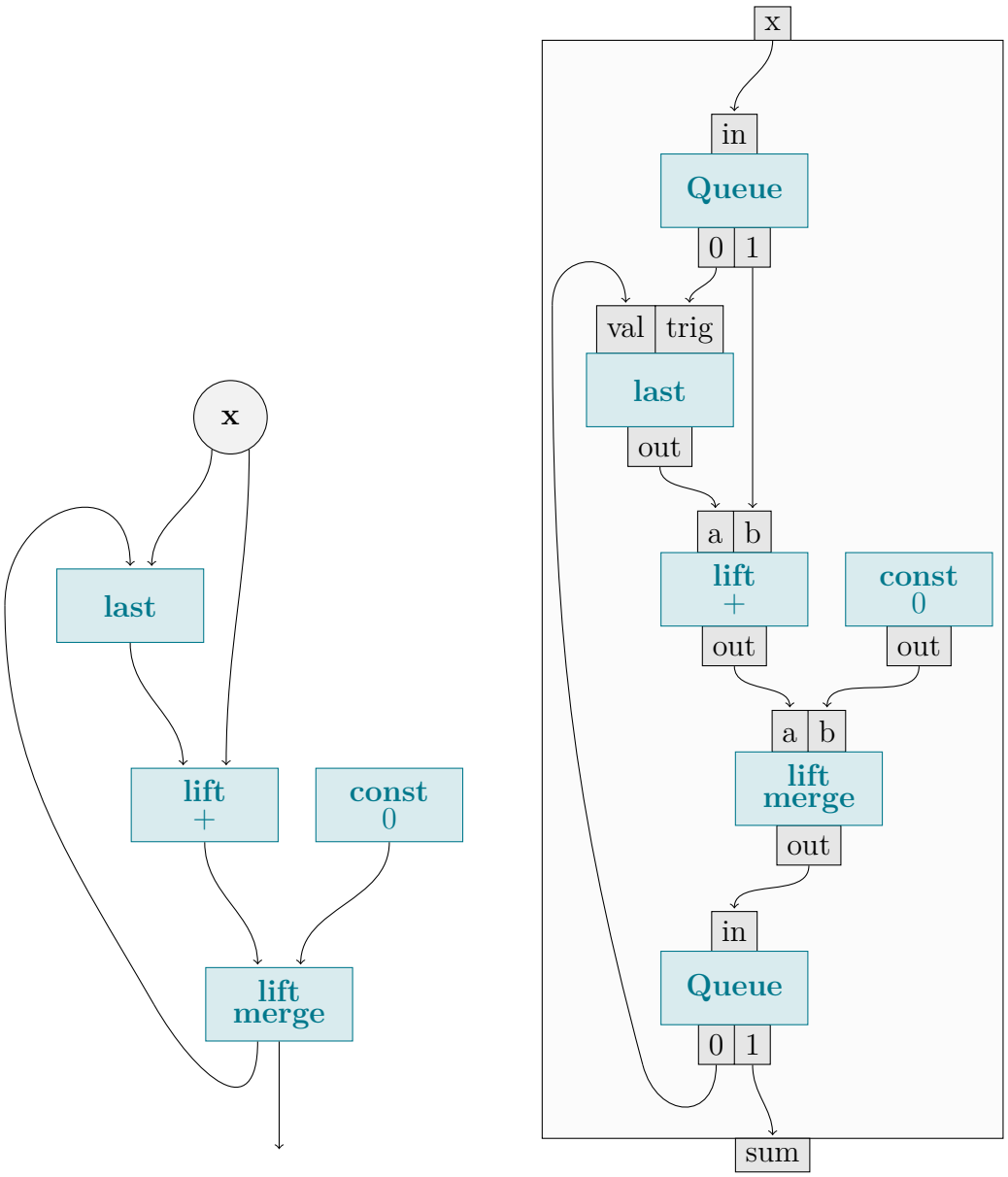
As stated in [CHL+18], every *TeSSLa* specification can be brought into a flat representation where each definition contains exactly one operator. This can be accomplished by moving each sub-expression into a new stream definition as follows:

```
in x: Events[Int]

def sum := merge(s2, 0)
def s1 := last(sum, x)
def s2 := s1 + x
out sum
```

This then directly represents the data flow graph shown in 3.5a. The specification can then be transformed into a module definition by replacing every node in this graph by its corresponding atomic module, and the edges within the data-flow diagram representing the wiring of the modules. Furthermore, every node with more

than one outgoing edge needs a queue module attached to it, which is responsible for handling the distribution of its events to the target modules. The resulting module graph of this specification is then shown in 3.5b.

The translation with use of IO adapters only differs in the way that the respective adapters (with their required queues) are wired around the specification module as shown in Figure 3.6.



(a) Data flow diagram **(b)** Module graph

Figure 3.5.: Data flow diagram and module graph of Listing 3.1

3.4. IO Adapters

If *TeSSLa* specifications have more than just one single input / output, the resulting hardware module would do so too. While this may be useful in most cases, like when incorporating the module as a part of a bigger hardware configuration, it can sometimes be unwanted. One example for this could be in an offline runtime verification scenario like in a log analysis. Here, all the input data to be processed would be stored inside a log file on the system and would need to be transferred to the hardware. Such a setup is also used in Chapter 6. For such a scenario, it would be preferable to be able to insert the inputs of multiple streams as one single aggregated, synchronised stream.

This mainly poses two requirements:

- Since such a aggregated, synchronised stream of data contains timestamps and events for every input, it is necessary to define of a binary event format as interface between the source and the specification module.
- Implementation of adapters which are able to interpret and accordingly modify the incoming data, as well as segregating and dispatching the events to the correct port on the specification module and vice versa (shown in Figure 3.6).

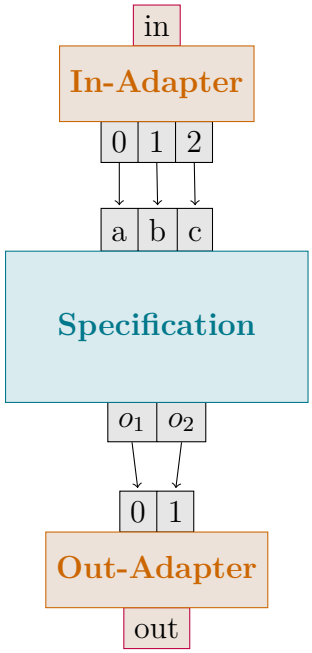


Figure 3.6.: A specification module with 3 inputs a, b, c and 2 outputs o_1, o_2, with adapter usage

In the following sections, the format for a single event and for such a synchronised input are defined. Afterwards, the input adapter and output adapter are defined.

3.4.1. Trace and Message Format

The first step is to define the expected structure of such a synchronised input. Given a specification with n input streams $s_i = t_{0,i}\ d_{0,i}\ t_{1,i}\ \ldots \in \mathcal{S}_{\mathbb{D}_i}, i \in \{1, \ldots, n\}$. Those streams can then be grouped by their timestamps and condensed into a single stream

$$s = t_0\ d_0\ t_1\ \ldots$$

where $d_i = s_0(t_i) \ldots s_n(t_i)$. The following example visualises this, by aggregating two streams x and y into a stream s. The timestamp values are marked bold for better readability:

$$x = \mathbf{1}\ 5\ \mathbf{3}\ \bot$$
$$y = \mathbf{1}\ 6\ \mathbf{3}\ 2$$
$$s = \mathbf{1}\ 5\ 6\ \mathbf{3}\ \bot\ 2$$

In the following, a synchronised input is expected to be in such a format, and also called *trace*.

Directly encoding the trace as shown here would have the advantage of not having to differentiate between timestamps and values with a flag, as the order is fixed (in the previously showcased example, each timestamp is followed by two values). Additionally, the order of the values directly reflects which stream they are associated with, thus also not requiring any addressing. However, this may lead to a significant overhead when only a few events are defined at the same timestamp, having to transfer $\bot$ with the same bit width of a data value for each other value. Therefore, to minimise the amount of values to transfer, a different approach is chosen:

1. A single bit flag is used to differentiate between timestamps and values.
2. An address is used to denote which stream a certain value is associated with. The corresponding field requires at least $\log_2(n)$ bits for n streams.

Figure 3.7 visualises the event format for a 16-bit channel and a 3-bit address range.

As *TeSSLa* supports arbitrary data types, it is required to transform the incoming binary data into the correct value of the data type it represents. However, this is solely dependent on the implementation and what data types it supports. Since this is of no further relevance here, those will be discussed in Chapter 5.

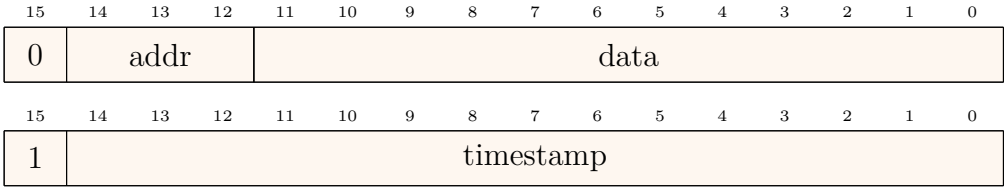


Figure 3.7.: Format of values/timestamps for a 16-bit channel with 3-bit address range.

The described approach allows to drop all $\perp$ values in the trace s when encoding. For the previously used example, this means that instead of 6 messages, only 5 messages would be required to transfer. With a total bit width of 8 bit, and an address width of 1 bit, the encoded format would look like follows:

```
1  0000001 -- timestamp 1
0 0 000101 -- x = 5
0 1 000110 -- y = 6
1  0000011 -- timestamp 3
0 1 000010 -- y = 2
```

The adapters then expect (respectively produce) a sequence of messages in such an encoded the format of s, where each t_j is encoded as timestamp and each element of d_j as a data message, skipping those which are $\perp$ at that timestamp.

3.4.2. Input-Adapter

The input adapter needs to split the incoming data into an address and a payload, which then is distributed to the according output. This process is visualized in Figure 3.8. One important aspect to discuss here is the use of storing mechanisms. Without buffering the decoded inputs, the input adapter would require every output out_i to have read the value before being able to read the next one. Additionally, for n input streams, the trace serving as input for the adapter has at most n values at a specific timestamp, one for each input stream. Based on to [LSS+19b, CHL+18] one can argue that if every input stream has a known value for a timestamp, at least one operation in the *TeSSLa* specification can fire. This can also be seen by analysing the module definitions for each atomic module; if there is new data at every input of a module, it will always consume at least one of those inputs. Therefore, to ensure that no deadlock can occur, the input adapter for n streams would need to be able to output at least n events independently from each other. A simple solution for this is to attach a queue of depth 1 at each output of the input adapter. This allows

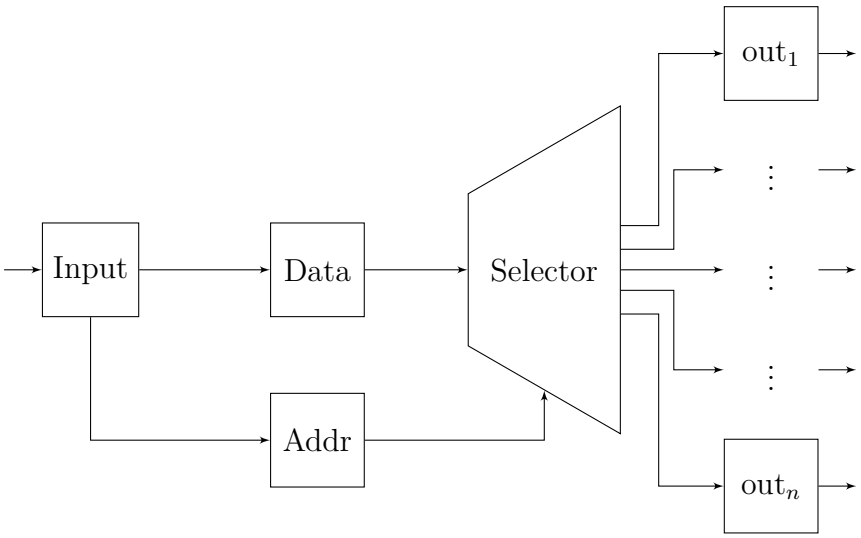


Figure 3.8.: Visualization of Input-Adapter

for a compact definition of input adapters, while buffering and dispatching is being handled by the queues.

The input adapter is a parametrized module, requiring the following three parameters:

- n being the amount of targets
- w_a being the bit width of the address range used, where it is expected that $w_a \geq \log_2(n)$
- w_i describing the bit width of the input, with $w_i > w_a + 2$.

Furthermore, $w_d = w_i - w_a - 1$ is used to describe the bit width of values, and $w_t = w_i - 1$ as bit width of timestamps. Both can be deduced directly by the format introduced in Figure 3.7.

The input adapter requires one single input port, which expects a bit sequence of size w_i. One important difference to the other module defined so far is that the input is here uninterpreted, meaning that using the extended ready-valid interface with timestamps would not make sense. Hence, the input is using the normal ready-valid protocol as depicted in Figure 3.1.

The amount of outgoing ports is then characterized by the parameter n. Since the adapter is used as an interface to the specification module, the outgoing ports are using the extended ready-valid interface.

```
port in
port out[n]
```

The data then needs to be split into the different fields as specified in Figure 3.7:

```
wire isTimestamp := in.value(w_i - 1)
wire addr := in.value(w_i - 2, w_i - w_a - 1)
```

Distributing the values is then covered by the following two cases:

- If the input is a timestamp, wait for all targets to be ready and then distribute it to them. This is needed as the trace is synchronised and all streams therefore progress simultaneously.
- If the input is a data value, set the valid flag for the associated target.

This leads to the following definition for the output ports:

```
wire allReady := out(0).ready && ... && out(n - 1).ready
for (i <- 0 to n - 1){
  out(i).valid := in.valid &&
                  if isTimestamp then allReady
                  else addr == i
  out(i).isTimestamp := isTimestamp
  out(i).timestamp := in.value(w_i - 2, 0)
  out(i).value := in.value(w_d - 1, 0)
}
```

Finally, the module needs to signal to its input if it can process the next input:

```
in.ready := if isTimestamp then allReady
            else out(addr).ready
```

3.4.3. Output-Adapter

The general idea of the output adapter is analogue to the input adapter: The incoming data from n different inputs need to be merged into a single output where in the format of Figure 3.7. One major difference here is that multiple valid inputs may occur at the same time, requiring a decision making on which input to process next. Adding to that, the resulting output should be a binary format of a valid *TeSSLa* trace, which requires the timestamps to be monotonically increasing. This means that processing the incoming values in the order they physically arrive is not an option, as their logical time needs to be considered.

The output adapter is parametrized with the same values n, w_i, w_d as the input adapter. The IO definition of the output adapter is as follows:

```
port in[n]
port out
```

Here, `out` is using the non-extended ready-valid interface, analogue to the input adapter.

As the timestamps of the resulting trace are expected to be monotonically increasing, the next timestamp should only be emitted if it is certain that no value will occur for the current timestamp anymore. This can be accomplished by synchronizing the inputs on their `isTimestamp` flag, such that every input currently is emitting a timestamp, and then progressing only those with the lowest timestamp value. This also assures that no values are emitted at incorrect timestamps, as the participants need to emit their timestamp first before being able to emit a value. Therefore, the timestamp to be emitted as well as the `isTimestamp` flag can be defined as follows:

```
wire isTimestamp = ⋀_i in(i).isTimestamp
wire timestamp = min_i{in(i).timestamp}
```

The address to emit can then be defined as the index of one valid input (the order does not matter as they all occur at the same timestamp):

```
wire isTimestamp = ⋀_i in(i).isTimestamp
wire timestamp = min_i{in(i).timestamp}
wire addr = switch{
  case in(0).valid && !in(0).isTimestamp -> 0,
  ...
  case in(n-1).valid && !in(n-1).isTimestamp -> n-1,
  else -> n
}
```

There are two cases to consider when deciding if an input i should be ready:

- A timestamp is emitted and this input emits the same timestamp, in which case this input should be allowed to progress further. Furthermore, all other inputs should be valid to assure that there is no possibility for a lower timestamp to occur in the future.
- A value is emitted and the selected address corresponds to this participant.

```
for (i <- 0 to n - 1) {
  wire othersValid = ⋀_{j,i≠j} in(j).valid
  in(i).ready := out.ready &&
                 if isTimestamp then
```

```
                    in(i).timestamp == timestamp
                    && othersValid
                  else addr == i
}
```

The output is then valid if all inputs are valid (for timestamps) or at least one valid value exists.

The output data is using the binary format shown in Figure 3.7. Since the address width w_a may be 0 in case of only one single output, this case needs to be handled separately.

```
wire allValid = ⋀_i in(i).valid
wire hasValidData = addr != n
out.valid := allValid || hasValidData
if (w_a == 0){
  out.data := if (isTimestamp) then
              1 ## timestamp_{w_o-1} else
              0 ## value_{w_d}
}else {
  out.data := if (isTimestamp) then
              1 ## timestamp_{w_o-1} else
              0 ## addr_{w_a} ## value_{w_d}
}
```

The operator ## describes a bit concatenation operator, and the notation $\mathtt{x}_b$ represents the value x, zero-padded or trimmed to the size b.

This chapter introduced the basic concept to synthesise *TeSSLa* specifications on hardware, including adapters and a trace format for the use case of synchronised traces. This is already sufficient to translate *TeSSLa* specifications to hardware definitions. However, to potentially improve performance and resource usage of those specifications, the next chapter motivates and defines multiple optimisation phases. This concept, including the optimisation phases of the following chapter, has been implemented which is described in Chapter 5 and evaluated afterwards.

4. Optimization

In the previous chapter, the general translation schema for translating *TeSSLa* specifications to a module description was described. However, realising a system in hardware is often motivated with an aim towards high performance. Additionally, hardware always comes with only a limited amount of resources. Therefore, while the previously described translation is already sufficient to translate specifications to hardware descriptions, this chapter introduces multiple different optimisation phases, whose aim is to improve performance or reduce the amount of resources used on the hardware.

4.1. Stream Merging: Motivation and Definitions

The purpose of this optimisation is to reduce synchronisation overhead and thus also reduce the amount of logic required, by inferring whether or not two streams are implicitly synchronous and modifying the specification accordingly in such cases.

4.1.1. Motivation

For lifted functions, as seen in Section 3.2, both inputs are synchronised on their logical timestamp in case of a binary lift. As timestamps are numeric values, and in realistic use cases can be expected to be up a width of 32 or more bits, this is an expensive operation. However, a stream $s := \mathbf{lift}(f)(x, y)$ using a binary lift can be rewritten as unary $s' := \mathbf{lift}_1(f')(x')$ where x' is a stream of tuples such that $x'(t) = (x(t), y(t))$ for $x(t) \neq ?$ and $y(t) \neq ?$, and $f'(a) = f(\pi_1(a), \pi_2(a))$, where π_i is the projection on the i-th element of a tuple. Since this stream s' is using a unary lift, no synchronisation is required.
In general, the tupled stream x' of course still needs to be generated, which could be accomplished by $x' := \mathbf{lift}(g)(x, y)$ where

$$g(a, b) := \begin{cases} \bot & \text{if } a = \bot \wedge b = \bot \\ (a, b) & \text{else} \end{cases}$$

By this, the function application and the synchronisation logic got effectively decoupled. Naturally, in the general case this provides no benefit as the synchronisation still needs to be performed. However, depending on the scenario, this decoupling and merging of streams can be propagated through the dependency graph and lead to an overall reduction of synchronisation. One important thing to note here is that this is not always favourable, as merging multiple streams into one synchronised stream generally defeats the purpose of making optimal use of parallelisation on the hardware. However, in the particular case where streams are implicitly synchronous, merging them comes with no drawbacks.

The following examples are used to illustrate the different scenarios and how they could be optimised according to this. Additionally, some examples also highlight the cases in which such a process would likely not prove useful.

Example 4.1. Considering the following example specification written in *TeSSLa* using the syntax of the *TeSSLa* compiler, calculating $\frac{(x-1)!}{(x-5)!}$:

```
in x: Events[Int]
out (x - 1) * (x - 2) * (x - 3) * (x - 4)
```

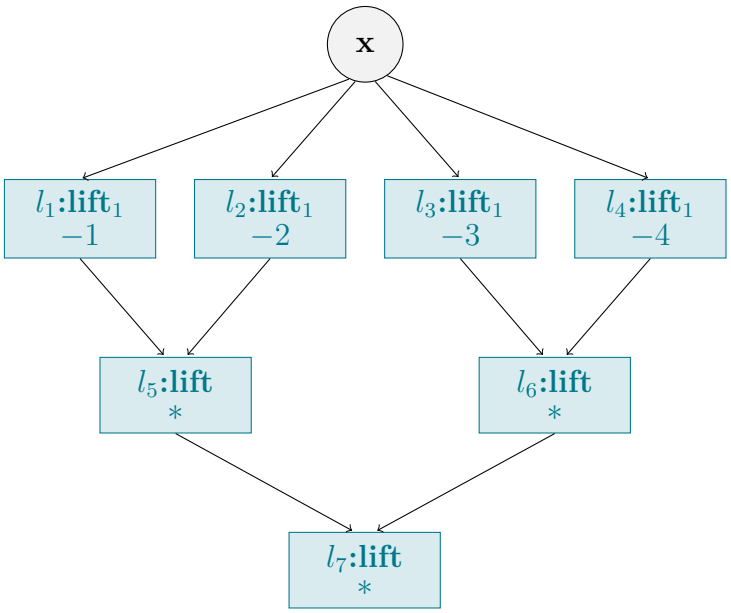


Figure 4.1.: Data-flow diagram for Example 4.1.

Assuming that all arithmetic operations get internally lifted to **lift** operations as already described in Subsection 2.1.3, this results in the data-flow diagram shown in Figure 4.1. It may seem counter-intuitive to not aggregate the entire chain of operations into one single liftable function $f(a) = (a-1) \cdot (a-2) \cdot (a-3) \cdot (a-4)$. However, considering that a function is assumed to be computed within a single clock

cycle, this may lead to timing issues, requiring to break up the function into multiple sub-functions to be lifted, such that the values can be buffered in between by using queues. By merging the sub expressions into a chain of unary lifted operations, the result of the here described operation will effectively be of such a form, which can be seen at the end of this example in Figure 4.3.

It is evident that without further modifications, this specification would require three pairwise synchronisations. Taking into account that every intermediate result here is generated solely by applying arithmetic operators on the input stream x, we can infer that their events will always occur at the same timestamps. This means that for every timestamp, the binary lifts shown here will have events either on both or none of their inputs, making the synchronisation obsolete.

In such a case where both inputs of a binary lift are already implicitly synchronous, it can be rewritten as a unary lift by merging its inputs into a tuple. In this case, l_5 and l_6 can be merged into a 4-ary **lift** $l_{5,6}$, taking l_1, l_2, l_3 and l_4 such that

$$l_{5,6} := \mathbf{lift}_4(f_{5,6})(l_1, l_2, l_3, l_4) \qquad \text{where } f_{5,6}(a, b, c, d) = (a \cdot b, c \cdot d)$$

This allows l_7 then to be redefined as a unary lift. This step is shown in Figure 4.2. In the next step, the inputs of $l_{5,6}$ also have events at exactly the same timestamps.

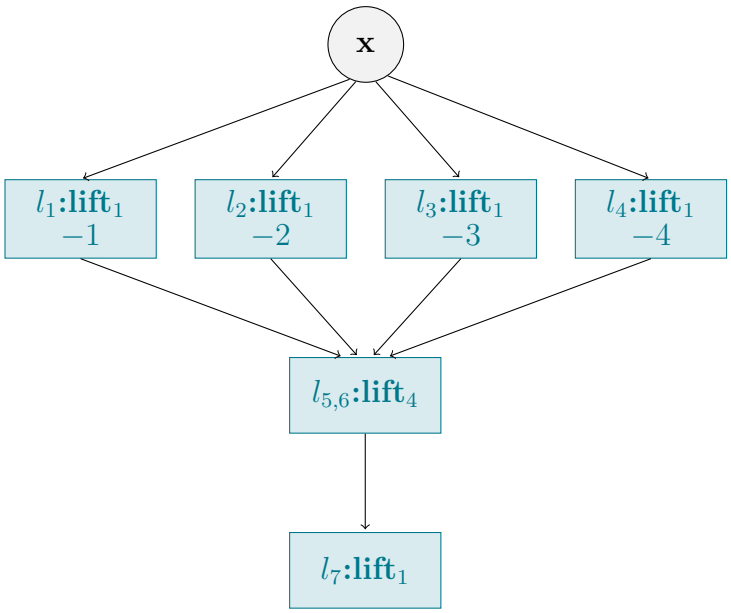


Figure 4.2.: Data-flow diagram for Example 4.1 after the first optimisation step.

Additionally, merging all 4 inputs into one single **lift** results in a unary lift, as all inputs are directly generated from the same input x, and also allows to rewrite $l_{5,6}$ as a unary lift as well.

In conclusion, the specification could therefore be rewritten as a chain of unary lift operators by merging the l_5 with l_6, and then l_1, l_2, l_3 and l_4 shown in Figure 4.3, resulting in the following definitions:

$$l_{1,2,3,4} := \mathbf{lift}_1(f'_1)(x) \qquad \text{where } f'_1(a) = (a-1, a-2, a-3, a-4)$$
$$l'_{5,6} := \mathbf{lift}_1(f'_2)(l_{1,2,3,4}) \qquad \text{where } f'_2(a) = (\pi_1(a) * \pi_2(a), \pi_3(a) * \pi_4(a))$$
$$l'_7 := \mathbf{lift}_1(f'_3)(l'_{5,6}) \qquad \text{where } f'_3(a) = \pi_1(a) * \pi_2(a)$$

where the synchronisation is omitted.

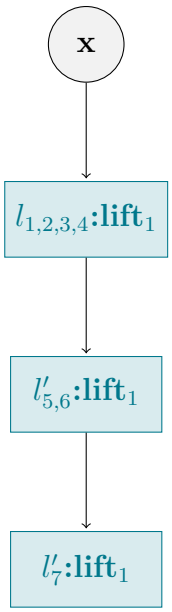


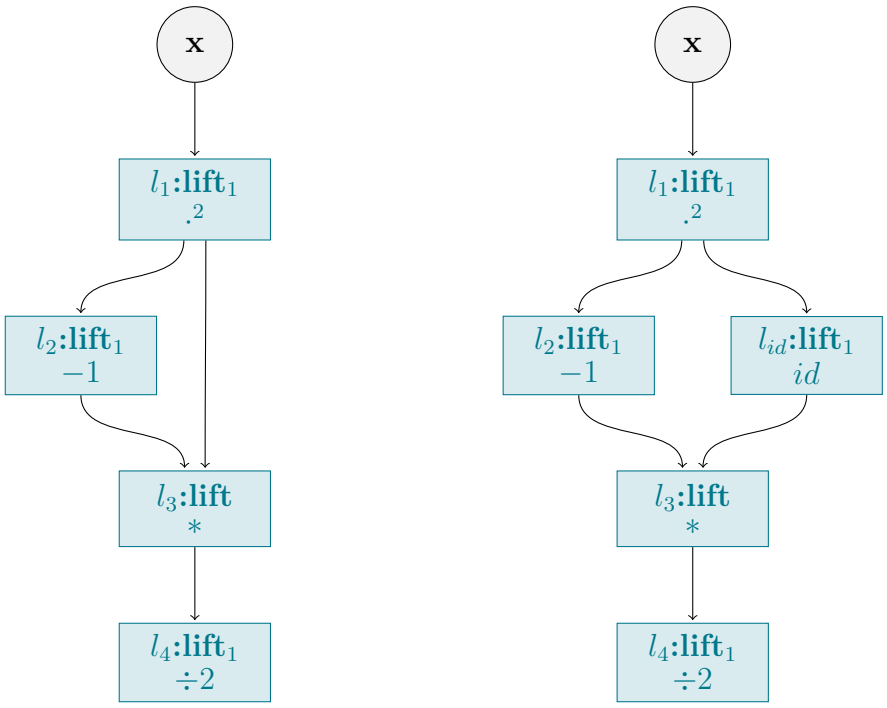
Figure 4.3.: Final data-flow diagram for Example 4.1.

Example 4.2. The first example depicted the general case for stream merging. However, it may also be that inputs of a stream are dependent from one another, in which case directly merging them is not an option. This scenario is analysed here. Consider the following specification:

```
in x: Events[Int]
def sqr = x²
out (sqr * (sqr + 1)) / 2
```

which calculates $\sum_{k=1}^{x^2} k$, with the associated data flow diagram shown in 4.4a. Just as in Example 4.1, there is a binary lift used even though it is easy to see that both inputs are already synchronuous on their logical time. The difference here is that the inputs of l_3 cannot simply be merged as l_2 is dependent of l_1. This can be solved by inserting a lifted identity l_{id} between l_1 and l_3 as shown in 4.4b, with

$$l_{id} := \mathbf{lift}_1(f')(l_1) \qquad \text{where } f(a) = a$$



(a) Initial data flow diagram. **(b)** Modified data flow diagram.

Figure 4.4.: Data flow diagrams of Example 4.2.

This allows the specification to then be merged in the same way as in Example 4.1, resulting in the following chain of unary lifts:

$$
\begin{aligned}
l'_1 &:= \mathbf{lift}_1(f'_1)(x) && \text{where } f'_1(a) = a^2 \\
l'_2 &:= \mathbf{lift}_1(f'_2)(l'_1) && \text{where } f'_2(a) = (a-1, a) \\
l'_3 &:= \mathbf{lift}_1(f'_3)(l'_2) && \text{where } f'_3(a) = \pi_1(a) * \pi_2(a) \\
l'_4 &:= \mathbf{lift}_1(f'_4)(l'_3) && \text{where } f'_4(a) = \frac{a}{2}
\end{aligned}
$$

Example 4.3. The two examples shown so far only considered cases where the streams were entirely synchronised on their timestamps. However, there may also be cases where streams are similar in terms of the occurrence of their events, but not exactly equivalent. For this, we consider the following specification, which performs a comparison of events on x with its previous events.

```
in x: Events[Int]
def prev := last(x, x)
out prev * 2 > x + 10
```

Looking back at the semantics of **last** in Subsection 2.1.2, you can see that the timestamps of **last** are a suffix of the timestamps of its second input (trigger), due

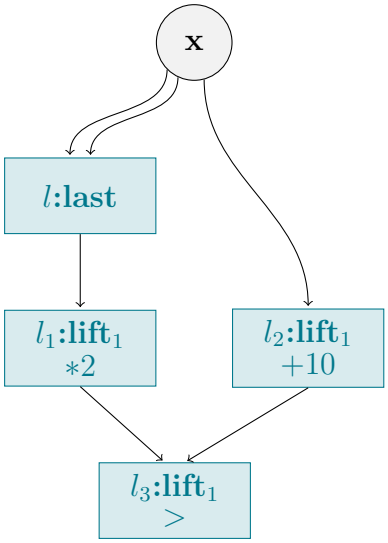


Figure 4.5.: Data flow diagram for Example 4.3.

to the fact that no result will be produced until the value stream is defined as well. From this follows that the two inputs of l_3 are not completely synchronous, but the sequence of timestamps of l_1 is a suffix of those of l_2. In particular, this means that there is a timestamp t after which both streams will be synchronous.

Specifically, here both streams are synchronous after initialisation of l_1. In such a case, it can still be desirable to merge the streams, under the assumption that the majority of both streams is synchronous. However, as the streams are not entirely synchronous, it is necessary to check if l_1 is already defined. This then results in merging l_1 and l_2 into $l_1' := \mathbf{lift}(f_1')(l, x)$ with:

$$f_1'(a, b) = \begin{cases} (a * 2, b + 10) & \text{if } a \neq \bot \\ (\bot, b + 10) & \text{else} \end{cases}$$

and rewriting l_3 as unary lift $l_2' := \mathbf{lift}_1(f_2')(l_2')$ where

$$f_2'(a) = \begin{cases} \pi_1(a) > \pi_2(a) & \text{if } \pi_1(a) \neq \bot \\ \bot & \text{else} \end{cases}$$

Note that in this specific case there is only one single merge performed, thus there was no direct benefit from it. However, the here described step can be propagated and combined in the same way as shown in the previous examples.

The inputs l and x of the newly created stream l_1' actually fall under the same category, with both streams being synchronised after l is initialised. The issue here is that they cannot be merged into a shared **lift** as l is a **last** operator. This is

not considered here, but possible approaches to optimise such cases are discussed in Section 7.1.

Example 4.4. This example now focuses on recursive definitions and how they should be handled, showing that in some cases, merging streams in a recursion might actually result in a worse performance. The following specification contains two recursive definitions y_1 and z_1, where y_1 counts occurrences of events on x, and z_1 adopts the new counter value on every 5th event.

$$y_1 := \mathbf{last}(\mathbf{merge}(\mathbf{lift}_1(f_1)(y_1), 0), x) \quad \text{where } f_1(a) = a + 1$$

$$z_1 := \mathbf{last}(\mathbf{merge}(\mathbf{lift}(f_2)(y_2, z_1), 0), x) \quad \text{where } f_2(a, b) = \begin{cases} a & \text{if } (a \bmod 5) = 0 \\ b & \text{else} \end{cases}$$

$$r := y_3 - z_3$$

For the following, the flattened form of this specification is used:

$$\begin{aligned} y_1 &:= \mathbf{last}(y_3, x) \\ y_2 &:= \mathbf{lift}_1(f_1)(y_1) \\ y_3 &:= \mathbf{merge}(y_2, 0) \\ z_1 &:= \mathbf{last}(z_3, x) \\ z_2 &:= \mathbf{lift}(f_2)(y_2, z_1) \\ z_3 &:= \mathbf{merge}(z_2, 0) \\ r &:= y_3 - z_3 \end{aligned}$$

The resulting data flow diagram is then shown in Figure 4.6. As none of the lifted operators perform any filtering of events, and both recursive definitions are triggered by x, both recursions will have the same timestamps. According to the examples seen so far, one could now decide to merge y_3 with z_3 and then z_2 with y_2 by passing the value through an identity operation.

However, as the recursion z is dependent of the other recursion y, merging both of them would result in a single recursion with a longer path. This may then result in requiring to place more queues within the cycle to counter potential timing issues, and therefore increasing the latency within the cycle. As a cycle has to be completely process an event before the next event can be processed, an increased latency inside a cycle directly decreases the throughput of the entire path.

Therefore, when considering cycles, merging should only be considered if the to be merged streams are not of two separate cycles.

Example 4.5. This final example represents the scenario of having streams with entirely different timestamps, in which case merging streams would provide no gains

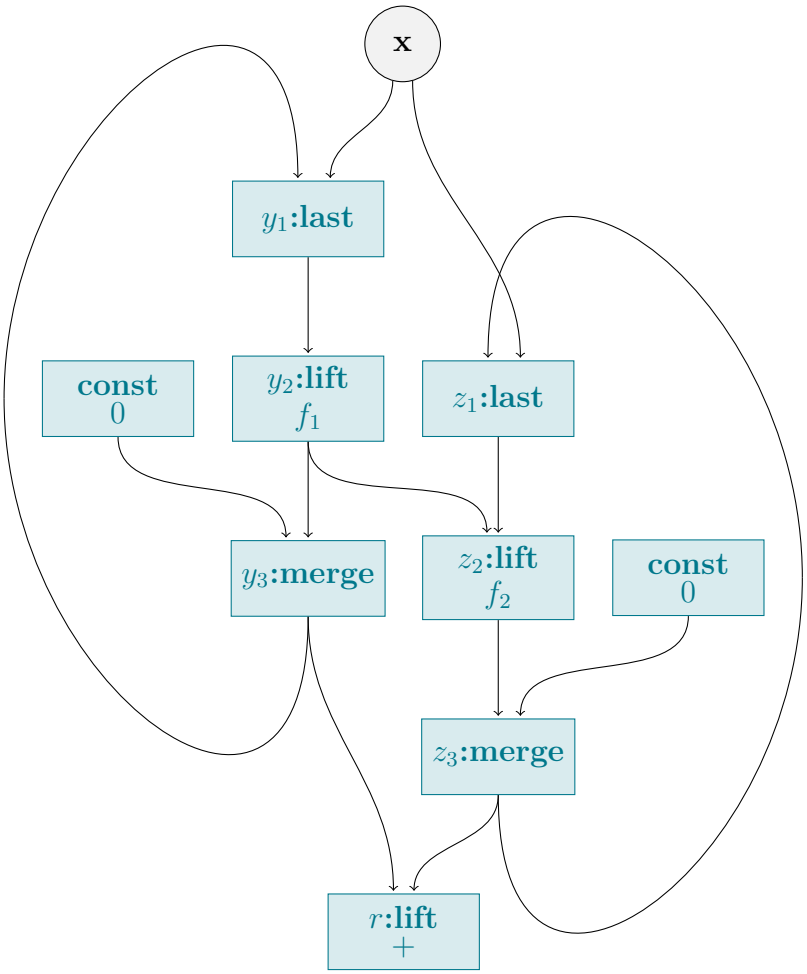


Figure 4.6.: Data-flow diagram for Example 4.4.

or even have negative effects. In this example, two inputs x and y are processed in parallel in two recursions z_1 and z_2. Finally, both partial sums are aggregated in the end:

```
in x: Events[Int]
in y: Events[Int]

z1 := merge(last(z1, x) + x, 0)
```

```
z2 := merge(last(z2, y) + y, 0)
out y + z
```

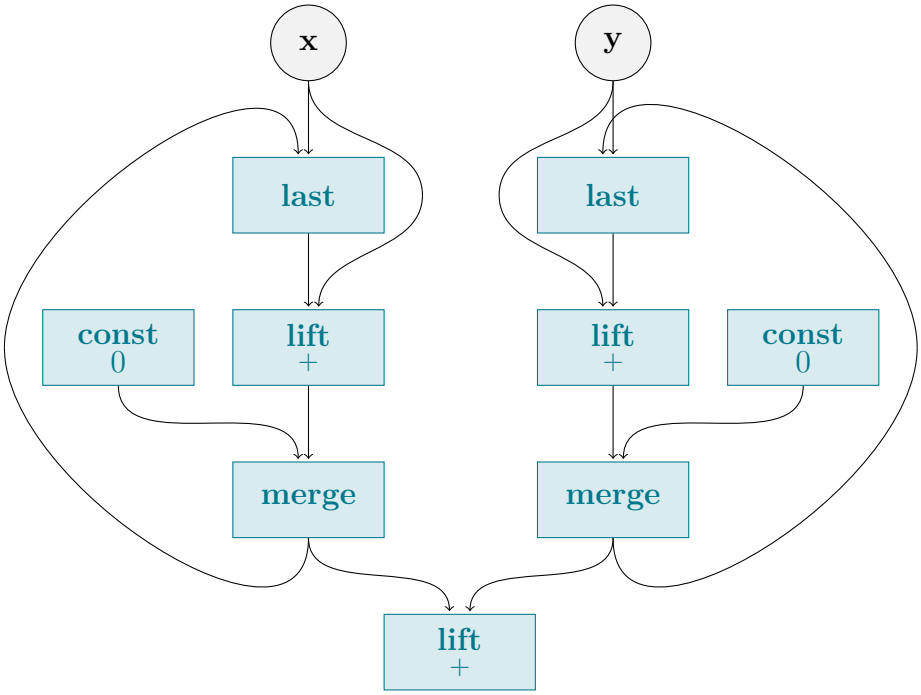


Figure 4.7.: Data-flow diagram of Example 4.5.

The resulting data-flow diagram is shown in Example 4.5. As both paths use different input streams, their timestamps may be entirely different. Furthermore, they also may differ in their frequency and delays of how often events occur on them. In this case, merging would result in synchronising both streams, and rewriting the specification as a single recursion, defeating the entire purpose of the parallel computation of those streams. Another example where merging is unwanted would be a segragation of a single input stream on multiple paths for an efficient parallel computation.

Those examples show that it is necessary to infer information about timestamps of events of a stream and defining a comparison between them to be able to identify the scenarios where stream merging should be applied. Thus, the goal of the following sections are to:

1. Define a classification and ways of comparing different stream definitions in respect to their timestamps.
2. Define rules for when and how to modify those definitions to reduce the amount of synchronisation needed.

4.1.2. Classification

The purpose of this classification is to partition the streams of a specification in accordance to the timestamps where their events will occur. Thus, for a specification φ with a set of stream definitions $A = \{x_0, \ldots, x_n\}$, the objective is to find partitions $A_i \subseteq A, 1 \leq i \leq m$, with $A = \bigcup_i A_i$, $i \neq j \implies A_i \cap A_j = \emptyset$ and

$$\forall_i \Big(x, y \in A_i \iff \forall_{s_1,\ldots,s_n} [\![\mathbf{time}(x)]\!]_{s_1,\ldots,s_n} = [\![\mathbf{time}(y)]\!]_{s_1,\ldots,s_n} \Big)$$

where $s_1, \ldots, s_n$ denote fixed input streams for this specification. Informally, this means that, if two streams are placed in the same partition, the evaluation of **time** — and thus the timestamps where their events occur — are always identical. However, the classification is taking place before the execution and therefore without an actual input for the streams. Furthermore, it is not possible to test this property for all possible combination of input streams. Thus, an abstraction of this property is required, which allows to classify the streams without considering input streams, while still entailing the described property of the partitions. Therefore, in the following section, timing expressions and a translation τ from *TeSSLa* definitions to those expressions is introduced, such that

$$\tau(x) \equiv \tau(y) \implies \forall_{s_1,\ldots,s_n} [\![\mathbf{time}(x)]\!]_{s_1,\ldots,s_n} = [\![\mathbf{time}(y)]\!]_{s_1,\ldots,s_n}$$

The general idea is to see each stream as a set of timestamps, which is then manipulated by the different stream operations. As an example, for the specification:

```
in x: Events[Int]
def r := x + 5
out r
```

the timestamp of x are unknown as it is an input stream, thus defining the abstraction $\tau(x) = v_x$, denoting that the stream x has a certain set of v_x of timestamps. As the lifted addition operation does not filter nor add any events, it can be deduced that $\tau(r) = v_x$ as well. This shows that by assigning symbols denoting the set of timestamps for each input stream, deductions on the timestamps of their dependent streams can be made.

Syntax

The syntax of timing expressions should be able to decently represent the modifications a stream operation can perform on the timestamps. As shown previously, a valid timing expression can be a variable symbol of the specification. Additionally,

to represent the timestamps of the constants **nil** and **unit**, the symbols $\emptyset$ and $\mathbf{0}$ are used.

As **last** (x, y) trims off events at the beginning of a stream, the notation of a suffix is required, which is denoted as y_x. Additionally, **lift** allows applying arbitrary functions to streams, allowing to filter events or also combine events of both streams. To classify the effect of those functions f on the timestamps, the operation symbols $\cup, \cap, \bar{}$ are used. Additionally, a fallback expression $f[x, y]$ is needed if the effect of f cannot be represented by those operations.

This results in the following syntax definition for timing expressions: For a set $x \in \mathbb{V}$ of variables of a specification φ, a timing expression is an expression t of the form

$$t ::= \emptyset \mid \mathbf{0} \mid v_x \mid t_t \mid t \cup t \mid t \cap t \mid \bar{t}$$

where $v_x \in \mathbb{V}'$ are variable symbols, using $x \in \mathbb{V}$ as index to associate this variable symbol with the symbol x in the specification.

Semantics

Definition 4.6 (Time projection)**.** Let $s \in \mathcal{S}_{\mathbb{D}}$ be an arbitrary stream on a time domain $\mathbb{T}$. The time projection $\delta_t : \mathcal{S}_{\mathbb{D}} \to \mathcal{P}(\mathbb{T})$ is a projection from that stream onto its timestamps

$$\delta_t(s) = \{t \in \mathbb{T} \mid s(t) \in \mathbb{D}\}$$

For a *TeSSLa* specification φ with input streams $s_1, \ldots, s_n$, those timing expressions can then be substantiated. This means that the semantics of a timing expression can be defined as a function mapping from a configuration for those input streams to the resulting set of timestamps. However, the timing of some streams can not be decided purely on the timing behaviour of inputs (for example when filtering a stream based on the occurring values). Therefore, we abstract from the specification in that case by considering those streams to be an input as well, adding it as parameter to the function. This then allows to define the semantics of a timing expression as a $n+k$-ary function mapping from n input streams, extended by k additional streams, to a set of timestamps:

$$[\![\cdot]\!]_t : \mathcal{T} \to \mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_{n+k}} \to \mathcal{P}(\mathbb{T})$$

with $\mathcal{T}$ being the set of all timing expressions. The semantics function is defined as follows:

$$[\![\emptyset]\!]_t = (s_1, \ldots, s_{n+k}) \mapsto \emptyset$$

$$
\begin{aligned}
[\![\mathbf{0}]\!]_t &= (s_1, \ldots, s_{n+k}) \mapsto \{0\} \\
[\![v_s]\!]_t &= (s_1, \ldots, s, \ldots, s_{n+k}) \mapsto \delta_t(s) \\
[\![u_v]\!] &= (s_1, \ldots, s_{n+k}) \mapsto \{t \in [\![u]\!]_t(s_1, \ldots, s_{n+k}) \mid t > \min([\![v]\!]_t(s_1, \ldots, s_{n+k}))\} \\
[\![u \cup v]\!]_t &= (s_1, \ldots, s_{n+k}) \mapsto [\![u]\!]_t(s_1, \ldots, s_{n+k}) \cup [\![v]\!]_t(s_1, \ldots, s_{n+k}) \\
[\![u \cap v]\!]_t &= (s_1, \ldots, s_{n+k}) \mapsto [\![u]\!]_t(s_1, \ldots, s_{n+k}) \cap [\![v]\!]_t(s_1, \ldots, s_{n+k}) \\
[\![\overline{u}]\!]_t &= (s_1, \ldots, s_{n+k}) \mapsto \mathbb{T} \setminus [\![u]\!]_t(s_1, \ldots, s_{n+k})
\end{aligned}
$$

where u, v describe arbitrary timing expressions, and v_s the variable symbol associated with the stream s.

In the following section, the translation from *TeSSLa* stream definitions to such timing expressions is defined.

Translation

For a *TeSSLa* specification φ with variables $x \in \mathbb{V}$, the translation schema τ mapping from an expression e of a stream definition to its timing expression is defined as follows:

$$
\begin{aligned}
\tau(\mathbf{nil}) &= \emptyset \\
\tau(\mathbf{unit}) &= \mathbf{0} \\
\tau(x) &= v_x \\
\tau(\mathbf{time}(x)) &= \tau(x) \\
\tau(\mathbf{last}(x, y)) &= \tau(y)_{\tau(x)}
\end{aligned}
$$

As for $\mathbf{lift}(f)(x, y)$, the function f to be lifted is able to filter or combine events of its input streams, and as such it has an effect on the timing of the resulting stream. Therefore, it is required to analyse said effect of function f to determine the resulting timing expression. As the function f is arbitrary, an abstraction is used for an easier classification of those functions. The chosen abstraction only considers functions where the information whether or not the result is defined can completely be inferred from the information which parameters are defined, meaning that it is not dependent from specific values. Those functions are represented by a set $\mathcal{F}$, where for $x \neq \bot$ and $y \neq \bot$:

$$
\begin{aligned}
\mathcal{F} = \{f : \mathbb{D}_1 \times \mathbb{D}_2 \rightharpoonup \mathbb{D} \mid\ & (\forall_{x,y} : f(x, y) \neq \bot \vee \forall_{x,y} : f(x, y) = \bot) \\
& \wedge (\forall_y :\ f(\bot, y) \neq \bot \vee \forall_y :\ f(\bot, y) = \bot) \\
& \wedge (\forall_x :\ f(x, \bot) \neq \bot \vee \forall_x :\ f(x, \bot) = \bot)\}
\end{aligned}
$$

Note that $\rightarrowtail$ is used to denote functions where the domains and range are extended by $\bot$.

Functions contained in $\mathcal{F}$ allow an abstraction f' of f where $f' : \{\top, \bot\} \times \{\top, \bot\} \to \{\top, \bot\}$ which only considers if a value is defined, and a homomorphism h mapping from f to the associated f', such that for all $x \in \mathbb{D}_1, y \in \mathbb{D}_2$:

$$f'(h(x), h(y)) = h(f(x, y)) \qquad \text{where } h(x) = \begin{cases} \top & \text{if } x \neq \bot \\ \bot & \text{else} \end{cases}$$

Resulting from the signature of f', it is evident that there are only 8 possible functions f'. This permits to introduce representative functions $f'_0, \ldots, f'_7$, which can then be used to classify the functions in $\mathcal{F}$ accordingly. The functions $f'_0, \ldots, f'_7$ are then defined as follows:

x	y	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7
$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\bot$	$\bot$	$\top$	$\top$
$\top$	$\top$	$\bot$	$\top$	$\bot$	$\top$	$\bot$	$\top$	$\bot$	$\top$

Finally, the translation of **lift** can be defined as follows.

$$\tau(\textbf{lift}(f)(x, y)) = \begin{cases} \emptyset & \text{if } f' = f_0 \\ \tau(x) \cap \tau(y) & \text{if } f' = f_1 \\ \tau(x) \cap \overline{\tau(y)} & \text{if } f' = f_2 \\ \tau(x) & \text{if } f' = f_3 \\ \overline{\tau(x)} \cap \tau(y) & \text{if } f' = f_4 \\ \tau(y) & \text{if } f' = f_5 \\ \overline{(\tau(x)} \cap \tau(y)) \cup (\tau(x) \cap \overline{\tau(y)}) & \text{if } f' = f_6 \\ \tau(x) \cup \tau(y) & \text{if } f' = f_7 \\ v_z & \text{if } f \notin \mathcal{F} \end{cases}$$

This abstraction provides an accurate interpretation of the effect for all functions in $\mathcal{F}$. In the final case, the behaviour of the function cannot be represented by any of the functions $f_0, \ldots, f_7$. In this case, a new variable v_z is introduced, where z is the variable symbol of this stream definition $z := \textbf{lift}(f)(x, y)$.

Using the *TeSSLa* semantics, it can then be shown that for every $x := e$, input streams $s_1, \ldots, s_n$ and additional streams $y_1, \ldots, y_k$ of a specification φ,

$$[\![\tau(e)]\!]_t(s_1, \ldots, s_n, [\![y_1]\!], \ldots, [\![y_k]\!]) = \delta_t([\![e]\!])$$

holds true, meaning that applying the substantiation of the timing expression, given the input streams and the evaluation of the additional streams $y_1, \ldots, y_k$, results in the same set of timestamps than evaluating the expression e with *TeSSLa* semantics and performing a projection on their timestamps does.

Equivalences

As the timing expressions need to be compared without the context of a specific input which would allow evaluation with $[\![\cdot]\!]_t$, it is necessary to define an equivalence relation.

Definition 4.7 (Equivalence)**.** Let u, v be two timing expressions. Then, define u as equivalent to v in respect to their timing, written $u \equiv_t v$, as follows:

$$u \equiv_t v \iff [\![u]\!]_t = [\![v]\!]_t$$

With the previous observation, this also entails that for two stream definitions $x := e_1$ and $y := e_2$,

$$\tau(x) \equiv_t \tau(y) \implies \forall_{s_1,\ldots,s_k}\ \delta_t([\![e_1]\!]_{s_1,\ldots,s_k}) = \delta_t([\![e_2]\!]_{s_1,\ldots,s_k})$$

stating that if the timing expressions associated with x and y are considered equivalent, their set of timestamps are the same for every possible configuration of input streams.

Using the previously defined semantics of timing expressions, we can observe that the usual equivalence properties from set algebra for the operators $\cup, \cap$ and $\overline{A}$ hold here. Additionally, we observe the following properties:

$$\begin{aligned}
y_x \cap \mathbf{0} &\equiv_t \emptyset \\
\mathbf{0}_x &\equiv_t \emptyset \\
\emptyset_x &\equiv_t \emptyset \\
x_\emptyset &\equiv_t \emptyset \\
y_{(\bigcup_i x_i)} &\equiv_t \bigcup_i y_{(x_i)} \\
(y_0)_x &\equiv_t y_x \\
(y_x)_x &\equiv_t y_x \\
\Big(\bigcup_i y_i\Big)_x &\equiv_t \bigcup_i (y_i)_x \\
\Big(\bigcap_i y_i\Big)_x &\equiv_t \bigcap_i (y_i)_x
\end{aligned}$$

So far, an abstraction for the timing of events, as well as a translation function from *TeSSLa* definitions to those expressions and an equivalence relation between them have been defined. This now allows us to define a classification of stream definitions as follows:

Definition 4.8 (Timing Classification)**.** Let φ be a *TeSSLa* specification with input variables x_i and stream definitions $y_i := e_i$. Let Y be the set of those stream definitions.

We define the timing classification of φ as a partition $A_i \subseteq Y$ with $Y = \bigcup_i A_i$ and $i \neq j \implies A_i \cap A_j = \emptyset$ such that

$$\forall_i \Big(y_j, y_k \in A_i \iff \tau(e_j) \equiv_t \tau(e_k) \Big)$$

Intuitively, this means that all the stream definitions in A_i, when evaluated for any configuration of input streams $s_1, \ldots, s_n$, will have events on the the same timestamps. This partition is therefore an abstraction of the partition defined at the very beginning of this section.

This timing classification can now be used to decide which streams should be merged with each other. However, the previous examples showed some special scenarios in which further information is needed. As Example 4.3 shows, the timings of two definitions don't have to be equivalent. Specifically, merging of two streams is still useful if the timing of one stream is a prefix of the other one. Therefore, we define a subsumption relation for timing expressions in this context:

Definition 4.9 (Subsumption)**.** Let u, v be two timing expressions. Then, call u subsumed under v, written $u \sqsubseteq_t v$, if

$$\{t \in v' \mid t \geq \min(u')\} = u'$$

where

$$u' = [\![u]\!]_t(s_1, \ldots, s_n, [\![y_1]\!], \ldots, [\![y_k]\!])$$
$$v' = [\![v]\!]_t(s_1, \ldots, s_n, [\![y_1]\!], \ldots, [\![y_k]\!])$$

for every possible configuration of input streams $s_1, \ldots, s_n$. Intuitively, this means that if $\tau(x) \sqsubseteq_t \tau(y)$, from the point on where x has its first timestamp, x and y have the same timing.

With the semantics of timing expressions, the following subsumption properties can be observed:

$$y_x \sqsubseteq_t y$$

$$\bigcup_i y_{(x_i)} \sqsubseteq_t y$$
$$\bigcap_i y_{(x_i)} \sqsubseteq_t y$$

Furthermore, as seen in Example 4.2, the merging procedure varies if inputs are dependent on each other. For those cases, a notion of dependence between two stream definitions is required.

Definition 4.10 (Dependence)**.** Given a *TeSSLa* specification φ with its associated data flow diagram $G = (V, E)$.

Then there is an acyclic modification of this diagram, $G' = (V, E')$ which is obtained by removing the edges going back into the initial **last** of each recursive definition. The reason for this is that a connection over such a path does not affect the value for this very timestamp, but for the next occurring event. This notion of dependence will be used to detect whether or not it is necessary to insert a lifted identity as shown in the previous examples, to then be able to merge them. This is not needed if they are only dependent over a delayed edge, hence removing those edges here.

A definition $y_1 := e_1$ is then called dependent of $y_2 := e_2$, written $y_2 \rightsquigarrow y_1$ if there is a directed path from y_2 to y_1 in the modified graph $G' = (V, E')$:

$$y_2 \rightsquigarrow y_1 \implies (y_2, y_1) \in E'^{+}$$

where E'^{+} is the transitive closure of E'.

Furthermore, Example 4.4 shows that it is not favourable to merge recursions which are dependent on each other, since that may actually reduce the throughput of the resulting specification. For this, it is necessary to be able to associate a node in the graph with its cycle and analyse how it stands in relation to other cycles in the graph. For this, strongly connected components can be used, which allow an abstraction of the graph by partitioning them according to their cycles. The following definition is derived from [Tar72].

Definition 4.11 (Strongly Connected Components [Tar72])**.** Given a directed graph $G = (V, E)$, a strongly connected component of G is then a maximal set $G' \subseteq G$ for which there is a directed path between any two nodes $u, v \in G'$:

$$\forall_{u,v} \in G' : (u, v) \in E^{+}$$

The graph G can then be partitioned into its strongly connected components $V' = \{V'_1, \ldots, V'_2\}$, allowing to define an acyclic graph S_G where each node is a strongly connected component of G:

$$S_G = (V', E')$$

$$E' = \{(X, Y) \in V' \mid \exists u \in X, \exists v \in Y : (u, v) \in E\}$$

An example for this is shown in Figure 4.8.

Furthermore, a node $u \in V_i'$ is called dependent of $v \in V_j'$ in respect to S_G, written $v \rightsquigarrow_{S_G} u$, iff $(V_i', V_j') \in E'^{+}$.

As an example, $z \rightsquigarrow_{S_G} b$ holds true in Figure 4.8.

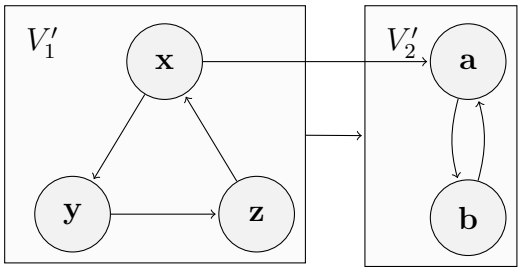


Figure 4.8.: Example for a graph G with S_G.

4.2. Stream Merging: Application

In the previous section, the different scenarios to account for were introduced, as well as all required notions to infer which case applies for a given input. In particular, the notions introduced included an equivalence relation for two timing expressions $\tau(x) \equiv_t \tau(y)$ for stream definitions x and y, their partitioning according to this relation, as well as subsumption $\tau(x) \sqsubseteq_t \tau(y)$, dependence $y \rightsquigarrow x$, and dependence regarding their strongly connected components $x \rightsquigarrow_{S_G} y$.

Using those definitions and the previously shown examples, this section now defines the stream merging procedure.

4.2.1. Identity and Accessors

Before being able to define the different scenarios on when and how to merge streams, some helper procedures need to be defined first. Those are then used later in this section.

As shown in Example 4.2, it can be that two inputs are dependent of each other, which can be alleviated by inserting a lifted identity to the specification. A lifted

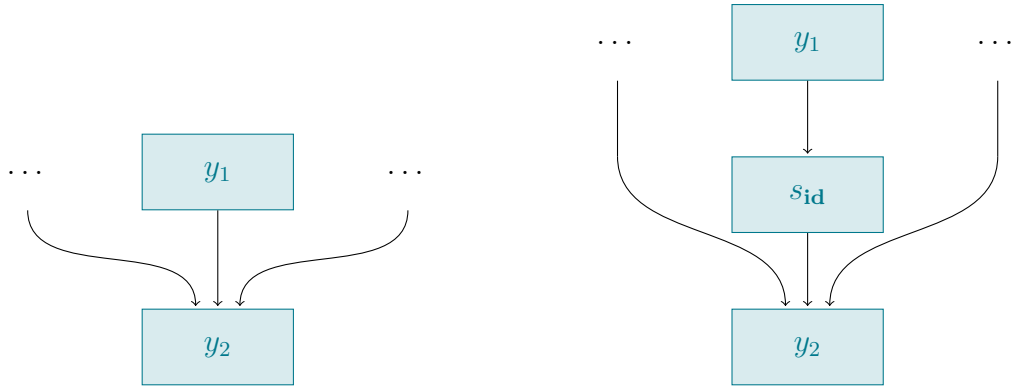
identity can be defined in a generalised manner as follows:
Given two stream definitions

$$y_1 := e_1$$
$$y_2 := \circ(a_1, \ldots, a_{k-1}, y_1, a_{k+1}, \ldots, a_n)$$

where $\circ$ is an arbitrary n-ary *TeSSLa* operator, with y_1 as parameter at the k-th position. Then, an identity stream description s_{id} can be added, modifying the definitions as follows:

$$y_1 := e_1$$
$$s_{\text{id}} := \mathbf{lift}_1(f)(y_1) \qquad \text{where } f(a) := a$$
$$y_2 := \circ(a_1, \ldots, a_{k-1}, s_{\text{id}}, a_{k+1}, \ldots, a_n)$$

This process is visualised in Figure 4.9.



(a) Initial data-flow diagram **(b)** Inserted s_{id} between y_1 and y_2

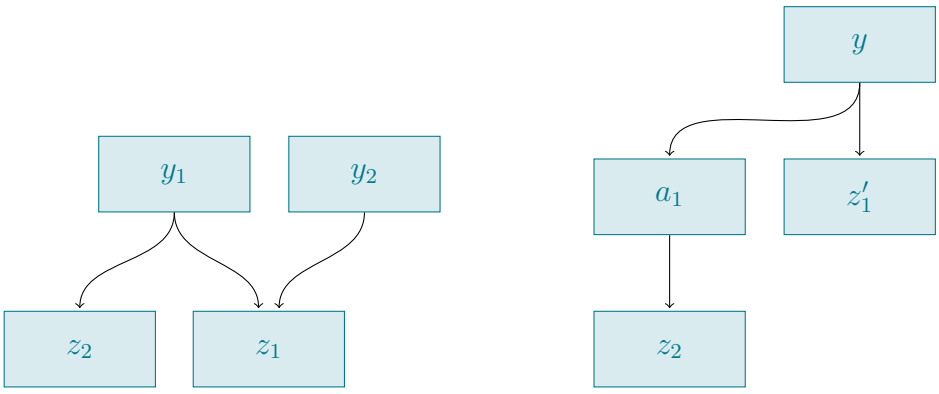
Figure 4.9.: Example for a lifted identity

Furthermore, stream merging groups multiple streams into one stream of tuples. However, it may be that one of those initial streams is still required by another definition. Therefore, a lifted accessor is required, extracting elements of those tuples:

Given a stream definition $y : \mathcal{S}_{\mathbb{D}_1 \times \cdots \times \mathbb{D}_n}$ representing a n streams in a tupled form. To extract the stream $\mathcal{S}_{\mathbb{D}_k}$ representing the k-th value of each event, the following stream definition can be defined:

$$a_k := \mathbf{lift}_1(f)(y) \qquad \text{where } f(a) := \pi_k(a)$$

An example for this can be seen in Figure 4.10, where on the right hand side, y_1 and y_2 have been merged into y, and a lifted accessor a_1 has been inserted to extract y_1 which is required by z_2.



(a) Initial data-flow diagram

(b) Merged y_1 and y_2 into y, and added accessor a_1.

Figure 4.10.: Example for a lifted accessor

4.2.2. Lift

Using the previously defined helper procedures, we can now define the general case for merging **lift** operators. Hereby, this is split into two parts, first introducing the stream merging for a binary lift, followed by the procedure for n-ary **lift**. This is required as stream merging may result in generating **lift** operators of higher arity than 2, as shown in Figure 4.2. As stated in the semantics of *TeSSLa* in Subsection 2.1.2, an n-ary **lift** can be represented by using binary lifts, allowing to break up those generated n-ary lift into binary lifts again. However, this would defeat the purpose of the optimisation as it may have been possible to merge multiple inputs of the n-ary **lift** and propagating this change through the dependency graph. Therefore, this optimisation step has to be defined for n-ary lifts as well. After processing the entire specification, any still existing n-ary lifts can then be represented through binary lifts again.

Binary Lift

In this section, the process of merging two **lift** operations into one merged **lift** operation is described. An example is shown in Figure 4.11. In particular, the following tasks have to be performed:

- Check if the scenario allows a merge.
- If it does, merge the inputs into a **lift** of according arity, and rewrite the target definition as unary **lift**.

(a) Initial data-flow diagram

(b) Merged y_1 and y_2 into y, and rewrite z as unary.

Figure 4.11.: Example for a merge of two **lift**s

Given two stream definitions

$$\begin{aligned} y_1 &:= \mathbf{lift}_{n_1}(f_1)(u_1, \ldots, u_{n_1}) \\ y_2 &:= \mathbf{lift}_{n_2}(f_2)(v_1, \ldots, v_{n_2}) \\ z &:= \mathbf{lift}_2(f_z)(y_1, y_2) \end{aligned}$$

where n_i describe their respective arities.

The streams y_1 and y_2 are eligible for merging if the following conditions hold true:

- They are not part of two different cycles which depend on each other:

  $$y_1 \not\rightsquigarrow_{S_G} y_2 \wedge y_2 \not\rightsquigarrow_{S_G} y_1$$

- The timing expression of one of them is subsumed by the other one:

  $$\tau(y_1) \sqsubseteq_t \tau(y_2) \vee \tau(y_2) \sqsubseteq_t (y_1)$$

  Note that this also holds true if $\tau(y_1) \equiv_t \tau(y_2)$.

If y_1 and y_2 fulfill those conditions, they can potentially be merged. It may still be that one of them is dependent of the other by

$$y_1 \rightsquigarrow y_2 \vee y_2 \rightsquigarrow y_1$$

However, this can be alleviated as described in Example 4.2, by using the previously introduced insertion of a lifted identity operation for the input causing the dependence. Therefore, for the following steps, y_1 and y_2 are assumed to not be dependent of each other.

The inputs y_1 and y_2 can then be merged into one $n_1 + n_2$-ary **lift** operation

$$y := \mathbf{lift}_{n_1+n_2}(f')(u_1, \ldots, u_{n_1}, v_1, \ldots, v_{n_2})$$

where in general

$$f'(a_1, \ldots, a_{n_1}, b_1, \ldots, b_{n_2}) = (r_1, r_2)$$
$$r_1 = \begin{cases} \bot & \text{if } \forall_i a_i = \bot \\ f_1(a_1, \ldots, a_{n_1}) & \text{else} \end{cases}$$
$$r_2 = \begin{cases} \bot & \text{if } \forall_i b_i = \bot \\ f_2(b_1, \ldots, b_{n_1}) & \text{else} \end{cases}$$

Catching the case where all a_i respectively b_i are $\bot$ is necessary, because f_1 are f_2 not directly lifted anymore. The semantics of **lift** already caught the case where all inputs are $\bot$, but since now both functions are nested into a new function f', this case needs to be handled explicitly. However, this is only the general case, and can be simplified depending on the inferred timing of streams. We can observe that for two streams x and y:

$$\tau(x) \sqsubseteq_t \tau(y) \implies (y(t) = \bot \Rightarrow x(t) = \bot)$$

This means that the set of parameters needed to check for $\bot$ in r_1 can be simplified to

$$r_1 = \begin{cases} \bot & \text{if } \forall_{a \in A} a = \bot \\ f_1(a_1, \ldots, a_{n_1}) & \text{else} \end{cases}$$

where $A = \{a_i \mid \forall_{j,j \neq i} : u_i \not\sqsubseteq_t u_j\}$ being the set of parameters whose associated stream is not subsumed under any other stream for this partition of inputs. If $A = \emptyset$, then all inputs are equivalent regarding their timestamps and any a_i can be used.

Additionally, if

$$\exists u \in \{u_1, \ldots, u_{n_1}\} \forall a \in \{u_1, \ldots, u_{n_1}, v_1, \ldots, v_{n_2}\} : a \sqsubseteq_t u$$

meaning that all other inputs of y are subsumed under u, r_1 can be further simplified to:

$$r_1 = f_1(a_1, \ldots, a_{n_1})$$

The reason for this is that the semantics of **lift** allows to assume that f' is always called with at least one stream not being $\bot$. As u subsumes all other streams, this means that the case for all inputs u_i being $\bot$ cannot occur.

Those simplifications can then also be naturally adapted to r_2.

Furthermore, the stream definition z can be rewritten as a unary **lift** z' as follows:

$$z' := \mathbf{lift}_1(f'') \qquad \text{where } f''(a) = f_z(\pi_1(a), \pi_2(a))$$

Finally, if there are other stream descriptions using y_1 or y_2 as well, a lifted accessor as introduced previously is inserted.

n-ary Lift

The previous description for a binary **lift** can for most parts be naturally extended for a n-ary **lift** as well. One notable difference here however is that for an n-ary **lift** not necessarily all n inputs are eligible to be merged together. Therefore, it is necessary to partition the inputs into maximal subsets such that each subset fulfills the previously noted conditions to be eligible for merging. The procedure to merge those partitions can be directly adapted from the binary lift. However, as a consequence of only merging subsets of inputs, the target stream z might not be rewritten as a unary **lift** as it was the binary case. Instead, it will result in a k-ary **lift** assuming that the inputs have been split into k partitions. As an example, if the stream definition in question is

$$z := \mathbf{lift}_5(f)(y_1, y_2, y_3, y_4, y_5)$$

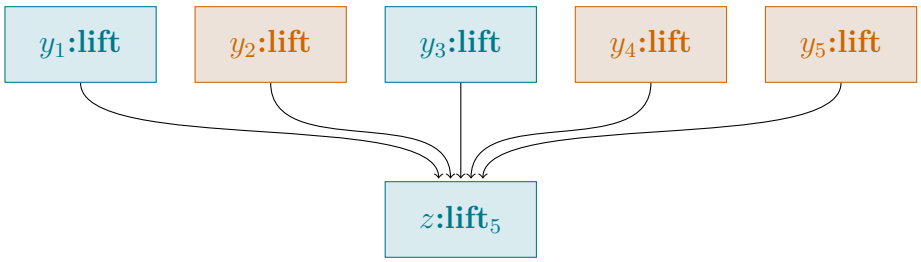
and assuming that the timing of inputs allows merging of y_1, y_3 into y'_1, and y_2, y_4, y_5 into y'_2 respectively, z could be rewritten as

$$z' := \mathbf{lift}_2(f')(y'_1, y'_2)$$

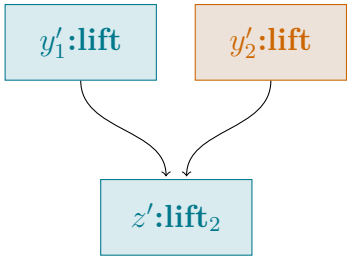
However, as a difference to the previous case, this requires explicit handling of the case where the value for y'_1 respectively y'_2 is $\bot$. Specifically, this means that f' is then defined as follows:

$$f'(a,b) = \begin{cases} f(\bot\ , \pi_1(b), \bot\ , \pi_2(b), \pi_3(b)) & \text{if } a = \bot \\ f(\pi_1(a), \bot\ , \pi_2(a), \bot\ , \bot\ ) & \text{if } b = \bot \\ f(\pi_1(a), \pi_1(b), \pi_2(a), \pi_2(b), \pi_3(b)) & \text{else} \end{cases}$$

A visualisation of this example is given in Figure 4.12, where the inputs are coloured depending on their timing.



(a) Initial data-flow diagram



(b) Merged inputs into y'_1 and y'_2, and rewrite z as binary.

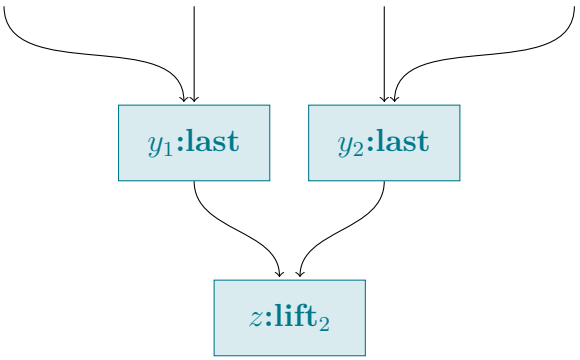
Figure 4.12.: Example for a merge of 5 **lift**s

Last

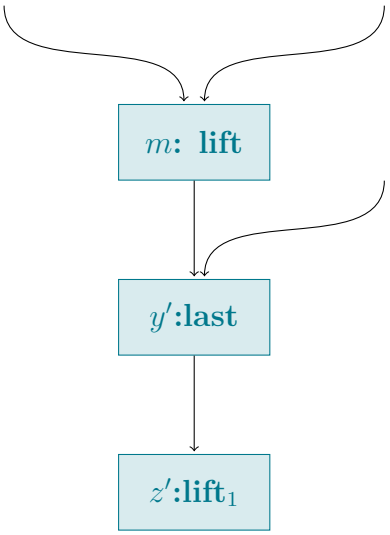
This case covers the scenario of merging n **last** having a shared **lift** as target:

$$\begin{aligned} y_1 &:= \mathbf{last}(u_1, v_1) \\ &\dots \\ y_n &:= \mathbf{last}(u_n, v_n) \\ z &:= \mathbf{lift}_n(f)(y_1, \dots, y_n) \end{aligned}$$

In this case, the general idea is to insert another **lift** definition which merges all u_i into one n-tuple, and using a single **last** to return the last known value of this tuple. This also effectively propagates the merge through **last**s, allowing to then merge the inputs u_i further, if possible. This process is visualised in Figure 4.13.



(a) Initial data-flow diagram



(b) Create tupled input in m, replace **last** and rewrite z as unary.

Figure 4.13.: Example for a merge of two **last**s

For a set y_i to be eligible for merging, the following conditions have to hold:

- They are not part of different, dependent cycles:

$$\forall_{i,j,i\neq j} : y_i \not\rightsquigarrow_{S_G} y_j \wedge y_j \not\rightsquigarrow_{S_G} y_i$$

- One of their value inputs has to subsume all other value inputs:

$$\exists_i \forall_{j,i \neq j} : \tau(u_j) \sqsubseteq_t \tau(u_i)$$

- The trigger inputs are be equivalent regarding their time:

$$\forall_{i,j,i \neq j} : \tau(v_i) \equiv_t \tau(v_j)$$

The condition for the trigger input are derived from the fact that applying **last** to a stream of tuples returns the previous tuple of values. This means that all triggers need to be equivalent regarding their timing, as otherwise it would generate unwanted events for some of the merged terms. The other conditions are similar to the previous case with **lift** and can directly be deduced from the examples.

Additionally, similar to the **lift** case, dependencies between the y_i are assumed to be resolved by inserting lifted identities at the according positions.

Then, the inputs u_i can be merged into a stream definition u as follows:

$$u := \mathbf{lift}_n(f')(u_1, \ldots, u_n) \qquad \text{where } f'(a_1, \ldots, a_n) = (a_1, \ldots, a_n)$$

and the **last** operations y_i can be replaced by a single **last** y with

$$y := \mathbf{last}(u, v_1)$$

Note that here, v_1 is used as trigger. Any other v_i would be valid as well, as they are all equivalent regarding their timestamps.

Finally, z can then be rewritten as a unary **lift**:

$$z' := \mathbf{lift}_1(f'')(y) \qquad \text{where } f''(a) = f(\pi_1(a), \ldots, \pi_n(a))$$

If only a subset of inputs of z are eligible for merging, the same solution as described for merging n-ary **lift** applies.

4.3. Bit Width Inference

In some use cases, there may be additional information known about the required (or sufficient) bit widths of values, allowing to adapt the bit width. This can lead to less space used on the actual hardware. In general, the underlying synthesis tool performs optimisations on the hardware description as well. However, using complex data types, which are being wrapped in a communication interface, stored in registers and dispatched with queues, it may be that those lower level optimisations are

not able to accurately detect every case where a lower bit width would be sufficient. Additionally, metadata provided with the specification can allow for further optimisation, and it can be used to take overflows into account as well by selecting the bit width of the result accordingly. For those reasons, a general way of performing a bit width inference on *TeSSLa* specifications is defined in this chapter.

The following example is used to highlight how a bit width inference for *TeSSLa* can be defined.

Example 4.12. Consider the following specification:

```
in x: Events[Int]
in y: Events[Int]

def u := x * 2
def v := u - y
def r := v % 4
```

For this example, we assume that there is additional information known about the inputs: Values of x are 6 bit signed integers, and values of y are 4 bit signed integers, meaning that their values are in the intervals $[-32..31]$ and $[-8..7]$ respectively.

Initially, the following bit widths for values of each stream definition can be deduced:

- Values of u are in $[-64..62]$ and require 7 bits.
- Values of v are in $[-71..70]$ and require 8 bits.
- Values of r are in $[0..3]$ and require 2 bits.

Then, considering how the modulo operator `%` affects its operands, and as v is not used in any other operator, it can be deduced that its left-hand side input v can therefore also be trimmed to 2 bits. This would then furthermore allow to trim u and y to 2 bits as well, and x to 1 bit.

Deducing from this example, the general idea is to first propagate the known input bit widths through the specification according to its data flow, which results in the required bit widths for each stream definition. In a second step, based on the context where a value is used, it can then be inferred how many bits are sufficient to still produce valid results for every input. These changes can then be propagated in the opposite direction to the data flow. Note that in the case of a recursive definition, this propagation may require multiple iterations through the cycle until a fixed-point is reached. Practically, such a fixed-point exists under the assumption that a maximum bit width for values is defined, and that there exists a total order on those values. However, reaching this fixed point may require an unfeasible amount

of iteration steps to reach, thus in the implementation requiring a limiting condition for the amount of iterations. In the following section, these steps are described in more detail and the required notions are introduced.

4.3.1. Algorithm

For the rest of this section, all data domains are treated as being finite, and it is assumed that there is a total order for them. Requiring the domains to be finite is no significant limitation, since values are in general limited by a maximum bit width anyway. Additionally, it is assumed that the total order is also reflects the bit width of those values in their binary representation, meaning that for $a \leq b$, the bit width of a is also less or equal to the bit width of b. In the following, a more general notion of such an interval is given, also called *value range*.

Definition 4.13 (Value range)**.** Given a domain $\mathbb{D}$ with a total order $\leq$ on this domain. Additionally, let $x := e$ be a stream definition of a specification φ, with input streams $s_1, \ldots, s_k$. Then, the interval $[a, b] \subseteq \mathbb{D} = \{d \in \mathbb{D} \mid a \leq d \leq b\}$ is called a value range for x if

$$\forall s_1, \ldots, s_k \quad \Big(\forall_t : [\![e]\!]_{s_1,\ldots,s_k}(t) \in \mathbb{D} \implies [\![e]\!]_{s_1,\ldots,s_k}(t) \in [a, b]\Big)$$

Meaning that every possible evaluation of x only has values within this interval.

For a specification φ with input streams of $\mathcal{S}_{\mathbb{D}_1}, \ldots, \mathcal{S}_{\mathbb{D}_n}$, initial value ranges are defined by their domain $[a_i, b_i] = \mathbb{D}_i$.

Those values can then be propagated along the dependency graph by calculating a resulting value range for each stream definition, based on the value ranges of its inputs:

Definition 4.14 (Value range generation)**.** Let y be a stream definition of φ which uses an arbitrary *TeSSLa* operator $f : \mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_n} \to \mathcal{S}_{\mathbb{D}}$. Then, a value range for y can be generated by a function $v_f : V_{\mathbb{D}_1} \times \cdots \times V_{\mathbb{D}_n} \to V_{\mathbb{D}}$ with

$$v_f([a_1, b_1], \ldots, [a_n, b_n]) = [a, b]$$

such that for all $s_1, \ldots, s_n, s$ with $f(s_1, \ldots, s_n) = s$:

$$\Big(\forall_i \forall_{t \in \delta_t(s_i)} : s_i(t) \in [a_i, b_i]\Big) \implies \forall_{t \in \delta_t(s)} s(t) \in [a, b]$$

Intuitively, this means that for streams $s_1, \ldots, s_n$, if v_f is applied to value ranges of those streams, the result is a value range for the resulting stream as well.

This general definition of the function v_f can then be used to define v_f for each *TeSSLa* operator.

- **nil** defines the empty stream, not returning any value. Thus, $v_{\text{nil}} = \emptyset$.
- As the domain of **unit** contains only a single value $\square$, the value range is always $v_{\text{unit}} = \{\square\}$.
- As **time** returns the current timestamp of a stream, the resulting value is a value of the time domain $\mathbb{T}$:

  $$v_{\text{time}}([a, b]) = \mathbb{T}$$

- **last** either filters or returns the value of its first input at a different timestamp:

  $$v_{\text{last}}([a_1, b_1], [a_2, b_2]) = [a_1, b_1]$$

- For **lift**(f), the effect of its application on the resulting values is solely dependent on the function f. Assuming f to be arbitrarily composed from a set of primitives $F = \{f_1, \ldots, f_k\}$, the value range $v_{\text{lift}(f)}$ can then be defined in dependence of those primitives as well, by defining the $v_{\text{lift}(f')}$ for each $f' \in F$ and then composing $v_{\text{lift}(f)}$ in the same manner as f.

  As an example, given primitives $F = \{+, \text{mod}\}$, the $v_{\text{lift}(f')}$ can be defined as follows:

  $$v_{\text{lift}(+)}((a_1, b_1), (a_2, b_2)) := (a_1 + a_2, b_1 + b_2)$$
  $$v_{\text{lift}(\text{mod})}((a_1, b_1), (a_2, b_2)) := (0, b_2 - 1)$$

  Then, a function $f(x, y) = (x + 5) \bmod y$ can have its value range computed by the function

  $$v_{\text{lift}(f)}((a_1, b_1), (a_2, b_2)) = v_{\text{lift}(\text{mod})}(v_{\text{lift}(+)}((a_1, b_1), (5, 5)), (a_2, b_2))$$

For each stream definition in a specification φ, such a function v_f can then be created and the initially defined value ranges for input streams can be propagated through the dependency graph. Therefore it is necessary to combine the value range of the previous iteration with the newly computed one.

So far, a general definition of the effect of *TeSSLa* operators on their value ranges and the propagation of those computed values has been defined. As this part of the computation only has an affect on its dependent stream definitions, this step is also called *forward propagation*. For the second part, as seen in Example 4.12, after having computed all value ranges for streams, it is possible to infer where value ranges may potentially be shortened without changing the outcome. This effect can be propagated in the direction opposite to the data-flow, therefore it is also called *backwards propagation*.

Definition 4.15 (Value range update)**.** Given a stream definition y using a *TeSSLa* operator $f : \mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_n} \to \mathcal{S}_{\mathbb{D}}$ and its computed value range $[a, b]$ as well as known value ranges $[a_i, b_i]$ for each input of y. Then, replacing the value range of the i-th parameter x_i by $[a'_i, b'_i]$ is called a value range update for y if the following conditions hold true:

- The binary representation of any value of $[a', b']$ requires less bits than of $[a, b]$
- For every possible input s_j:

  $$f(s_1, \ldots, s_i, \ldots, s_n) = s \implies f(s_1, \ldots, s'_i, \ldots, s_n) = s$$

  where s'_i is a modification of s_i with $s_i(t) \in [a_i, b_i] \implies s'_i(t) \in [a'_i, b'_i]$, projecting the values of s into the new value range by trimming its most significant bits.

Intuitively, this means that in such a case, a certain amount of bits may be trimmed from the i-th input without affecting the result.

Additionally, since the stream x_i may be used at other stream definitions apart of y, having different effects on the value range of x_i. Therefore, the updates are computed locally as $r_j = (a'_j, b'_j)$ for each y_j using x_i as input, and then aggregated as

$$(a'_i, b'_i) := \bigcup_j r_j$$

ensuring that the resulting interval is correct for every target y_j.

Analogue to the forwards propagation, this update can then be propagated throughout the specification.

Applying the here described algorithm to a specification φ results in every stream definition being annotated with a value range, which can then directly be used to deduce the bit width for the value port for that stream definition on the hardware.

4.4. Queue Placement

In Subsection 3.2.1, queues were introduced to provide the possibility to buffer values and to dispatch values to their target modules. As the use of queues is very resource-intensive, with an amount of registers used directly proportional to the bit width of values and timestamps, it is preferable to use as few queues as possible. Therefore, this chapter introduces a heuristics based algorithm to define the placement and depth of queues.

The timing of paths on hardware plays a crucial role for placing queues. In static timing analysis, the notion of *slack* is used to check for timing constraint violations. There, the slack is defined as the difference between the required arrival time of a signal and its actual arrival time $s = t_r - t_a$. As an example, on a frequency of 125 MHz, the period and therefore the maximum time a signal is allowed to need within one clock cycle is $t_r = 8\,\text{ns}$. A negative slack then denotes a violation of timing constraints by the design and therefore needs to be avoided.

The time a signal needs can be split into two main aspects: The logic, or more specifically the amount of look-up-tables required to implement the logic on a path, and the networking, meaning the distance the signal has to travel between the look-up-tables and registers. As both of those aspects are highly dependent on the specific hardware and synthesis tool used, and especially for network in general hardly possible to infer, the approach here is to use a heuristic to try and reduce the amount of queues required without invalidating the resulting hardware description.

Additionally, the depth of queues is also a relevant optimisation criteria, as an increased depth of a queue allows for more buffering, which depending on the specification increases the throughput, but at the same time uses up more resources. To further motivate this, consider the following example:

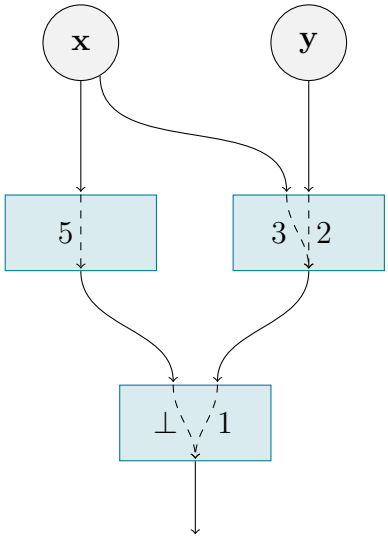
```
in x: Events[Int]
def l3 := last(last(last(x, x), x), x)
def diff := l3 - x
out diff
```

In this example, computation of the left input of `diff` is a chain of three **last** operators, which each require storing of the incoming value in a register. Thus, that path requires three clock cycles to compute, while the right hand side takes the value at x directly. This means that the input at x can only be consumed if the final **lift** operator is ready to consume it. The difference in path lengths here would then cause the specification to only be able to consume one message every 4 clock cycles. Adjusting the depth of x to 4 would allow full pipelining of the left path and thus allow to consume one message per clock cycle.

4.4.1. Placement

To estimate the amount of logic a certain path requires on the hardware, a weighing for the data-flow diagram is used. As modules can have multiple inputs, which then also traverse different paths throughout the module, it is necessary to differentiate between each input signal and their passage through a module to deduce the timing. This is represented by assigning each stream definition y using an n-ary *TeSSLa*

operator $f : \mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_n} \to \mathcal{S}_{\mathbb{D}}$ an n-tuple of internal weights $in_y \in (\mathbb{N} \cup \{\bot\})^n$ which are empirically determined. The value at position i in the tuple denotes the weight of that path through the module representing this stream definition. An example is shown in the following figure:



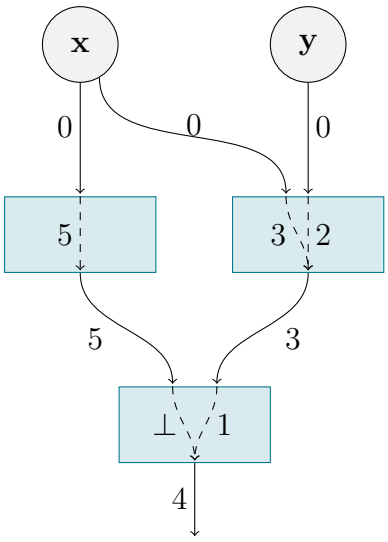
Such a data-flow diagram, extended with the internal weights for each stream definition, can then be used to compute the weighting. The weighting of a stream definition $y := f(x_1, \ldots, x_n)$ is defined as a function $w_y : \mathbb{N} \times \cdots \times \mathbb{N} \to \mathbb{N}$, mapping from weights of all incoming edges to the resulting output weight, where

$$w_y(v_1, \ldots, v_n) = \max_{1 \leq i \leq n} \begin{cases} 0 & \text{if there is a queue on the edge } (x_i, y) \\ v_i + \pi_i(in_y) & \text{else} \end{cases}$$

where $\pi_i(in_y)$ denotes the projection on the i-th value of in_y.

Applying this weighting function iteratively on the graph where the edges of input streams are weighted with 0 then allows to fully weight the graph. Recursive definitions pose no issue here since they cannot have a combinatorial loop, as discussed previously.

Using the previous example again, the weighted graph would then look as follows:



This weighted graph can then be used to deduce placement of queues. The following describes and motivates the different conditions on when a queue is placed:

- If the result of a definition y is used at multiple targets $z_1, \ldots, z_n$, a queue is placed. This is necessary as queues handle the dispatching.
- If the stream definition y uses an n-ary operator with $n > 1$, a queue is placed at each input x_i of y. The reason for this is that with an increasing amount of inputs, the probability for at least one of the wires of its inputs being located at a significant distance to the destination in the synthesised layout increases, which in turn also increases the time the signal needs to get from its source to its destination and thus also reducing the slack. Therefore, for this optimisation, only unary operators are considered. This limitation is also addressed in Chapter 7.
- If the stream definition y uses a unary operator with an input x of weight w_x, and $w_x < t_w \wedge w_y > t_w$ for a globally specified threshold t_w, signalling that applying y additional to the current path length of x might cause a timing violation, hence a queue is placed on the edge between x and y. The threshold to choose varies depending on the specific hardware in question.

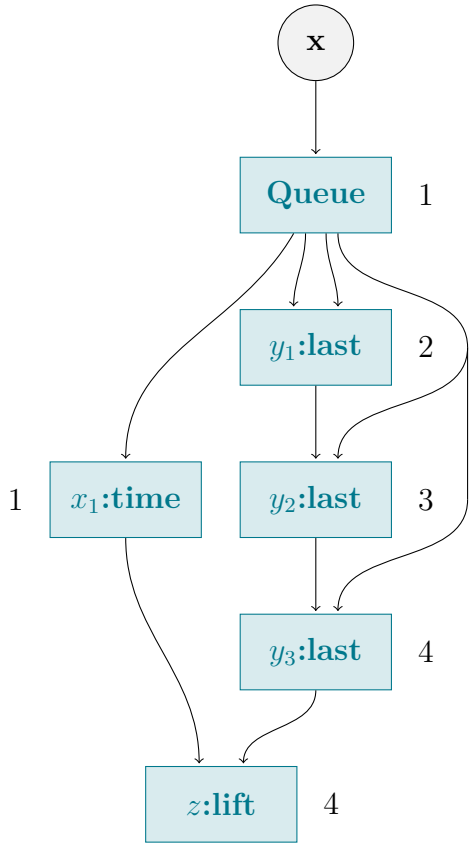
These conditions can then be applied repeatedly, recalculating the weights of affected nodes after insertion of new queues, until either no more queues can be placed none of the conditions apply any more. The result is then a specification extended with annotations about which edges require a queue to be inserted.

So far, only the placement of queues got discussed, without taking the depth of queues into account. This is the focus of the next section.

4.4.2. Depth

As motivated earlier, cases where one value is split onto multiple paths which require different amount of clock cycles to compute can significantly decrease the throughput of a specification. This can be alleviated to select a fitting queue depth, allowing to sufficiently buffer values, balancing out the difference in path lengths. This difference in path lengths can be deduced by comparing the latency of each path. Computing the latency of a path can be achieved by using the data-flow diagram, including the previously generated placements for queues. The weight of each node is then defined as the maximum of the weight of its inputs added with the latency of its associated module itself. The latencies of each module can directly be inferred from the module definitions of each module, specifically: **last** and queues have a latency of one, while **time** and **lift** have a latency of zero clock cycles.

Those weights then represent a simulated run of a data value through the system, where the weight denotes the clock cycle at which the value passes through the module. An example for such a weighting can be seen in the following figure:



As this queue depth inference only considers paths which dispatch a value to multiple paths, there will always be a queue present at that position. Given the latencies of those paths, the difference can be computed, which in this example is $\Delta = 4 - 1 = 3$. This difference denotes the depth to add to the initial queue depth of one, meaning that the optimal queue depth required here would be 4.

It is important to note that increasing the depth of a queue also drastically increases the amount of resources required to represent the queue in hardware. Thus, it would be useful for a future improvement to define a more complex mechanism to decide the depth of queues, allowing to prioritise where an increase in depth proves more beneficial. Also, using metadata on the frequency and delays of different input streams would also allow further conclusions to the required depths of queues. This is discussed further in Section 7.1.

In this chapter, we defined methods to analyse and modify *TeSSLa* specifications to allow for more performant or more resource-friendly implementations on hardware. The following chapter then highlights aspects of an implementation of this concept, including all here introduced optimisation phases.

5. Implementation

So far, the concept for a translation of *TeSSLa* specifications to a hardware description was defined, followed by multiple optimisation phases aiming to improve the result in terms of their performance and resource usage. The described translation including the optimisations have been implemented, some aspects of which are highlighted in this chapter.

5.1. Used Tools

The programming language chosen for the implementation is Scala [OSV08], which is a staticly typed, functional programming and object oriented language compiled to Java bytecode, thus also providing the platform independence of Java. The functional language features of Scala allow easy manipulations of collection data structures. Considering that the purpose of the implementation is to compile *TeSSLa* code into a hardware description, and that all described optimisation phases are relying on graph structures in some manner, this is beneficial. Another reason for this choice is the fact that the implementation of the *TeSSLa* front-end is written in Scala as well, which is used as an interface to this implementation. The input to the front-end is a *TeSSLa* specification, potentially with annotations or macro definitions, and handles parsing, type checking, macro expansion and cycle detection on the specification. The result of this is then a core language *TeSSLaCore*, which represents a flat *TeSSLa* specification with all types known. Using this core representation as input allows to branch off multiple different back-ends from *TeSSLa*, such as here the translation to Verilog.

As described in Subsection 2.2.3, we use the language Chisel to generate describe hardware modules in Scala. More specifically, the here implemented compiler describes the hardware in Chisel, and then uses the back-end provided by the Chisel framework to compile Chisel code into Verilog.

5.2. Architecture

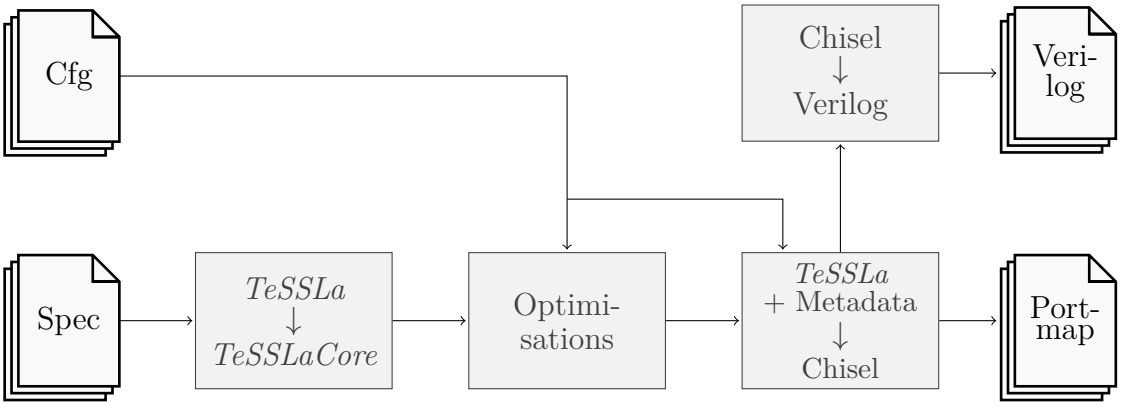


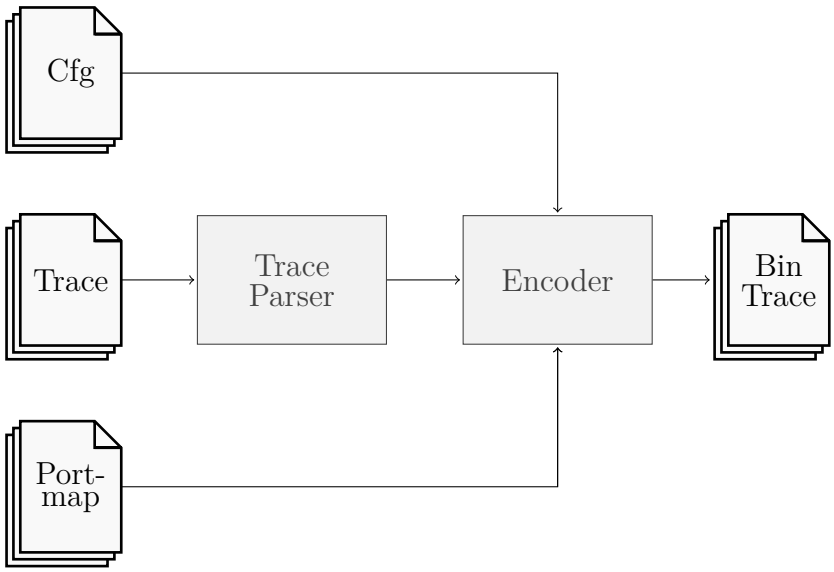
Figure 5.1.: Compiler Pipeline

This chapter gives a broad overview on the architecture of the implementation. Figure 5.1 shows an overview of the compiler pipeline. The input consists of the *TeSSLa* specification, which will first be processed into *TeSSLaCore* as described previously, and a configuration file (which is discussed in more detail in Section 5.5) containing values for every user-configurable parameter. In the next step, the optimisations described in Chapter 4 are applied, and finally the optimized result is transformed into a Chisel module. This Chisel description can then be compiled into Verilog code by using the back-end provided by the Chisel framework. Finally, there are then two results:

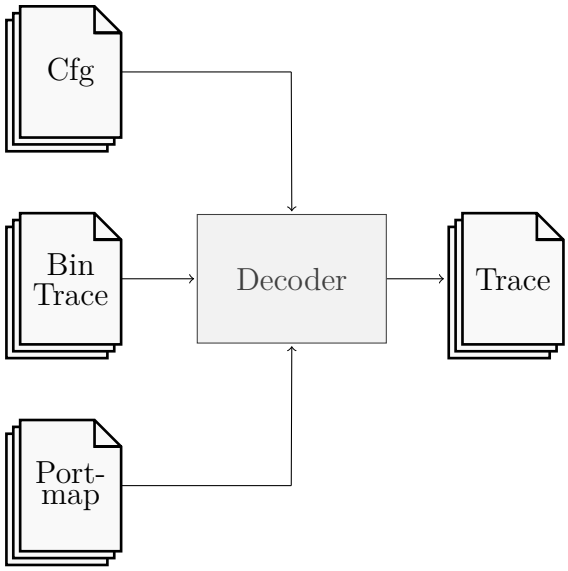
- The resulting module in Verilog
- If IO adapters are used: A mapping, assigning an address to each input and output stream of the initial specification, which is required for en-/decoding of input/output trace data. Additionally, the mapping also stores the names of streams, to restore those when decoding.

Additional to the compiler pipeline, an encoder and a decoder are implemented, which are visualised in 5.2a and 5.2b. They are used to transform a human-readable trace file into a binary format and vice-versa, as described in Subsection 3.4.1. Here, the *TeSSLa* front-end is used once again, to transform a give trace-file into an iterable data structure. In combination with the bit widths provided by the configuration file and the mapping from streams to addresses through the port map, this trace file can then be transformed into a binary trace. The supported data types are introduced in Section 5.4.

In the next sections, some aspects of the implementation are described in more detail, namely: The implementation of operators as Chisel modules, data types,



(a) Encoder



(b) Decoder

Figure 5.2.: Architecture of encoder and decoder

configuration files, and the test framework. Additionally, a full example is given to demonstrate the use of the entire toolchain.

5.3. Chisel Modules

Each atomic module is defined in Scala by using Chisel. Those modules are using type parameters allowing to instantiate those modules for different data types and widths in use. As an example, see the following – slightly simplified – definition of the **time** operator in Scala:

```
class Time[T <: Data, V <: Data] extends Module {
  val io = IO(
    val a = new ChannelInterface[T, V]()
    val out = Flipped(new ChannelInterface[T, T]())
  )
  val timeReg = RegInit(0)
  val time = Mux(io.a.valid && io.a.isTimestamp,
                 io.a.timestamp, timeReg)
  timeReg := time

  io.a.ready := io.out.ready
  io.out.valid := io.a.valid
  io.out.isTimestamp := io.a.isTimestamp
  io.out.value := time
  io.out.timestamp := time
}
```

It can be seen that for most parts, the implementation of a module in Chisel can be directly deduced from the pseudo-code definitions used previously. The type parameters `T` and `V` represent the type for timestamps and data values to be used. The IO ports can then be defined calling `IO`. The `ChannelInterface` in this case implements the modified ready-valid interface used, by defining the signals and their direction:

```
class ChannelInterface[+T <: Data, +V <: Data] extends Bundle {
  val ready = Output(Bool())
  val valid = Input(Bool())
  val isTimestamp = Input(Bool())
  val value = Input(dataType.tpe)
  val timestamp = Input(timeType.tpe)
}
```

As Chisel is a framework for Scala, all features of Scala can freely be used to modify and connect Chisel constructs, for example by using type parameters to easily define hardware modules for varying data types or bit widths. Another aspect where this

is useful is when defining an aggregate signal, as in this code extract from the output adapter:

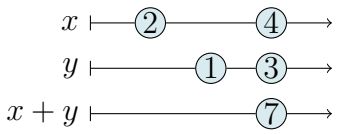
```
val minTime = io.a.map(_.timestamp).reduce[T]{
  case (a,b) => Mux(a <= b, a, b)
}
```

In this case, the Collection API of Scala is used to take the `timestamp` flag of each port of `a` (which is a vector of ports), and aggregating them to a signal returning the smallest of all timestamps.

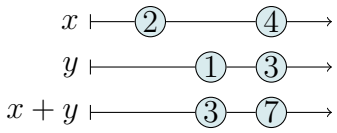
Another aspect to note is that the *TeSSLa* front-end also uses a standard library, which also allows defining additional built-in operators. This allows to extend *TeSSLa* by more operators and also define specific Chisel modules for those operators, allowing for some case specific modules and hand-crafted optimisations compared to defining them as a combination of the existing *TeSSLa* operators.

The implementation features multiple custom built-in operators, some of which are of greater importance and are thus described here:

SignalLift As discussed in Subsection 2.1.3, an operation `x + y` can be interpreted to be lifted in different ways. In the previous chapters, such an operation was interpreted as a total function, such that both inputs are required to be present at that timestamp for an output to be produced:



However, especially for arithmetic or logic operators, it may be wanted to interpret those inputs as signals rather than events, thus not requiring them to occur at the same timestamp, but rather taking the most recent value:



Using the base *TeSSLa* operators, such a *signal lift* can be defined as follows:

$$\begin{aligned} aa &:= merge(a, last(a, b)) \\ bb &:= merge(b, last(b, a)) \\ slift(f)(a, b) &:= lift(f)(aa, bb) \end{aligned}$$

where *aa* and *bb* take the current value of their respective stream if it exists, or the last seen value otherwise. However, as this would require the use of 3

lift modules and 2 **last** modules, this would result in a significant amount of overhead. Instead, we use a dedicated module for this purpose. This module is very similar to the **lift** module, the difference being that it includes registers for each input to store the last seen value, and use that value of no value exists for the current timestamp.

Fold A folding functionality is well known from functional programming languages, allowing to iterate over a sequence and applying a function on the previous value and the current input, generating the next value and thus aggregating the entire sequence. In the context of timed streams, folding a function $f : A \times A \to A$ over a stream $x = t_0\ d_0\ t_1\ d_1 \ldots$ would result in $x' = t_0\ f(0, d_0)\ t_1\ f(f(0, d_0), d_1) \ldots$ for a starting value 0. Such a stream can also be defined recursively, however, as a recursive definition always uses at least one **last** their output is delayed by one clock cycle, which may cause its dependencies to stall depending on the situation. Simple definitions like a counter should instead rather be defined by using a built-in fold operator and its according module, which uses a register to store the last result and applies an arbitrary function to the stored value and the input.

As previously shown, the Chisel modules support different data types by use of type parameters. In the next section, the different supported data types which can be used with those modules are introduced.

5.4. Data Types

Since so far, only a generalised encoding for traces has been given, this section introduces the supported data types as well as their encodings to a binary format. In particular, this binary format is also used by the encoder and decoder to transform traces into the correct binary representation and vice-versa. The primitive data types supported are *integer*, *boolean* and *unit*. Integers are encoded using the two's complement representation in case of being signed, and the natural binary representation for unsigned integers. The *TeSSLa* front-end also allows the definition and use of annotations to attach meta-data to definitions. This is used here to declare the bit width and the sign of an integer input. As an example, the following definition:

```
@unsigned
@width(8)
in x: Events[Int]
```

declares x as an 8 bit unsigned integer. If no bit width is given for an integer stream, a default integer bit width is applied, which is given in the configuration file (see Section 5.5). This information can then also be used by the bit width inference.

The primitive type *unit* represents the data domain $\mathbb{U} = \{\square\}$ used in **unit** representing a single element. As there is no distinction of values to be made here, the value itself would not require any bits to be encoded, resulting in a bit width of zero. This does not mean that the entire communication channel can be omitted, as each value is wrapped in a message using the ready-valid interface, and the value only being one part of it. However, Chisel does not allow a bit width of zero for any wire, thus a bit width of 1 bit for **unit** is used, where the value of the bit is ignored.

Additionally, in the previous chapters we made use of tuples as well as values which might be undefined ($\bot$), which therefore need to be supported as well. To permit definition of more complex interfaces, Chisel provides the `Bundle` class, which acts as an aggregation of multiple other data types, while per default its binary representation is defined as the concatenation of the binary representation of its fields. A data type for optional values would require two fields: A single bit flag signalising whether or not the value is set, and a field for the value itself. The latter requires the interface to have a type parameter, as any supported value type should be allowed to be used with an optional as well. The following shows a simplified variant of the Chisel code for this interface:

```
class OptionInterface[V <: Data] extends Bundle {
  val value: V
  val defined: Bool
}
```

A tuple can be defined by using one field for each value of the tuple, each with their respective type parameter:

```
class Tuple2Interface[A <: Data, B <: Data]
                      extends Bundle {
  val a: A
  val b: B
}
```

To avoid having to define separate interfaces for different tuple arities, n-ary tuples are represented as nested binary tuples $(x_1, \ldots, x_n) = (x_1, (x_2, (\ldots, (x_{n-1}, x_n) \ldots)))$.

5.5. Configurations

As some aspects of the synthesis are parametrised (as for example the previously described default bit width for integer data types) and can thus be configured depending on the users needs, a configuration file is used for an intuitive way of defining those parameters. The existing parameters are as follows:

- Bit widths of address, data and the total width of the input and output channel. Missing widths are being inferred from given ones, if possible.
- Bit widths for the integer type and timestamps, which are mandatory.
- The threshold for spacing of queues which was introduced in Section 4.4 and a maximum depth for queues, which is used to limit queues in their size if needed.
- The strategy to use by the bit width inference. The algorithm described in Section 4.3 can be instantiated in multiple different ways. The strategies used differ in the way that on one of them, the bit width inference takes overflows into accounts (for example, adding two 32 bit values resulting in a 33 bit value), while the second strategy ignores those overflows.
- Flags whether or not the input-adapter or the output-adapter should be used.

As format for the configuration files, both JSON and YAML are supported, as both allow an intuitive, human-readable definition of data structures. Additionally, the format is being validated against a schema file to ensure proper use. An example for a configuration file can be seen in Section 5.6.

5.6. Example

We now show an example on how the toolchain can be used by use of an example. Given the following specification file `spec.tessla`:

```
in x: Events[Int]
in y: Events[Int]

def sum = fold(filter(x, x == y), 0,
               (a: Int, b: Int) => a + 1)
def total = fold(x, 0, (a: Int, b: Int) => a + 1)
out sum
out total
```

which returns the amount of events on x where $x == y$ and the total amount of events on x. Also, the following configuration file `config.yaml` is used:

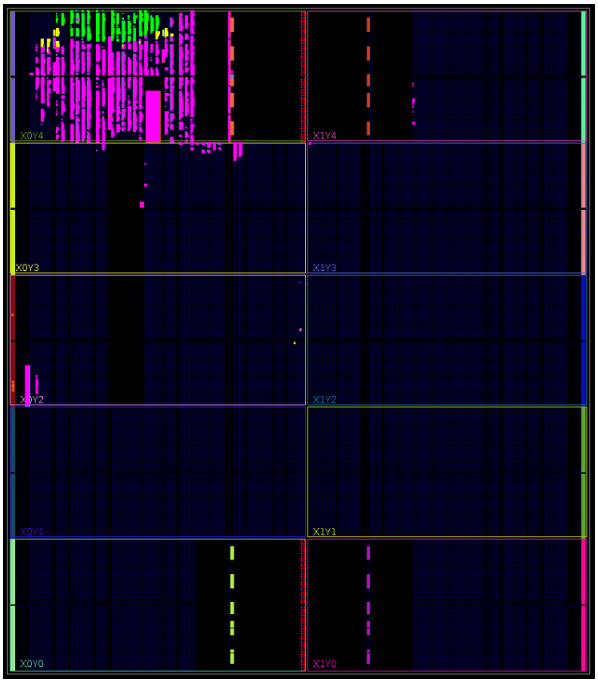
```
widths:
  in: 32
  out: 32
  integer: 20
  timestamp: 20
queueSize: 4
queueSpacing: 10
bwiStrategy: dynamic
inputAdapter: true
outputAdapter: true
```

Here you can see the input and output width each being defined as 32 bit. Given that the specification uses 2 streams as input and output each, the required bit width for addresses is 1. With this, the maximum allowed with for data types of 31 bits can be inferred. Additionally, the bit widths for the integer type and timestamps are both set to 20 bits. The queue size defines the maximum depth of a queue, here being 4, while the queue spacing defines the threshold w_t defined in Subsection 4.4.1. The bit width inference strategy is set to *dynamic*, denoting the mode which adapts to overflows. Furthermore, both adapters are enabled.

Then, the Verilog code can be generated by calling the synthesis toolchain of the implementation as follows:

```
java -jar tessla-synthesis.jar synthesize
          --config config.yaml
          --file out.v
          --mapping ports.map
          tessla.spec
```

This generates the previously described port mapping for the encoder/decoder and stores it in `ports.map`, and the resulting Verilog code in `out.v`. The Verilog code can then be used in combination with a synthesis tool like Xilinx Vivado to synthesise the specification on an FPGA. To define the IO between the system and the FPGA, Xillybus is used, which is an FPGA IP core using PCIe. This is described in more detail in Chapter 6. The synthesised layout is visualised in the following figure:



The used resources are represented using three colors, where pink is the Xillybus module, yellow being the used FIFO queues as connection between the Xillybus and the generated specification, and green being the generated specification.

Then, an input trace file `trace.txt` which looks as follows:

```
1: x = 5
1: y = 3
2: x = 7
2: y = 7
4: x = 8
5: x = 7
```

can be encoded using the previously defined configuration file and the generated mapping:

```
java -jar tessla-synthesis.jar encode
          --config config.yaml
          --bin trace.bin
          --map ports.map
          trace.txt
```

This generates a binary representation of that trace file in `trace.bin`, which can then be used as input to the input adapter of the specification module, and the

output generated by the module stored in a binary file. This resulting binary file can then be decoded in a similar manner:

```
java -jar tessla-synthesis.jar encode
          --config config.yaml
          --bin out.bin
          --map ports.map
          out.txt
```

storing a human readable formatting of the result in `out.txt`:

```
0: total = 0
0: sum = 0
1: total = 1
2: total = 2
2: sum = 1
4: total = 3
5: total = 4
5: sum = 2
```

5.7. Testing

In the following, the different tested aspects of the implementation are described. The test framework for the hardware is mainly split into two parts: Unit tests, which test for correct functionality of each atomic module in different use cases, and integration tests, which test the functionality of the compiler pipeline and its consistency to the software solution.

Chisel provides the class `PeekPokeTester`, allowing to stimulate modules and simulate their behaviour step by step, thus allowing to test their functionality directly in Scala. This allows to define unit and integration tests, to simulate the behaviour of atomic modules as well as entire specifications directly within the project itself, resulting in a very intuitive test framework.

The `PeekPokeTester` is attached to an instance of a module and provides (amongst others) functionality to

- Set the values of input pins with *poke*
- Read the output pins with *poke*
- Assert properties to an output and failing with some message if the assertion failed, with *expect*

- Progress the computation by n clock cycles through *step(n)*.

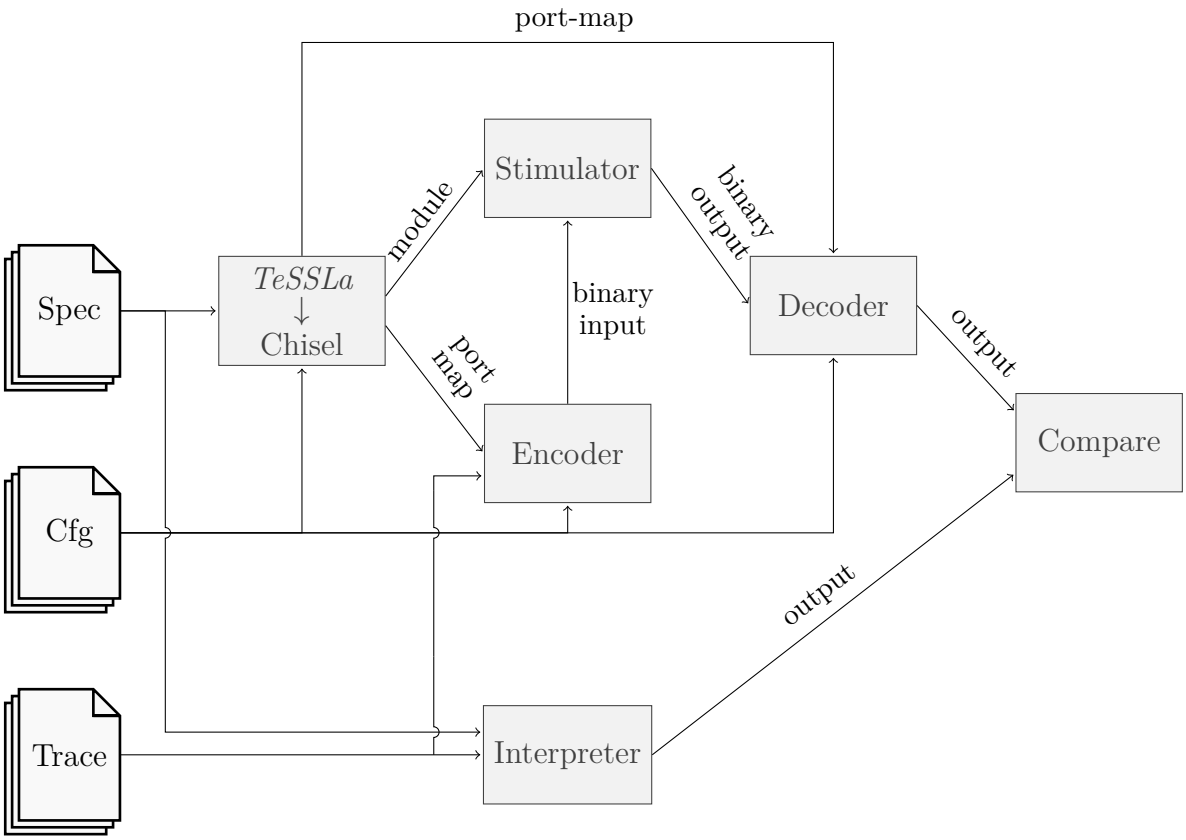
Other components like the encoder, decoder and the parser for configuration files are tested using unit tests as well.

The unit tests stimulate the atomic modules directly, to specifically test their behaviour in certain use cases. Here, the `PeekPokeTester` writes and reads values directly on the channels used by that module.

The integration tests use the entire compilation pipeline. Specifically, each test case is given a specification, a configuration and an input trace file, which are then used to generate the Chisel module, encode the input and simulate the module given the input using a `PeekPokeTester` driving the adapters. The resulting output is then decoded and compared against the result of the software *TeSSLa* backend, which provides an interpreter for *TeSSLa* specifications. The decoded trace can then be compared to the trace generated by the interpreter, which also ensures consistency between both tools.

This separation of testing components locally as units and performing integration tests allows for easier debugging and localisation of errors. Especially in early phases of the implementation, the unit tests are useful to cover the basic expected functionality of that component, and can then potentially be extended by test cases which were initially missing if a failing integration test revealed an uncovered case.

A structural overview of the integration tests is given in the following figure:


port-map
Stimulator
module
binary
output
Spec
TeSSLa
↓
Chisel
binary
input
Decoder
output
port
map
Encoder
Compare
Cfg
output
Trace
Interpreter

6. Evaluation

The purpose of this chapter is the evaluation of the implementation introduced in the previous chapter. As the different optimisation phases are a major aspect in this thesis, the evaluation of those is the main focus of this chapter. In the following sections, the performance and resource usage of different specifications is measured and evaluated.

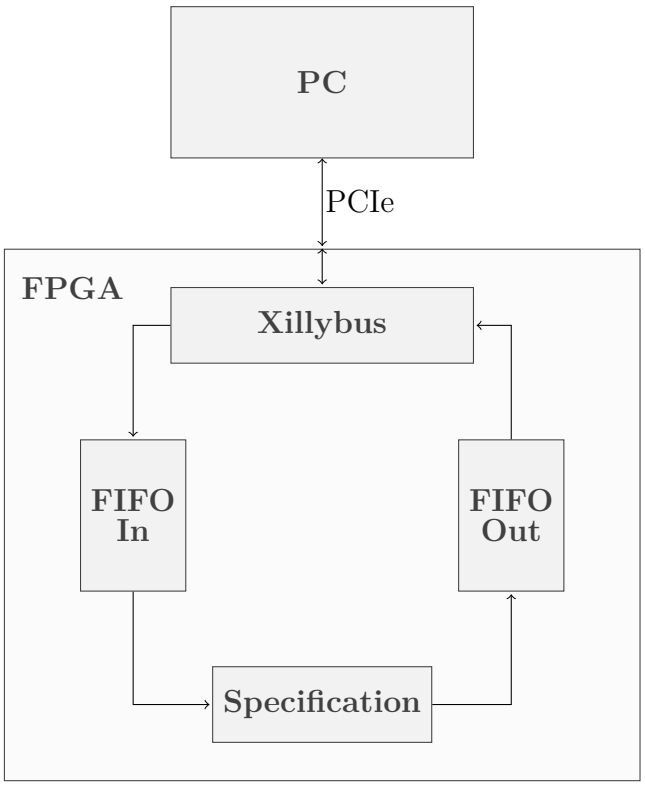
6.1. Performance: Setup

In this section, the hardware setup used for the performance measurements is introduced. The FPGA used is the XC7A200T-FBG676 of the Artix-7 series and operated at a clock frequency of 125 MHz, on the Artix-7 AC701 evaluation board. The generated Verilog code is synthesised onto the hardware by using Xilinx Vivado 2018.2. Furthermore, there are multiple possible use cases for such a hardware, which naturally would also result in different benchmarks and measurements. The use case selected here is the one of a logfile analysis, which has the advantage that the previously introduced adapters, formats and encoders already provide all the tools required to set this up. The specifications of the system are as follows:

OS	Ubuntu 18.04
Motherboard	Gigabyte B85M-D3H
Disk	SanDisk SD8SBAT1
Memory	2x Hynix HMT351U6CFR8C-PB
Processor	Intel Celeron G1840

The data transfer from host to FPGA and vice-versa is performed over PCIe and handled by use of Xillybus[1], acting as interface between for example a FIFO and a PCIe hardware block, thus allowing the use of simple constructs like FIFOs to define communication of the application logic. The communication over PCIe to the generated module is visualised here:

[1] http://www.xillybus.com/



However, the here presented use case comes with a certain amount of overhead for their IO operations, skewing the benchmarks. This is a significant drawback when it comes to accurately measure the effect of the here described optimisation phases. Therefore, to evaluate the performance of optimisations, another, more artificial setup is used: Instead of transferring data to and from the FPGA, the input events are generated and consumed directly on the hardware itself. Assuming that both of those end points finish their computation within one clock cycle, this does not generate any additional overhead, thus the resulting calculation time is then exactly the processing time of the specification on the hardware.

Additionally, to have the performance measurement as accurate as possible, a benchmarking module is used. The following describes that module, as well as the required modules to generate and consume events on the hardware directly.

Benchmark

The benchmarking module here is used as a wrapper around the specification, providing the same in- and out-going ports, simply forwarding the values going into and coming from the specification. Any further generated metrics by the benchmarking

module can then be emitted through an additional output port. This is visualized in Figure 6.1.

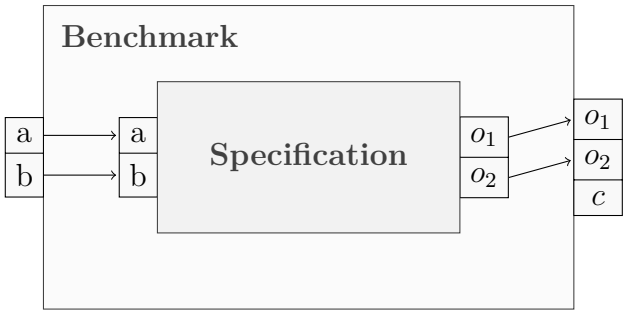


Figure 6.1.: A specification module with two inputs and outputs, wrapped by a benchmarking module

The benchmark should only count the clock cycles between input of the first event, and output of the last event. As information when all events are processed, the highest possible timestamp for the provided bit width is used. Additionally, the benchmarking results should only be emitted at the very end of the run to reduce the amount of overhead the benchmark poses to the output channel. This means that, after occurrence of the maximum timestamp at all output ports of the specification, it can be assumed that no further events will arrive. This assumption only holds if the actual timestamp does not naturally increase up to the flushing value, which would indicate an overflow of the timestamp for the next input. This is most likely an unwanted behaviour for timestamps and would thus only occur when choosing an unfitting bit width for that use case. Therefore, excluding the just noted case, the benchmark results can be emitted after the maximum timestamp occurred. A simplified pseudo-code for this module can be seen in Listing A.4.

Event Generator

The main requirement of the event generator is to be able to generate new data for every input within one clock cycle, such that there is always a valid data present whenever the specification module is ready to process the next input. The generated data of course has to be a valid sequence of timestamps and data values. Additionally, as the behaviour of specifications may also heavily vary depending on the values inserted, using a simple counter or even a constant value to emit as event value is unfavourable. Instead, the value should preferably be randomised in a simple enough way such that it still computes within a single clock cycle.

For the randomisation, the generator uses a pseudo-random number generator. The here chosen algorithm is a 32-bit XORshift random number generator described in

[Mar03]. While only providing a rather small period of $2^{32} - 1$, it can easily be implemented in hardware and only requires one clock cycle to calculate. This is considered sufficient as true randomness is not required for this use. Furthermore, the event generator takes a seed as input which is used as initial seed for the random number generator, allowing to reproduce a specific run if needed.

The event generator can then be defined as a module parametrised by n, denoting the amount of output ports. For each of those ports it then contains registers to store the timestamp and value to emit for this output. Note that in this case, the input is not synchronised on their timestamps as it would be when using an input adapter, hence the requirement that each output has their own timestamp register. To ensure proper alternation of timestamps and values, each output also requires a toggle to decide if a timestamp or a value should be currently sent. The event generator is being initialised by two values, one being the amount of events to generate, the second one being the seed for the random number generator. The pseudo code for this module can be seen in Listing A.5.

Output Sink

The final component to define is the output sink, whose role is to reduce the amount of output generated. It is important to note here that simply discarding all outgoing events would most likely be detected by optimisation phases of Vivado during the synthesis. Thus it is required that the computation still has an effect on the output, but without emitting any events during the actual computation. The approach here is very similar to the idea used on the benchmarking module, which emits its measurements after all events were processed. Similarly, the output sink contains a single register, aggregating all incoming data values in this register, and finally emit a single aggregate value after the run completed.

IO-Decoupled Setup

Using the previously defined modules then allows the construction of a benchmarking environment which is completely decoupled from any IO overhead during the calculation process: The only IO performed is the configuration for the event generator which takes place before the calculation process begins, and the output of the aggregated result and benchmarking values which comes after the calculation. The full setup of the modules can be seen in Figure 6.2. This setup can then in the following measurements be used to exactly measure how many clock cycles a certain specification required, and compare those results against those of their optimised counterpart.

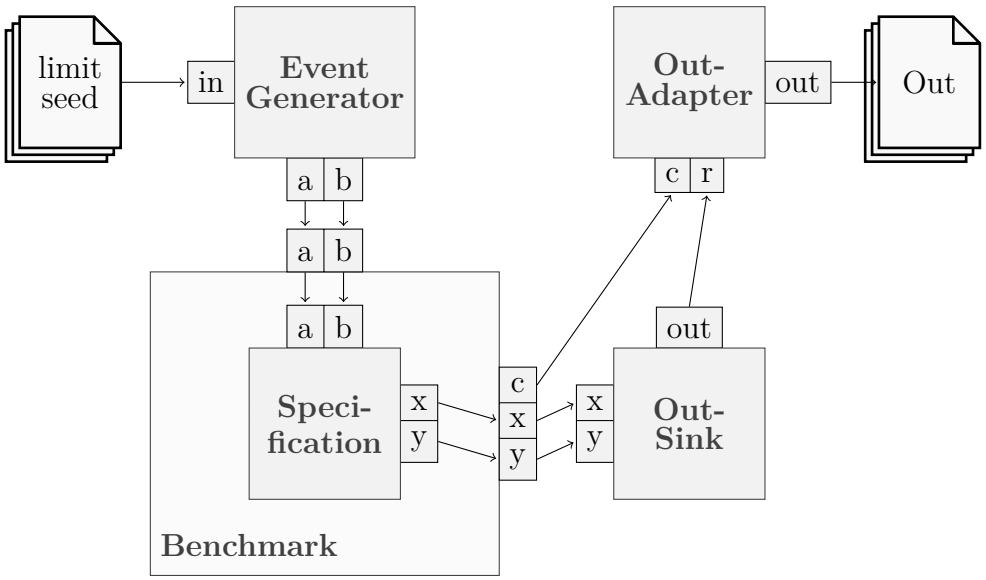


Figure 6.2.: IO decoupled benchmarking setup.

6.1.1. Metrics

The measurement of clock cycles needed for the computation can be used to calculate the throughput. Additionally, using the slack of a layout, a maximum clock frequency can be calculated as well. The operating frequency for which the layouts are generated is 125 MHz, and the data bit width used is 32 bit.

For the measurements where the data is read from and written to a disk, the throughput is a useful metric to take into account. If the specification only has a single input stream, the incoming data is an alternating sequence of timestamps and values, thus consisting of 64 bits per event. In that case, the throughput can be calculated as:

$$TP = 64 \cdot n \text{ bit } \cdot \frac{125\,\text{MHz}}{c}$$

where c is the amount of clock cycles the computation needed.

The slack of different paths in a circuit has an effect on the frequency a specification can be run at, which also has a direct effect on their throughput. A specification may require relatively few clock cycles to compute, but not allow for much higher clock frequencies, while another specification may be the exact opposite in those aspects. Therefore, additionally to the measured throughput on the operating frequency, it may also be useful to compare the maximum throughput of a specification. Having a positive slack s effectively means that this specific path could be slowed down by at most s without compromising the functionality. Hence, the minimum over all slacks within the circuit gives us the amount of time the period can be shortened.

The minimum over all slacks is shown by Vivado as worst negative slack *WNS*. The maximum clock frequency[2] is then

$$f_{max} = 1/(T - WNS)$$

where T denotes the period used for the elaboration of the circuit with $T = \frac{1}{f}$. So in this case, $T = \frac{1}{125\,\mathrm{MHz}} = 8\,\mathrm{ns}$.

This maximum frequency can then also be used with the previously calculated throughput TP to extrapolate the maximal throughput:

$$TP_{max} = TP \cdot \frac{f_{max}}{125\,\mathrm{MHz}}$$

6.2. Performance: Measurements

After introducing the setup and metrics used for the different performance benchmarks, this section now shows each benchmarked scenario and specification followed by an evaluation of its result. The benchmarks are split into three different categories with different focuses:

Effect of IO uses a logfile analysis setup as previously described and aims to measure the impact of IO in this setup on the total calculation time.

Adapters evaluates the effect of adapters in such a logfile analysis setup, and how this effect is affected depending on different traces.

Optimisation uses the second described setup, which generates and consumes events directly on the FPGA, to measure the speedup reached by the optimisations.

6.2.1. Effect of IO

To be able to compare the effect of IO, the selected specifications are run in multiple different configurations, using the both the logfile analysis setup as well as the setup using a generator and sink. Additionally, hybrids of both setups are considered as well to get more fine-grained results.

[2] `https://www.xilinx.com/support/answers/57304.html`

Passthrough

The first specification considered here simply forwards a single input to its output, thus maximising the effect of the IO on the total calculation time. The *TeSSLa* specification looks as follows:

```
in x: Events[Int]
out x
```

Events	EvG&S	S	EvG	None
10 000	20 000	22 480	23 750	24 805
100 000	200 000	225 052	488 565	439 708
1 000 000	2 000 000	2 308 843	3 852 961	3 672 851
10 000 000	20 000 000	25 496 958	40 181 087	45 149 915
100 000 000	200 000 000	269 894 934	409 096 101	474 899 314

Figure 6.3.: Benchmarking of **Passthrough** in # clock cycles

The measurement results of this specification can be seen in Figure 6.3, where the first column states the amount of events used, while the other columns denote all different combinations for the IO configurations by stating which modules were used (*EvG* for *Event Generator*, *S* for *Output Sink*). As you can see, all different scenarios scale proportional to the amount of events used.

To get a proper comparison of those results, we can make use of the fact that each event requires at least two clock cycles to be processed, which means that the minimal processing time for n input events is $2n$ clock cycles. This optimal processing time can then be used to calculate the ratio of the measured processing times to it. This is shown in Figure 6.4, where the results for different event counts are aggregated by averaging them.

EvG&S	S	EvG	None
0%	20.5%	92.2%	98.1%

Figure 6.4.: Slow down of **Passthrough** caused by IO.

The configuration using both the sink and generator has the optimal processing time. This is to be expected as the specification itself is only passing the events through, which should not require any additional clock cycles. Furthermore, observing the two next results shows that transferring the result onto the disk has a significantly higher impact on the performance than reading the input from the disk. Considering the result of the last configuration and the fact that this *TeSSLa* specification performed no calculations, it can be assumed that the effect of the IO will not be higher

than double the optimal processing time for this setup, under the assumption that the adapters do not act as a bottleneck. This case will be further analysed in a later section. Another aspect to note especially for lower input sizes is that, when transferring data from the system to the hardware, it is buffered. On lower input sizes the size of the buffer is sufficiently high, such that only one block of data needs to be transferred.

Malloc

This specification is used to provide a more realistic evaluation of how the IO affects the overall calculation time, as the first specification only required one clock cycle to compute. The specification is as follows:

```
in mallocAddress: Events[Int]
in freeAddress: Events[Int]

def t = merge(mallocAddress, freeAddress)

def lookup(key: Events[Int], f: Events[Bool],
                             size: Int): Events[Int] = {
  def l: Events[Option[Int]] = last(reg, key)

  def add = f && isNone(l)
  def remove = !f && isSome(l) && getSome(l) == key

  def reg = merge(if add then Some(key) else
                  if remove then None[Int] else l, None[Int])

  static if size == 0 then 0 else
  lookup(key,
         f && !(add || remove), size - 1) +
           if isSome(reg) then 1 else 0
}

def allocated = lookup(t, time(mallocAddress) == time(t), 3)

out allocated
```

Here, two inputs are taken which describe a memory address which should be allocated or freed. The specification then returns how many memory addresses are allocated. This is implemented by modelling a lookup table using a chain of recursive definitions, which each represent one entry in the lookup table. The measurements

for this specification can then be seen in Figure 6.5. As this specification uses two input streams, the inputs used for this measurement had one event per timestamp, alternately.

Events	EvG&S	S	EvG	None
10 000	130 034	130 034	130 034	130 030
100 000	1 300 034	1 300 034	1 452 057	1 414 866
1 000 000	13 000 034	13 011 304	14 876 864	15 034 011
10 000 000	130 000 034	144 601 915	153 647 952	155 025 566
100 000 000	1 300 000 034	1 484 627 024	1 560 098 617	1 575 619 414

Figure 6.5.: Benchmarking of **Malloc** in # clock cycles

It can easily be seen that when increasing the input size, the effect of the IO increases significantly less compared to the previous example. Furthermore, the difference between using both the event generator and the output sink to using none averages here only 13% as shown in Figure 6.6, while the previous example averaged at around 98%. In conclusion, it can be assumed that for more complex specifications, the IO has a rather small affect on the total performance.

EvG&S	S	EvG	None
0%	5.1%	12.9%	13%

Figure 6.6.: Benchmarking of **Malloc**

6.2.2. Adapters

Considering that adapters take or produce an aggregated, synchronised stream of inputs, they can be a severe bottleneck to the specification. Processing n events at a specific timestamp will require at least $n + 1$ clock cycles. This means that if there is more than one event occurring at a single timestamp, the adapters might act as a bottleneck. In particular, this also means that if there are more output events generated than there are input inserted, the output adapter may slow down the process even further. How strong of a bottleneck they effectively are is not only dependent of the specification, but also highly dependent on the nature of the input trace.

To demonstrate this, the following specification is used and has its processing time measured with different traces as inputs:

```
in x: Events[Int]
in y: Events[Int]
in z: Events[Int]

out x
out x + 5
out x + y + z
```

For the following deductions, it is assumed that all streams have already been initialised. The occurrence of a new event on x would cause a new event on four output streams, while a new event on y or z would generate a new event on only one of them. Additionally, each of the paths can compute within a single clock cycle, which simplifies deductions about the bottleneck of IO adapters. Therefore, it would be expected to see the following effects depending on the input:

- If at one timestamp only x has an event, the input adapter does not act as a bottleneck and only requires 2 clock cycles. However, the output adapter has to emit three events, thus needing 4 clock cycles.
- If both y and z have an event, the output adapter will only require two clock cycles, while the input adapter requires three.

There are four different input traces used for measurement, where each of them initialises all streams at timestamp 0:

- The first input trace then has one million timestamps with events only on x.
- The second one has one million timestamps with events only on y.
- The third has one million timestamps with events on both y and z.
- And the final input has one million timestamps with events on all three input streams.

The results of those measurements can be seen in Figure 6.7. For the first input trace, it was expected to be the least performant as the occurrence of only events on x causes a difference of 2 clock cycles between input and output adapters per timestamp. For the second and the fourth input, both adapters require the same amount of clock cycles, thus being the most performant. The third specification causes the input adapter to require 1 clock cycle more per timestamp than the output adapter, therefore ranging in between the other results. This shows that the effect of adapters on the performance of a specification only indirectly depends on how many input and output streams are used, but rather how the streams relate to each other and how often they share events on the same timestamp.

	x	y	y, z	x, y, z
# Clk.	8 379 172	4 496 139	5 306 051	8 763 913
# Data	2 000 004	2 000 004	3 000 004	4 000 004
TP	954.8 Mbit/s	1779 Mbit/s	2262 Mbit/s	1826 Mbit/s

Figure 6.7.: Measurement results for adapter usage

6.2.3. Optimisation

This section focuses on comparing the performance of specifications optimised through the procedures introduced in Chapter 4 with their non-optimised counterparts. As the goal here is to evaluate the impact of the optimisation, the setup with an event generator and output sink is used.

Factorial

This example serves to demonstrate the potential advantage the optimisation phases can provide. The following specification calculates $\frac{n!}{n-k}$ for a fixed k and an input stream providing n:

```
in x: Events[Int]

def factorial(n: Events[Int], k: Int):Events[Int] = {
    static if (k == 0) then 1 else n * factorial(n-1, k - 1)
}

out factorial(x, 10)
```

Listing 6.1: *TeSSLa* code for the factorial example

This specification leaves a lot of room for optimisation, mainly because of two reasons:

- There are a lot of operations performed on inputs with the same logical timing, which can then be merged.
- The terms are aggregated in a chain instead of a balanced manner, thus resulting in long paths with stalls after each calculation step.

The stream merging phase introduced earlier should be able to merge all those terms into a chain of unary expressions, which can then be fully pipelined and should thus result in the minimal amount of clock cycles required (apart from an initial latency to fill the pipeline).

# Events	No Opt.		Opt.		Speedup
	# Clk.	Throughput	# Clk.	Throughput	
10 000	200 020	400	20 020	3 996	10
100 000	2 000 020	400	200 020	4 000	10
1 000 000	20 000 020	400	2 000 020	4 000	10
10 000 000	200 000 020	400	20 000 020	4 000	10
100 000 000	2 000 000 020	400	200 000 020	4 000	10

Figure 6.8.: Results of the factorial example, with throughput given in Mbit/s

The measurements are shown in Figure 6.8. Note that the throughput here only denotes the amount of data the hardware would be able to process per second, while ignoring all costs of IO operations, thus diverging from the throughput in a real scenario. Both variants have a latency of 19 clock cycles, meaning that passing a timestamp followed by a value through will require 20 clock cycles. However, the unoptimised specification has a dependency from the very first to the last module, thus stalling throughout the entire chain. The optimised version on the other hand is fully pipelined and can thus emit a new result every clock cycle, which effectively leads to a speedup of factor 10 in this case.

	No Opt.	Opt.
WNS	0.192 ns	0.136 ns
f_{max}	128.1 MHz	127.2 MHz
TP_{max}	410 Mbit/s	4069 Mbit/s

Figure 6.9.: Maximum clock frequency and throughput of the factorial example.

As shown in Figure 6.9, the optimised version resulted in a slightly worse slack, hence the lower maximum frequency.

Malloc

Here, the memory allocation example, which was already used in evaluating the IO, is reused. However, this time we measure the effect of the optimisation instead of focusing on the IO aspect. As a chain of recursive definitions, this specification cannot be fully pipelined as the previous example was, but nonetheless some streams can be merged here as well, and some of the stalls can be alleviated by according placement and depth of queues. The specification used is the same as previously used in Subsection 6.2.1, and similar to the previous example, the specification is benchmarked with and without optimisation, with the results shown in Figure 6.10 and Figure 6.11.

# Events	No Opt.		Opt.		Speedup
	# Clk.	Throughput	# Clk.	Throughput	
10 000	130 034	615,2	80 021	999,7	1.625
100 000	1 300 034	615,4	800 021	1 000	1.625
1 000 000	13 000 034	615,4	8 000 021	1 000	1.625
10 000 000	130 000 034	615,4	80 000 021	1 000	1.625
100 000 000	1 300 000 034	615,4	800 000 021	1 000	1.625

Figure 6.10.: Results of the malloc example, with throughput given in Mbit/s

	No Opt.	Opt.
WNS	1.022 ns	0.107 ns
f_{max}	143.3 MHz	126.7 MHz
TP_{max}	705.5 Mbit/s	1013 Mbit/s

Figure 6.11.: Maximum clock frequency and throughput of the malloc example.

As you can see here, the optimisation provides a speedup of 62.5% and requires on average 8 clock cycles per event. The main issue here is that the specification used is a chain of three recursive definitions. A recursive definition always consists of at least one **last** and one queue, therefore requiring a minimum of two clock cycles each. Additionally, the value `key` used in the specification is used as input for each recursion, thus requiring to be passed to multiple different paths with very differing lengths. This was attempted to alleviate with choosing a fitting queue depth in Subsection 3.2.1. However, it was not possible to synthesise the specification with a queue of sufficient size due the layout resulting in a negative slack. Therefore, the majority of this specification could not be pipelined.

A possibility to use further improve this result is by replacing the recursions with fold operations, or to try and reduce the amount of queues needed inside the recursion by merging outgoing paths into one, both of which are discussed in Chapter 7.

6.3. Area

This section examines the amount of resources the resulting specifications take up on the FPGA as shown by Xilinx Vivado. While the bit width inference introduced in Chapter 4 attempts to reduce the required resources, the placement of queues, depending on their spacing and depth, can have a severe impact on the space usage of an optimised specification compared to their non-optimised counterpart. Therefore, this evaluation compares the used resources for each specification, evaluating the op-

timised against the non-optimised variant. The used area is measured by considering the amount of lookup tables and registers needed by the implementation.

6.3.1. Bit Width Inference

The first test aims to isolate the effect of the bit width inference from the other optimisations. Furthermore, remember the bit width inference also permits including knowledge about integer value ranges for streams using the newly added annotations `@width`, which allows for further reductions. Therefore, the here used specification is compared using three optimised variants, where one uses all optimisations but the bit width inference, the second one using all optimisations, and the third one using all optimisations with additional width information.

The specification used here is the following:

```
in value: Events[Int]
in dispatch: Events[Int]

in x: Events[Int]

def tupled[T, U](ev: Events[T], u: U) :=
      lift1(ev, (a: Option[T]) => Some((getSome(a), u)))
def distribute(n: Int) ={
  def v := filter(value, dispatch == n)
  tupled(default(maximum(((v >> 11) * 100) & 511), 0), n)
}

def getMin(from: Int, to: Int) = {
    def rec(from: Int, to: Int): Events[(Int, Int)] = {
        def diff := to - from + 1
        static if diff > 1 then {
            def lhs := rec(from, from + diff/2 - 1)
            def rhs := rec(from + diff/2, to)
            slift(lhs, rhs, (a: (Int, Int), b: (Int, Int)) =>
              if a._1 <= b._1 then a else b)
        } else distribute(from)
    }
    lift1(rec(from, to),
      (a: Option[(Int, Int)]) => Some(getSome(a)._2))
}
```

```
out x * getMin(0, 31)
```

Listing 6.2: Example: Dispatcher maximum

Here, the input of stream *value* is segragated onto 32 different paths dependent on the value of *dispatch*. Then, the values are manipulated and the maximum for each of those partitions is taken which then are aggregated with *getMin*, returning the index of the partition which currently holds the global minimum of all the local maxima. Then, another stream x is multiplied by that index.

Additionally, the bit widths were chosen as follows:

```
widths:
  in: 32
  out: 32
  integer: 25
  timestamp: 27
```

The first result is still using all optimisations except the bit width inference, so that all differences between both results solely are due to this phase. Assuming that the value of *dispatch* is only within the range used here, it can be limited to a 5 bit unsigned integer. Furthermore, it is assumed that *value* is a 10 bit unsigned integer. The results shown in Figure 6.12 show the amount of registers and the amount

	LUTs		Registers	
	#	%	#	%
Fixed	7071	5.28	5472	2.04
BWI	6592	4.93	4979	1.86
BWI+Meta	4173	3.12	4233	1.58

Figure 6.12.: Results of the dispatcher specification in different configurations.

of look-up-tables (*LUTs*) is used by each variant. The here used FPGA provides 133800 LUTs and 267600 registers in total, which results in the percentual usage seen in the table. For the example using additional metadata, it was assumed that the stream *dispatch* only contains values in the range the values are filtered, which makes a 5 bit unsigned integer sufficient for it. Additionally, it was assumed that the *value* only contains 16 bit unsigned integers.

The difference between fixed bit widths and the bit width inference is rather small, with around 7% less LUTs and 9% less registers used. This is to be expected as synthesis tools like Vivado are highly optimised and thus are most likely able to detect bit widths which exceed their needed range quite often. The main advantage of the bit width inference can then be seen in the third test, where it was used in combination with additional metadata, which permitted to reduce the used LUTs

by around 40% and the used registers by around 23%. Therefore, use cases where such information about the input is known prove optimal for this phase.

6.3.2. General

After specifically evaluating the effect of the bit width inference on the area used, we are now reusing the memory allocation example once again to analyse how the entirety of optimisation phases affects the area. The difference in size between the unoptimised and the optimised version is shown in Figure 6.13. As the optimisations

	LUTs		Registers	
	#	%	#	%
No Opt.	6140	4.58	5697	2.13
Opt.	4517	3.38	6176	2.31

Figure 6.13.: Results of the malloc specification with and without optimisation.

have positive and also negative effects to the area used, a generalisation of this is hardly possible. However, a more detailed analysis allows to analyse where the majority of those resources are actually used, which was done for the unoptimised version and represented in Figure 6.14. As you can see here, the vast majority of

	LUTs	Registers
	#	#
Queues	5999	5220
Other	141	477

Figure 6.14.: Area used by queues in the malloc specification without optimisation.

resources is actually being used by queues. The specification was compiled with a queue depth of 4. This shows that when it comes to optimisation of the used area, a good placement as well as the selection of a useful depth for queues is crucial.

In this chapter, we used the implementation of Chapter 5 to evaluate the concept defined in this thesis. More specifically, the area usage and the difference in performance have been measured in different scenarios, to evaluate how adapters, optimisation phases and different input traces affect this. The results have shown that the optimisation phases are able to improve the initial specification drastically in some scenarios, but still leave a lot of room for improvement in others.

7. Conclusion and outlook

We examined the specification language *TeSSLa* and how specifications in *TeSSLa* can be translated into a hardware definition for FPGAs. In a first step, the overall concept was discussed, where the main idea was to represent each stream operator by an individual module representing the semantics of its respective *TeSSLa* operator, and define a communication between them by using an adapted variant of the ready-valid protocol. This concept also permitted to synthesise recursively defined streams and computing their fixed point without it resulting in a combinational loop. Additionally, queues were introduced to handle buffering of values, meet timing constraints and distribute events to multiple target modules.

Multiple different options for optimisations for *TeSSLa* code are discussed, with the goal of improving the synthesised result in the aspects of performance and area used. Overall, the three following optimisation phases were introduced:

1. The stream merging focused on inferring knowledge about how timestamps of different streams relate to each other, which subsequently can be used to merge streams with same or similar timestamps into one tupled stream. This reduces the required amount of synchronisation and may also reduce the amount or size of queues required as events need to be dispatched to less targets than before.

2. The bit stream inference then attempts to infer – potentially with additional metadata provided – required and sufficient bit widths for the different operators. This allows for a reduction in area used, however highly dependent of the nature of the given metadata. Additionally, this also allows to properly catch overflows as well by choosing bit widths such that no overflow occurs.

3. Finally, the queue placement aims to deduce where queues should optimally be placed by weighting of the call graph, giving an estimation of the path length in the resulting graph. Additionally, as queues are significantly big in size, the second goal is to choose useful depths for each queue, depending on the context where it is used.

An implementation of this concept, using the programming language Scala with the library Chisel, is described and evaluated. The evaluation mostly focuses on the effect of the described optimisation phases as their efficiency is a major aspect of this thesis. For this purpose, two separate setups to measure realistic scenarios with

IO as well as purely measuring the processing time to evaluate optimisations have been chosen.

The concept described in this thesis provides therefore a functioning translation from arbitrary *TeSSLa* specifications into efficient hardware definitions. However, there surely is a lot of room for improvement and further optimisations. Some ideas for future features or optimisations are listed in the now following section.

7.1. Outlook

TeSSLa with Delay

In this thesis, we considered the language *TeSSLa* without the delay operator. As using delay also provides a higher expressiveness, it would be favourable support delay as well, by extending the here described concept and optimisations accordingly.

Communication Interface with Timestamp and Value simultaneously

The here described concept does not allow timestamps and values to be transmitted and processed in parallel, thus always requiring at least two clock cycles per event. Modifying this such that both timestamps and values can be processed at the same clock cycle would drastically improve the performance.

Small Recursions as Fold

Currently, even simple definitions like a counter are still kept as recursive definitions, requiring multiple clock cycles to process. However, if their modification of the data solely bases on the trigger and the previous value of the stream, and the modification can be expressed as a one clock cycle operation while still meeting all timing constraints, that definition could also be represented by using the *fold* module described in Section 5.3. This would only need a single clock cycle to process, and also require less resources than a recursive definition does. Therefore, an optimisation which would recognize recursive definitions fitting this pattern and automatically translate them to a equivalent expression using *fold* would pose a significant improvement for recursions.

Reduction of Latency in Cycles

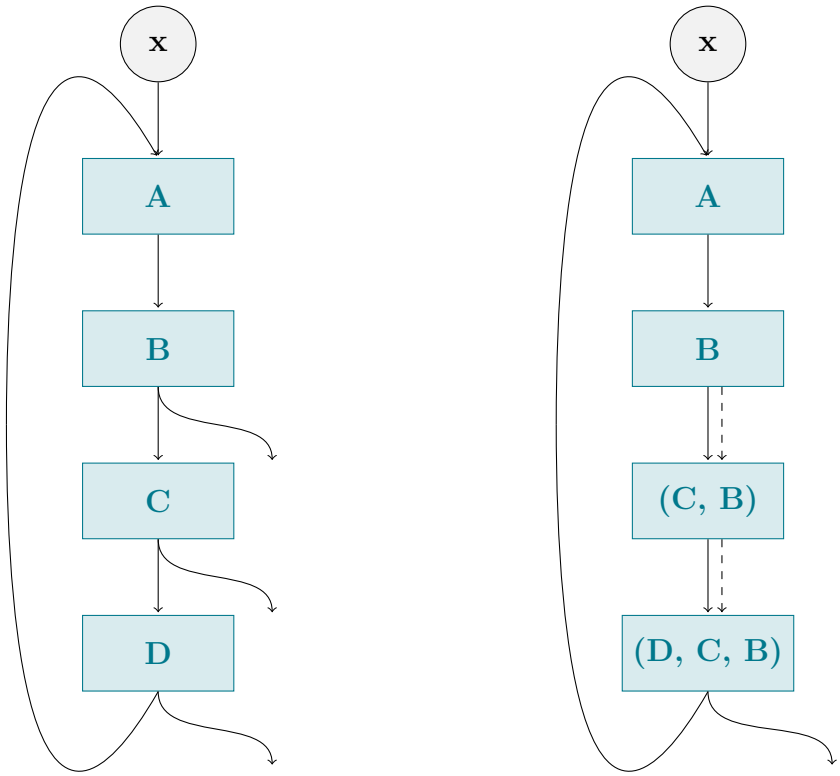
As seen in the previous evaluation, recursive definitions also come with another issue: As the event needs to pass through the whole recursion before the next event can be processed, the latency of the path within the recursion has a direct effect on the throughput of the overall specification. Thus, while the importance of latency for other parts of the specification is rather low, it would prove highly beneficial to keep the latency within a recursion at a minimum. One way to accomplish this would of course be the previously described optimisation, allowing to replace some recursions with *fold*. Another option would be a similar approach to the one used in stream merging: If there is an intermediate result of the recursion which needs to be distributed to multiple targets, a queue has to be placed which increases the latency by one clock cycle. Depending on the path length however, one could merge this value to a tuple with other values, and propagate it further until the next placed queue, such that only one queue is required instead of two.

This concept is shown in Figure 7.1, where the first figure 7.1a represents the initial recursion. In this example, there are three outgoing edges from the recursion, at the respective modules B, C and D, and thus requiring three queues. Assuming that timing processing all operations from A to D in the same clock cycle does still meet the timing constraints, the events can be passed through as tuples as shown in 7.1b, where only one queue is needed.

Improving Placement and Depth Inference of Queues

The concept introduced here to decide the queue placement does not very accurately estimate the actual synthesised design. The first improvement would be to define a more fine-grained weighting to more accurately infer how many LUTs are required on a path. Furthermore, a heuristic to deduce signal travel distance would prove very useful, as currently, all modules with more than one input get always queued. However, this is an inherently difficult task as it is usually not transparent how the synthesis tool generates the layout. However, analysing the call graph and how modules interact with each other, as well as determining where a certain value is being propagated to, may allow for estimates on how they will be laid out on the hardware.

The depth of a queue has a significant effect on the overall size of the specification. This means that it may not be possible to generate queues with the required size at every location in the specification. Therefore, it may prove useful to prioritise some queues over others, such that the biggest queues are placed on the most beneficial positions.



(a) Initial data flow diagram. **(b)** Modified data flow diagram.

Figure 7.1.: Visualisation of passing through values to reduce the amount of queues.

Finally, for a specification with multiple inputs, those inputs may be driven at different frequencies or delays. This also allows conclusions on required queue depths.

Further Reduction of Synchronisation

When motivating the stream merging in Example 4.3, the final result still contained a binary **lift** with two synchronous (after initialisation) inputs. However, with the so far described methods, it is not possible to define a merging for such a case, as all merges so far were based on the inputs being either all **lift** or all **last**. One option here would be to define a mixed module which takes a tupled input and performs different operations on each side of the input. This would be optimal as it would

allow to freely merge inputs independent of their operation, however this probably is rather complex in its implementation. Another approach would be to define a synchronous **lift** module, which is similar to the normal **lift** but without performing a comparison on the timestamps. This is easier to implement and allows to remove that synchronisation step as well, however as there is no actual merge performed, this can not be propagated through the rest of the data-flow diagram as a merged node could.

A. Appendix

This chapter contains more detailed information about some aspects of this thesis.

A.1. Pseudo-Codes

In this section you will find the full pseudo code of modules introduced in Chapter 3 and Chapter 6.

```
register timeReg = 0

wire time = if in.valid && in.isTimestamp
            then in.timestamp else timeReg
timeReg := time

port in {
  ready := out.ready
}

port out {
  valid := in.valid
  isTimestamp := in.isTimestamp
  value := time
  timestamp := time
}
```

Listing A.1: Pseudo code for the time module.

```
wire hasInput = a.valid && b.valid

wire progress = a.isTimestamp && b.isTimestamp
wire progressA = a.timestamp <= b.timestamp
wire progressB = b.timestamp <= a.timestamp

wire processA = !a.isTimestamp
wire processB = !b.isTimestamp
```

```
port a {
  ready := hasInput && (progress && progressA && out.ready
           || processA && (!op.outValid || out.ready))
}

port b {
  ready := hasInput && (progress && progressB && out.ready
           || processB && (!op.outValid || out.ready))
}

port op {
  aValid := processA
  a := a.value
  bValid := processB
  b := b.value
}

port out {
  valid := hasInput && (progress || op.outValid)
  isTimestamp := progress
  value := op.out
  timestamp := if progressA then a.timestamp else b.timestamp
}
```

Listing A.2: Pseudo code for the lift module.

```
port a {
  ready := out.ready || !a.isTimestamp && !op.outValid
}

port op {
  a := a.value
}

port out {
  valid := a.valid && (a.isTimestamp || op.outValid)
  isTimestamp := a.isTimestamp
  value := op.out
  timestamp := a.timestamp
}
```

Listing A.3: Pseudo code for the unary lift module.

```
register active = false
register flush = false
register written = false
register count = 0

port x_1,...,x_n
port y_1,...,y_m
port c

wire anyInValid = x_1.valid || ... || x_n.valid
active := active || anyInValid

count := if !flush && active then count + 1 else count

wire done :=
     y_1.valid && y_1.isTimestamp && y_1.timestamp == t_max
   && ...
   && y_m.valid && y_m.isTimestamp && y_m.timestamp == t_max

flush := flush || done

written :=  writing || (flush && c.ready)

c.value := clock
c.valid := !written || !flush
c.timestamp := t_max
c.isTimestamp := !flush
```

Listing A.4: Pseudo code for the benchmarking module.

```
port in
port out[n]

register timelimit = 0
timelimit := if in.valid && inits == 0
               then in.value else 0

wire initialised = inits == n
inits := if in.valid && !initialised
           then inits + 1 else inits
in.ready := !initialised
```

```
foreach (1 ≤ i ≤ n){
  register timestamp = 0
  register sendTS = false
  xorshift rng

  wire initialising = in.valid && inits == i

  rng.seed := if initialising then in.value else 0

  wire ready = out[i].ready && initialised
  wire done = timestamp > timestamplimit
  wire proceed = ready && !done

  wire send = if proceed then sendTS ^ out[i].ready
                else sendTS && !done
  wire incTS = !sendTS && send
  sendTS := send

  wire nextTS = if incTS then timestamp + 1 else timestamp
  timestamp := nextTS

  wire flushing = sendTS && done
  rng.next := proceed && !sendTS

  out[i].isTimestamp := sendTS
  out[i].timestamp := if flushing then t_max else nextTS
  out[i].valid := (flushing || !done) && ready
  out[i].value := rng.rand
}
```

Listing A.5: Pseudo code of event generator

```
port in[n]
port out

foreach (i){
  in[i].ready := true
}

out.isTimestamp = false
out.timestamp := t_max
out.valid := in[1].isTimestamp && in[1].valid
             && in[1].timestamp == t_max
```

```
                && ...
                && in[n].isTimestamp && in[n].valid
                && in[n].timestamp == t_max
out.value := in[1].value ^ ... ^ in[n].value
```

Listing A.6: Pseudo code of output sink

List of Figures

List of Definitions

List of Listings

Bibliography

[axi17] *AXI Reference Guide*. https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf, 2017

[BVR+12] Bachrach, J. ; Vo, H. ; Richards, B. ; Lee, Y. ; Waterman, A. ; Avižienis, R. ; Wawrzynek, J. ; Asanović, K.: Chisel: Constructing hardware in a Scala embedded language. In: *DAC Design Automation Conference 2012*, 2012, S. 1212–1221

[CHL+18] Convent, Lukas ; Hungerecker, Sebastian ; Leucker, Martin ; Scheffel, Torben ; Schmitz, Malte ; Thoma, Daniel: TeSSLa: Temporal Stream-based Specification Language. In: *CoRR* abs/1808.10717 (2018)

[DDG+18] Decker, N. ; Dreyer, B. ; Gottschling, P. ; Hochberger, C. ; Lange, A. ; Leucker, M. ; Scheffel, T. ; Wegener, S. ; Weiss, A.: Online analysis of debug trace data for embedded systems. In: *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2018, S. 851–856

[DGH+17] Decker, Normann ; Gottschling, Philip ; Hochberger, Christian ; Leucker, Martin ; Scheffel, Torben ; Schmitz, Malte ; Weiss, Alexander: Rapidly Adjustable Non-intrusive Online Monitoring for Multi-core Systems. In: Cavalheiro, Simone (Hrsg.) ; Fiadeiro, José (Hrsg.): *Formal Methods: Foundations and Applications*. Cham : Springer International Publishing, 2017, S. 179–196

[htt] https://electronics.stackexchange.com/users/71487, Francesco C.: *SystemC vs HDLs*. https://electronics.stackexchange.com/a/163208, . – [Online; Accessed the 26.01.2020]

[JBG+15] Jakšić, S. ; Bartocci, E. ; Grosu, R. ; Kloibhofer, R. ; Nguyen, T. ; Ničković, D.: From signal temporal logic to FPGA monitors. In: *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, 2015, S. 218–227

[LSS+18] Leucker, Martin ; Sánchez, César ; Scheffel, Torben ; Schmitz, Malte ; Schramm, Alexander: TeSSLa: Runtime Verification of Non-Synchronized Real-Time Streams. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. New York, NY, USA : Association for Computing Machinery, 2018 (SAC '18), S. 1925–1933

[LSS+19a] Leucker, Martin ; Sánchez, César ; Scheffel, Torben ; Schmitz, Malte ; Thoma, Daniel: Runtime Verification for Timed Event Streams with Partial Information. In: Finkbeiner, Bernd (Hrsg.) ; Mariani, Leonardo (Hrsg.): *Runtime Verification*. Cham : Springer International Publishing, 2019, S. 273–291

[LSS+19b] Leucker, Martin ; Sánchez, Cesar ; Scheffel, Torben ; Schmitz, Malte ; Schramm, Alexander: *Runtime Verification of Real-Time Event Streams under Non-synchronized Arrival*. 2019

[Mar03] Marsaglia, George: Xorshift RNGs. In: *Journal of Statistical Software* 08 (2003), 01

[MN04] Maler, Oded ; Nickovic, Dejan: Monitoring Temporal Properties of Continuous Signals. In: Lakhnech, Yassine (Hrsg.) ; Yovine, Sergio (Hrsg.): *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2004, S. 152–166

[OSV08] Odersky, Martin ; Spoon, Lex ; Venners, Bill: *Programming in Scala: A Comprehensive Step-by-step Guide*. 1. Auflage. Artima Incorporation, 2008

[SJN+17] Selyunin, Konstantin ; Jaksic, Stefan ; Nguyen, Thang ; Reidl, Christian ; Hafner, Udo ; Bartocci, Ezio ; Nickovic, Dejan ; Grosu, Radu: Runtime Monitoring with Recovery of the SENT Communication Protocol. In: Majumdar, Rupak (Hrsg.) ; Kunčak, Viktor (Hrsg.): *Computer Aided Verification*. Cham : Springer International Publishing, 2017, S. 336–355

[ST20] Schmitz, Malte ; Thoma, Daniel: *Internal technical description of TeSSLa synthesis*. 2020

[Tar72] Tarjan, Robert.: Depth-First Search and Linear Graph Algorithms. In: *SIAM Journal on Computing* 1 (1972), Nr. 2, S. 146–160