UNIVERSITÄT ZU LÜBECK

**From the Institute for Software Engineering
and Programming Languages
of the University of Lübeck
Director: Prof. Dr. Martin Leucker**

Efficient Implementation of Stream Transformations

Dissertation
for Fulfillment of
Requirements
for the Doctoral Degree
of the University of Lübeck

from the Department of Computer Sciences and Technical Engineering

Submitted by

Malte Schmitz
from Hamburg

Lübeck 2022

First referee: Prof. Dr. Martin Leucker

Second referee: Prof. João Lourenço, PhD

Date of oral examination: 9 December 2022

Approved for printing. Lübeck, 13 December 2022

Acknowledgement

I have had the support of many people in writing this thesis. I am very grateful to them for supporting me and teaching me so much.

First of all, I would like to thank my supervisor, Martin Leucker, for his support and guidance throughout my academic journey. He introduced me to all the different aspects of runtime verification, provided me with a variety of research opportunities and collaborations, and gave me the freedom and environment to develop ideas and results. I would also like to thank João Lourenço for reviewing this thesis and Mladen Bereković for chairing the examination board.

I am grateful to Daniel Thoma for all our fruitful discussions. His sharp analysis and ideas have greatly improved this thesis. I would also like to thank Torben Scheffel for his continuous support. We met on the first day of our computer science studies and have worked very well together on many projects since then.

I would like to thank Alexander Weiss and Albert Schulz for providing me with the EPUs and for our great cooperation in developing and debugging the EPUs and the corresponding compiler.

I would like to thank the TeSSLa team for their dedicated work on the compilers and ecosystems. I am especially grateful to Sebastian Hungerecker for his work on the TeSSLa compiler frontend, for teaching me how to code properly, and for being the best office mate. I would also like to thank Hannes Kallwies and Thiemo Bucciarelli for their great implementation work on the TeSSLa software compiler, the EPU compiler, and the FPGA compiler.

Abstract

This thesis compares different implementations of the stream transformation language TeSSLa, which is designed as a general-purpose specification language to analyse traces. The focus lies on online monitoring. The monitors do not have access to the entire trace but sequentially process the events of the input trace at the runtime of the system under test.

A TeSSLa specification consists of (potentially recursive) functional definitions of streams deriving new streams from existing streams. This thesis compares monitoring syntheses that translate TeSSLa specifications into a generic software application, into a configuration for dedicated pipelining hardware and into an FPGA image.

The generic software application uses a synchronous evaluation approach. The specification's flow graph is compiled into a Java or Rust program. In a synchronous loop, the graph is entirely evaluated for a timestamp until inputs of the following timestamp are considered.

The dedicated hardware called Event Processing Units (EPUs) is implemented on an FPGA and was developed by Accemic specifically for TeSSLa. The EPUs are inspired by data flow processors and reconfigurable hardware approaches.The TeSSLa implementation on EPUs combines the synchronous approach with pipelining.

The FPGA synthesis employs TeSSLa's support for asynchronous evaluations to utilise the inherent parallelism of FPGAs: The specification's flow graph is directly mapped to the hardware in the form of a network of operators connected with low-level channels. The synchronisation of the inputs happens at every individual operator based on the logical timestamps.

Further, an interpreter written in Scala is introduced as a reference implementation to test the correctness of the other implementations. It follows the same synchronous approach as the compiler, but it dynamically creates an object graph from the specification's flow graph at runtime.

The different monitoring syntheses follow different approaches adapted to the respective environments. In this thesis, they are compared with each other, especially with regard to the following two main questions:

1. How can the different monitor implementations be related to a formal TeSSLa semantics? Different variations of the TeSSLa semantics are discussed and related to each other. The correctness of the different monitor syntheses is shown using synchronous and asynchronous semantics introduced as abstractions of a common monitoring semantics of TeSSLa.

2. How efficient are the different monitoring implementations? The throughput, i. e. the number of processed events over time, is compared empirically on real-world specifications obtained from several research projects. The results show the practical feasibility of all three approaches. Further, the effect of adjusting different parameters is measured using synthetic specifications to gain insights into which monitoring implementation is particularly suitable in which scenarios.

Zusammenfassung

Diese Arbeit vergleicht verschiedene Implementierungen der Stromtransformationssprache TeSSLa, die als universelle Spezifikationssprache zur Analyse von Traces konzipiert ist. Der Schwerpunkt liegt dabei auf dem Online-Monitoring. Die Monitore haben keinen direkten Zugriff auf den ganzen Trace, sondern verarbeiten sequentiell die Ereignisse des Input-Traces zur Laufzeit des zu testenden Systems.

Eine TeSSLa-Spezifikation besteht aus (potentiell rekursiven) funktionalen Definitionen von Strömen, die neue Ströme aus bestehenden Strömen ableiten. In dieser Arbeit werden Monitorsynthesen verglichen, die TeSSLa-Spezifikationen übersetzen in eine generische Softwareanwendung, in eine Konfiguration für dedizierte Pipelining-Hardware und in ein FPGA-Image.

Die generische Softwareanwendung verwendet einen synchronen Evaluierungsansatz. Der Flussgraph der Spezifikation wird in ein Java- oder Rust-Programm kompiliert. In einer synchronen Schleife wird der Graph für einen Zeitstempel vollständig ausgewertet, bis die Eingaben des folgenden Zeitstempels berücksichtigt werden.

Die dedizierte Hardware namens Event Processing Units (EPUs) ist auf einem FPGA implementiert und wurde von Accemic speziell für TeSSLa entwickelt. Die EPUs sind inspiriert von Datenflussprozessoren und rekonfigurierbaren Hardwareansätzen. Die TeSSLa-Implementierung auf EPUs kombiniert den synchronen Ansatz mit Pipelining.

Die FPGA-Synthese nutzt TeSSLas Unterstützung für asynchrone Auswertungen, um die inhärente Parallelität von FPGAs zu nutzen: Der Flussgraph der Spezifikation wird in Form eines Netzwerks von Operatoren, die mit Low-Level-Kanälen verbunden sind, direkt auf die Hardware abgebildet. Die Synchronisation der Eingänge erfolgt bei jedem einzelnen Operator auf Basis der logischen Zeitstempel.

Außerdem wird ein in Scala geschriebener Interpreter als Referenzimplementierung eingeführt, um die Korrektheit der anderen Implementierungen zu testen. Er verfolgt den gleichen synchronen Ansatz wie der Compiler, erstellt aber zur Laufzeit dynamisch einen Objektgraphen aus dem Flussgraphen der Spezifikation.

Die verschiedenen Monitoring-Synthesen verfolgen unterschiedliche Ansätze, die an die jeweiligen Umgebungen angepasst sind. In dieser Arbeit werden sie miteinander verglichen, insbesondere im Hinblick auf die folgenden zwei Hauptfragen:

1. Wie können die verschiedenen Monitor-Implementierungen mit einer formalen TeSSLa-Semantik in Verbindung gebracht werden? Es werden verschiedene Varianten der TeSSLa-Semantik diskutiert und zueinander in Beziehung gesetzt. Die Korrektheit der verschiedenen Monitor-Synthesen wird anhand synchroner und asynchroner Semantiken gezeigt, die als Abstraktionen einer gemeinsamen Monitoring-Semantik von TeSSLa eingeführt werden.

2. Wie effizient sind die verschiedenen Monitor-Implementierungen? Der Durchsatz, d. h. die Anzahl der verarbeiteten Ereignisse über die Zeit, wird empirisch mit realen Spezifikationen aus verschiedenen Forschungsprojekten verglichen. Die Ergebnisse zeigen die praktische Tauglichkeit aller drei Ansätze. Darüber hinaus wird die Auswirkung der Änderung verschiedener Parameter anhand synthetischer Spezifikationen gemessen, um Erkenntnisse darüber zu gewinnen, welche Monitor-Implementierung für welche Szenarien besonders geeignet ist.

Contents

1. Introduction **1**
1.1. Formalisms for Analysing Traces . . . 3
1.2. TeSSLa . . . 6
1.3. Contributions . . . 11
1.4. Outline . . . 13
1.5. Related Work . . . 17

2. Preliminaries **21**

3. TeSSLa **27**
3.1. Motivating Example . . . 29
3.1.1. Lifting Functions on the Data Domain to Streams . . . 29
3.1.2. Synchronisation of Events . . . 30
3.1.3. Filtering Events and Explicitly Handling the Absence of Events 33
3.1.4. Timestamps, Previous Events and Event Creation . . . 35
3.1.5. Aggregating Data Along the Streams . . . 37
3.2. Semantics . . . 40
3.2.1. Streams . . . 41
3.2.2. Syntax . . . 44
3.2.3. Semantics . . . 45
3.2.4. Properties . . . 49
3.3. Common Derived Operators . . . 53
3.3.1. Operators Derived From **lift** . . . 53
3.3.2. Accessing Previous Values . . . 54
3.3.3. Signal Lift . . . 55
3.3.4. Default Values . . . 57
3.3.5. Recursive Equations . . . 58
3.3.6. Generating New Timestamps . . . 61
3.3.7. Implicit Type Conversions and Type Checking . . . 61
3.4. Design Choices . . . 62
3.4.1. Lifting Nested Functions . . . 63
3.4.2. Basic Operators . . . 66
3.4.3. Events and Signals . . . 67
3.4.4. Generating Zeno-Streams . . . 69
3.4.5. Memory Usage . . . 69

3.5. Monitoring 73
3.5.1. Monitoring Streams 74
3.5.2. Monitoring Semantics 77
3.5.3. Examples 79
3.5.4. Fixed Points in the Monitoring Semantics 89
3.5.5. Relation to Semantics 92
3.5.6. Maximal Refinement 93
3.5.7. Fixed Points in the Semantics 95
3.6. Expressiveness of TeSSLa 96
3.7. Conclusion 103

4. Interpreter and Software Compiler **105**
4.1. Semantics 107
4.1.1. Progress 107
4.1.2. Synchronised Streams 109
4.1.3. Operator Functions 114
4.1.4. Synchronised Monitoring Function 117
4.1.5. Examples 120
4.1.6. Correctness and Properties 125
4.2. Implementation Concepts 127
4.2.1. Imperative Algorithm for the Synchronised Monitoring Function 128
4.2.2. Implementing the Closed Operator Function 130
4.3. Interpreter 133
4.3.1. Implementing the Closed Operator Function 133
4.3.2. Implementing the Synchronised Monitoring Function 134
4.3.3. Example 136
4.3.4. Scala DSL 138
4.4. Software Compiler 139
4.4.1. Implementing the Synchronised Monitoring Function 140
4.4.2. Implementing the Closed Operator Function 140
4.4.3. Example 141
4.4.4. Compiler Frontend 143
4.5. Integration and Test Setup 144
4.5.1. Trace Encoding 145
4.5.2. Test Setup 146
4.6. Conclusion 146

5. TeSSLa on Embedded Procssing Units (EPUs) **149**
5.1. Data Flow Processors 150
5.2. EPU Hardware 153
5.2.1. Inner Pipeline 158

5.3. Formal EPU model . . . 159
5.3.1. Execution of a single EPU . . . 164
5.3.2. Execution of an EPU Network . . . 165
5.3.3. Mapping Events to EPUs . . . 169
5.3.4. EPU Simulation . . . 170
5.4. EPU Commands for TeSSLa Operations . . . 171
5.5. Mapping the Dependency Graph on an EPU Network . . . 177
5.5.1. Example . . . 177
5.6. Recursion . . . 180
5.6.1. Example . . . 185
5.6.2. Expressiveness . . . 188
5.7. Fulfilling Hardware Restrictions . . . 188
5.7.1. Splitting Up EPU Commands . . . 190
5.7.2. Condition Configuration . . . 190
5.7.3. Placement of EPU Commands in the Network . . . 194
5.7.4. Enqueuing Commands . . . 198
5.8. Practical Simplifications . . . 198
5.8.1. Flow Graph Optimisations . . . 198
5.8.2. EPU Network Optimisations . . . 200
5.8.3. Translating Recursive Specifications . . . 200
5.9. Optimising Simple Recursions . . . 202
5.10. Integration and Test Setup . . . 209
5.11. Conclusion . . . 210

6. Implementing Asynchronous TeSSLa 211
6.1. Abstract Monitoring Streams . . . 213
6.2. Abstract TeSSLa Operators . . . 219
6.2.1. Delay . . . 222
6.3. Abstract TeSSLa Semantics . . . 229
6.3.1. Quality of the TeSSLa Abstract Monitoring Semantics . . . 231
6.3.2. Equivalence of TeSSLa Specifications . . . 233
6.4. Conclusion . . . 234

7. FPGA Synthesis 237
7.1. Finite Memory . . . 240
7.2. Operator Networks . . . 243
7.3. Translating TeSSLa to Operator Networks . . . 247
7.3.1. Imperative Semantics of the Operators . . . 247
7.3.2. Translating TeSSLa Specifications to Operator Networks . . . 253
7.3.3. Example . . . 257
7.3.4. Simplifications for Timestamp-Conservative Specifications . . . 259

7.4. Implementation Details 260
7.4.1. Implementation of Channels 260
7.4.2. Chisel 262
7.4.3. Implementation of Channel Operators 264
7.4.4. Implementation of Queues 266
7.5. Tuplification Optimisation 270
7.5.1. Timestamp Relations 271
7.5.2. Dependencies 273
7.5.3. Graph Transformations 274
7.6. Integration and Test Setup 277
7.7. Conclusion 280

8. Evaluation **281**
8.1. Measurement Methods 282
8.1.1. Event Generators 284
8.1.2. Interpreter and Compiler 284
8.1.3. EPUs 289
8.1.4. FPGA Synthesis 290
8.2. Real-World Specifications 291
8.2.1. Specifications 291
8.2.2. EPU Optimisation for Simple Recursions 296
8.2.3. Backend Comparison 298
8.3. Synthetic Specifications 300
8.3.1. Specification Depth 300
8.3.2. Recursion Depth 304
8.3.3. Number of Inputs 305
8.3.4. Summary 308
8.4. Comparison of Workflows 310
8.5. Conclusion 312

9. Conclusion and Future Work **315**
9.1. Conclusion 315
9.2. Outlook and Future Work 319

A. Evaluation Appendix **321**
A.1. Specifications 321
A.2. Generators 324
A.3. Measurement Data 329
A.4. Hardware Utilisation 347

1 Introduction

Verification is essential for implementing and integrating software and hardware projects to ensure their correctness. The qualitative and quantitative analysis of execution traces is a branch of verification techniques that can be considered between classic tests [Mye04, BJK+05] and static formal verification techniques like e. g. model checking [CGP99]. While the former checks fixed sets of input/output relations and typically focuses on smaller units of the system under test, the latter statically considers all possible executions of the system and thus often faces the state explosion problem [CKNZ11]. It is rather complicated to consider long executions of complex systems in their environment with both approaches. However, with the growing popularity of distributed, parallel, decentralised and cyber-physical systems, those bugs become common, which can only be detected in the integrated system running in its production environment. Such bugs are sometimes called Heisenbugs because they tend to disappear if one starts probing or isolating them [GT05].

Runtime verification (RV) is a branch of verification techniques that tries to overcome these limitations by analysing and processing execution traces of the system under test with monitors [KVB+99, SKK+99, GH01, HR01, LS09]. RV neither replaces tests nor static verification, but it complements these techniques with tools and methods for monitoring. Among other use cases, RV is well suited for long-term monitoring of systems in their production environment. In general, RV consists of two central aspects:

- The trace observation, i. e. how to get an execution trace from the system under test, and
- the monitoring, i. e. how to process the observed trace.

A distinction can be made between online and offline monitoring: Offline monitoring analyses pre-recorded traces. The monitor has random access to the entire trace. Online monitoring analyses events of the system under test directly during its execution. The monitor processes the events of the trace in the order of their occurrence.

The implementation of the trace observation highly depends on the system under test and the level of interest. A common approach for software monitoring is to instrument the system under test such that it emits events at runtime. When to

emit events is determined by an observation specification defining points of interest, e. g. calling a function, entering or exiting a function, executing a statement, or accessing a variable. The sequence of such events forms the execution trace used as input for the monitor. A common approach for software instrumentation are tools for aspect oriented programming [KLM+97] like AspectJ[1] or AspectC++[2].

While instrumentation is a widespread technique due to its ease of use, it has the drawback of modifying the system under test for the purpose of monitoring. This might affect the timing behaviour of the system and lead to disappearing bugs or introduce new ones. Less invasive techniques utilise debugging interfaces of processors or existing logging layers of communication frameworks. For example the embedded tracing units (ETU) available on many modern processors provide debugging information that can be used for the purpose of runtime verification [CHS+18, DGH+17, DDG+18]. Typical ETUs are available as part of the processor hardware independent of the logic responsible for the execution of the program and thus does not interfere with the execution. To use ETUs for long-term online monitoring dedicated trace processing hardware such as [WLa, WLb] is required to process the provided debugging information online and reconstruct points of interests.

This thesis entirely focus on the specification and implementation of monitors. While monitors could be programmed manually, a common approach in RV is to synthesize monitors from a high-level specification. The usage of high-level specifications abstracts away from implementation details. Different trace sources impose different settings and requirements on the implementation of the monitors, leading to various approaches for the implementation of monitors: Compiling a specification into a software program allows the monitor to be executed alongside the system under test. Software monitors are generic and versatile as they can be executed on many different target platforms.

If the trace observation is done using dedicated FPGA hardware, e. g. for the reconstruction of ETU traces, this hardware can be used for the monitoring, too. By using the same hardware for the observation and the monitoring, transmission and conversion of data for other platforms can be avoided. Further, the inherent parallelism of FPGAs can be utilised for the monitoring, too. While synthesizing a monitor for an FPGA can utilise the properties of the FPGA in order to create an efficient implementation, this is a complex process that might be more involved than compiling a software monitor. A possible compromise between the software and the hardware monitoring approach is dedicated monitoring hardware that can be configured. Such a monitoring configuration can be changed without the need to

[1] `https://www.eclipse.org/aspectj/`
[2] `https://www.aspectc.org`

synthesise a new FPGA image. Fast reconfiguration of the monitor allows interactive debugging sessions [DGH+17].

This thesis discusses online monitoring implementations for the specification language TeSSLa [CHL+18]. TeSSLa allows the specification of monitors in terms of transforming streams. The concept of stream transformation is introduced in detail in the next section. The following three approaches motivated above are discussed for TeSSLa: Compiling a specification

- into a general-purpose software program,
- into a configuration for dedicated monitoring hardware on an FPGA, and
- into a native FPGA image.

The focus lies on monitors that process traces to analyse and verify complex systems in their production environment. Low level debugging utilising debugging interfaces like, for example, ETUs generates a massive amount of data. However, complex bugs might occur only after days of monitoring, leading to online monitoring as the most feasible approach because storing the entire trace is not an option. Instead, the memory consumption should be independent of the length of the trace.

The contribution is elaborated in detail in Section 1.3 below, after discussing different approaches for analysing traces in general and introducing TeSSLa in particular in the following two sections. After an outline of this thesis is given in Section 1.4, this chapter is concluded with an overview of existing work on related monitoring synthesis.

1.1. Formalisms for Analysing Traces

We distinguish the following classes of formalisms suited for analysing and processing execution traces:

- *Logics* like regular expressions [Kle56, Tho68, HMU07] or Linear Temporal Logic (LTL) [Pnu77] can be used to specify sets of allowed traces. In this simple case, a trace is a sequence of letters from a finite alphabet, representing discrete events. Synthesised monitors can then check if the observed trace is an element of the set of allowed traces. [HR02, BLS11]

  Events can be equipped with timestamps which leads to RV with extended logics like timed LTL [BLS11] and timed regular expressions [ACM02]. In the case of Signal Temporal Logic (STL) [MN04] the atomic propositions of discrete events are derived by sampling a continuous signal and Time-Frequency Logic (TFL) [DMB+12] additionally derives propositions from the frequency domain of the input signal. In this classification, the commonality of logics

is the mapping from an observed trace to a single verdict indicating if the observed trace is an element of the specified set of allowed traces. Quantitative regular expressions (QREs) [AFR16] add quantifications to this approach which extends the verdict to numeric values.

Further, some logics can express properties over traces with rich data domains, like e. g. Monitoring Modulo Theories (MMT) [DLT16] which introduces the Temporal Data Logic (TDL) as an extension of LTL. In TDL, instead of fixed atomic propositions, one specifies calculations and comparisons on the data domain. However, the computed values are again only used to specify the set of allowed traces and not to analyse the derived trace.

- *Time-series Databases (TSDB)* are databases with special compression algorithms to efficiently store large time series. [DMF12]. If they are used for runtime verification purposes, then the entire trace is stored in the database and can then be analysed. The main benefit of TSDBs is the fast access to the entire trace, which allows efficient realisations of analyses comparing events and data values at arbitrary positions in the trace. TSBDs are not that well suited for online monitoring of arbitrary long executions of the system under test because their main approach is to store and manage large but finite data sets. Popular examples for TSDBs are Prometheus and InfluxDB. [Ste18]
- *Stream Transformations* take input streams and transform those into output streams. Similar to traces, a stream is a sequence of discrete events. The events can carry values and depending on the formalism be equipped with timestamps, too. So instead of a final verdict or a query result, the analysis provides output streams. Compared to logics, the specification of a stream transformation usually describes how the output streams are derived from the input streams in a more or less constructive way, while logics give the set of correct traces as an existential description. Stream transformations can be compositional, i. e. specifications consist of smaller parts generating intermediate results which are streams, too. These intermediate streams are then used as inputs for the next parts of the specification until the final output streams are computed. Compared to TSDBs, stream transformations do not require a database but are suitable for online monitoring where the output streams are derived from the input streams in a linear style, i. e. the input streams are read, and the output streams are generated sequentially. Focus [BS01] is a well-known formalism for the specification of streams and stream-based systems. In the area of runtime verification, this form of stream processing is sometimes called Stream Runtime Verification (SRV) [BS16, BS14]. Stream transformations support streams with rich data domains as input, intermediate and output streams, and thus naturally combines correctness checks with statistical analysis and other aggregating computations on the streams.

In this thesis, we focus on stream transformations as a tool to analyse traces in order to verify and debug program executions. Stream transformations are a simple and versatile tool to analyse traces. They allow engineers to directly describe operations on the input stream of events instead of specifying a set of allowed executions using a logic. It still provides an entirely different perspective to the system under test than its actual implementation: The global view onto traces allows focusing on specific aspects of the integrated system without considering other implementation aspects.

There are, in general, two different styles of stream transformations: Event processors and synchronous stream transformations:

Event processors consist of operators processing input events in the order of their appearance. If an operator combines multiple streams, then it takes those events available at the evaluation of the transformation without a semantic synchronisation mechanism. There is usually no explicit notion of a logical time. The implementation entirely defines the timing of the event processing.

The idea of event processing is based on the common architectural pattern pipes and filters [Ort05, BMR+96] as well as the idea of functional reactive programming (FRP) [EH97]. The idea of event processing can be found in different architectural patterns like Event Sourcing [Pac18] and is implemented in many popular frameworks, like for example, Akka Streams [Dav19, Chapter 6: Akka Streams] or Azure Functions [KL19]. There are also implementations specialised for RV like BeepBeep 3 [HK17, BKH17, HK18]. In these event-queue based systems, functions on the data domain are applied to queued events without any form of synchronisation other than the order in which the events arrive at the operators.

Synchronous stream transformations follow the synchronous hypothesis, which "states that a system reacts to environmental events in no time. [...] The synchronous hypothesis thus separates the notion of [logical time] from the execution time of the system, which is largely a side-effect of how it is implemented." [PEB07] So in comparison with event processors, we no longer have events being passed through a chain of operators, but we have time instants at which outputs are derived from their inputs. In addition to a causal relation, we now relate input and output events regarding the logical timing of events.

Common synchronous stream transformations are the synchronous stream programming languages Lustre [CPHP87, Hal05], Esterel [BG92, Ber00a, Ber00b] and Signal [GL87] as well as SCADE [Ber07, CPP05] which is a graphical version of Lustre. In the application domain of runtime verification the language LOLA [DSS+05, FFST16] is closely related to Lustre. LOLA adds the ability to refer to future events.

These languages for synchronous stream transformation use synchronous streams: In case of multiple input streams all input streams have events at the same time instants. All derived and output streams have events at these instants, too. The input streams are organised in discrete steps, which usually represent a fixed amount of time. The derived intermediate and output streams are organised in the same steps.

Synchronous stream transformations can be used on asynchronous streams, too. RTLola [FFS+19] is an extension of LOLA, which introduces asynchronous streams to perform aggregations over real-time intervals. Asynchronous streams require the events to carry timestamps indicating their ordering across multiple streams. It is no longer required that all streams have events at the same timestamps. The synchronous hypothesis still applies: Input and output events are related regarding their logical timestamp. All synchronous languages follow the same idea of implicit or explicit logical time that relates the derived events to their origins.

The main difference between Lustre and Esterel is the programming style. Lustre and LOLA consist of stream transforming operations that are applied to the streams in a functional way: Streams and their events' data values are considered immutable, and applying an operator to a stream derives a new independent stream. There is no implicit control flow. Every state must be explicitly realised in the form of an additional stream carrying the current state in its events. Esterel has an imperative programming style: The user specifies an imperative program with an implicit control flow that reads and writes events. The control flow state of this imperative program is implicitly preserved over the entire execution. Explicit statements pause the program until the next instant is reached.

The imperative programming style introduces additional challenges. For example, Esterel supports parallelism on its non-deterministic control flow, making the sequential scheduling of an Esterel program a non-trivial task. In the case of the functional programming style, a deterministic scheduling follows more or less directly from the specification.

1.2. TeSSLa

This thesis is about implementing monitors to analyse and derive additional information from observed traces. This thesis uses TeSSLa [CHL+18] to specify such monitors. TeSSLa is a generic specification language supporting synchronous stream transformation applied to streams with explicitly timestamped events over a continuous time domain. As motivated in the previous section, TeSSLa is a synchronous language with asynchronous streams supporting asynchronous evaluation:

- TeSSLa is a synchronous language adhering to the synchronous hypothesis: Events on the input and output streams are related through the notion of a logical time. The logical time is explicitly available in TeSSLa through timestamps attached to every event.
- TeSSLa uses asynchronous streams: The logical timestamps attached to the events refer to a common global clock and indicate the order of events across multiple streams. It is not required that all streams contain events at the same timestamps.
- The TeSSLa semantics support asynchronous evaluation: The semantics define the transformations performed by TeSSLa's operators using an explicit notion of how far a stream was already processed, called the progress of that stream. This notion of progress defines partial evaluations for streams independent of a global synchronisation.

TeSSLa specifications are written in a functional and compositional style: Streams are considered immutable and operators are applied to streams and return new derived streams. TeSSLa consists of a few basic operators that are composed into more complex operators.

An early version of TeSSLa was first used in [DGH+17]. This early version cannot express recursive specifications, i. e. specifications with cycles in the data flow graph, and it consists of a large set of practically motivated operators. It was formally presented together with an asynchronous evaluation scheme based on message passing in [LSS+18, LSS+20]. The version of TeSSLa used in this thesis was presented in [CHL+18]. It adds the ability for recursive specifications and boils down the formal semantics to three essential operations: Lifting functions on the data domain to streams, accessing previous values and creating events with additional timestamps. TeSSLa was used in several projects to analyse trace data of embedded tracing units of processors. [DDG+18, CHS+18]

[Sch20] introduces future operators for TeSSLa similar to LOLA. However, this thesis focusses entirely on specifications that derive events only based on events with the same or an earlier timestamp. We will introduce this concept as future-independent stream-transformation functions later.

TeSSLa considers a stream as a sequence of discrete events. Every event of a stream has a unique timestamp of a continuous time domain: A stream can have at most one event at a timestamp, but there is no maximal number of events that can occur between two timestamps. The stream's *type* indicates what data values are attached to the stream's events.

Synchronous stream transformations of asynchronous streams are suited for analysing observed traces: Events observed from a real-world system usually have some kind

of a real-world timestamp attached to them. Those timestamps are not following a synchronous pattern of instants, especially not if different trace sources are combined. For example, consider a network interface generating an event for every message sent and received and a processor's debugging interface generating an event for every indirect jump performed by the processor. Both event streams do not have a fixed frequency of events, and even the average event frequency between the two streams might vary heavily. Asynchronous streams can naturally represent such information by equipping events with explicit timestamps. Other than with instants at a fixed frequency, there is no longer a need to encode an event's absence explicitly. Asynchronous signal can also be used to represent sampled real-world continuous signals in a compact way: The continuous signals can be sampled into piece-wise constant signals with dynamically adjusted and non-synchronised sample rates.

However, we are still in the setting of synchronous stream transformations. Every derived event has a precise timing relation to its origin regarding its logical time. The main job of every implementation is to preserve this relation and synchronise processed events based on their logical timestamp.

Using event streams with timestamps over a continuous time domain has two main benefits:

1. It lays the foundations for implementations with an efficient encoding of streams with irregular event patterns. The absence of events is not explicitly encoded. Languages like Lustre and LOLA use streams with a fixed event rate. While this is not necessary a difference in the expressiveness, the concept of timestamps as a first-class citizen in TeSSLa is a different perspective that influences the implementations, especially the hardware backends.

2. TeSSLa can create additional events at arbitrary timestamps. On a discrete sequence of instants, there is no such thing as an additional timestamp, but on a continuous time domain, one can add arbitrary many additional events before the next input event is processed.

TeSSLa orders the events on the streams based on the assumption of a common global clock. Asynchronous streams on a continuous time domain allows the insertion of additional events at arbitrary timestamps. TeSSLa has the expressiveness to describe such complex event generations to gain the most of the extension to continuous streams: Arbitrary event generation is a natural way to monitor complex timing properties on traces.

However, asynchronous streams over a continuous time domain comes with a price: Additional synchronisation effort is needed in every implementation because the input events are synchronised based on their timestamps. Having such a synchronisation mechanism as a fundamental feature of the semantics makes TeSSLa well suited

for analysing observed real-world trace sources. However, it also requires additional implementation effort compared to event processors or stream transformations with a synchronous evaluation of synchronous traces. Hence, TeSSLa implementations usually have no guarantees regarding the execution based on the logical timestamps. While especially the latter is very important for synthesising controllers for applications with hard real-time requirements, we focus on analysing observed traces and gain additional flexibility for that application domain by introducing explicit timestamps over a continuous time domain.

If the insertion of additional events on a continuous time domain is not further restricted, this allows inserting arbitrarily many events between two existing events. The creation of additional events even allows the generation of Zeno streams, i. e. streams with infinitely many events in a finite period. If a TeSSLa implementation must generate infinitely many events before it can read the next input event, this monitor is obviously no longer functioning properly. TeSSLa does not restrict the generation of streams in this respect and thus leaves it up to the user to avoid this problem.

We focus on online stream transformation, i. e. the monitor is a black box reading some events and producing some events without access to the entire stream. In theory, this black box could store the entire stream with unlimited memory, but with the idea of continuous observation for long periods and hardware implementations, this thesis focuses on online monitoring without random access to the entire trace.

A stream transformation function for online monitoring should not change already produced output events after reading further input events. Once the black box has outputted an event, there is no way to change it later. Further, the stream transformation function should not change its behaviour at the transition from finite stream prefixes to infinite streams. The theory is about infinite streams, but the practical implementations are only seeing prefixes of these infinite streams, so we need a relation between these finite prefixes and the infinite streams. We will use Scott continuity [AJ94] to formally characterise these behaviours. Finally, a stream transformation function should be independent of future events because the black box should be able to generate an output based on the events seen so far. Since we do not want to store the entire stream or even larger pieces of streams, the black box must be able to produce an output based on what it has seen so far without the need to wait for future events. We will define *future independence* to formally characterise this behaviour.

TeSSLa is a generic formalism for such stream transformation functions over a continuous time domain. Future-independent stream transformation functions are represented as recursive equation systems of stream transforming operators:

- **lift** lifts functions on values to functions on streams,

- **last** gives access to values of previous events, and
- **delay** generates events with additional timestamps.

Together with the basic stream **unit** containing exactly one initial event and the accessor function **time** used to access an event's timestamp instead of an event's data, these three operators are everything required to formalise arbitrary stream transformation functions over timed event streams.

To sum up these considerations: This thesis discusses different implementations of TeSSLa, which is a specification language designed for the analysis of observed traces and can be characterised as follows:

- TeSSLa is a synchronous functional specification language, i. e. we assume a global clock and input and output events are related regarding their logical timestamps. TeSSLa operators derive streams from streams in a functional way.
- TeSSLa operates on streams of timestamped events on a continuous time domain, i. e. there is no need to encode the absence of events explicitly, but we can always insert arbitrarily many additional events between two existing events. Streams are asynchronous in the sense that there is no global pattern. They naturally encode the asynchronous environment.
- TeSSLa supports asynchronous evaluation. The semantics is given in a way that allows synchronous and asynchronous evaluation. The progress of the individual derived streams depends only on the available data on the inputs on which they depend.
- TeSSLa can express future-independent stream transformations, which are the functions we identified as suitable for analysing observed traces.
- TeSSLa is designed to have explicit memory usage. While it is possible to write TeSSLa specifications that use unbounded memory, this is always done by explicitly using unbounded data structures and never hidden inside the semantics.
- TeSSLa is designed to have a small set of three basic operators. More convenient operators are derived from these basic operators as syntactic sugar.

A detailed introduction into timed event streams and TeSSLa is given in Section 3.1.

1.3. Contributions

As motivated before, this thesis operates in the context of online monitoring using synchronous stream transformations on asynchronous streams with the specification language TeSSLa. Depending on the application area and the origin of the traces, it can be desirable to execute TeSSLa specifications either as a generic software application or on dedicated hardware. This thesis compares software and hardware implementations of TeSSLa. In particular, the following three approaches are discussed:

- *Software compiler.* The TeSSLa specification is compiled into imperative source code that can be compiled and executed on various platforms. The execution scheme is synchronous, and the events of all input streams are processed in sequence. The specification's flow graph is compiled into a sequential imperative program evaluated in an outer loop for every timestamp.
- *Compilation for specialised monitoring hardware.* The TeSSLa specification is compiled for specialised hardware called Event Processing Units (EPUs), which is implemented on an FPGA and was developed by Accemic specifically for TeSSLa [Weia, Weib]. The EPUs are inspired by data flow processors and reconfigurable hardware approaches using switching networks or routing: The EPUs are small processing units organised in a pipeline, and the contained address determines how incoming messages are processed. The TeSSLa implementation on EPUs combines synchronous evaluation with pipelining.
- *FPGA synthesis.* The TeSSLa specification is compiled directly for an FPGA (via Chisel and Verilog). This approach employs TeSSLa's support for asynchronous evaluations in order to utilise the inherent parallelism of FPGA hardware: The specification's flow graph is mapped to the hardware in the form of a network of operators connected with low-level channels. Messages passed between these operators contain timestamps and data values of events such that every operator can advance as far as possible. The synchronisation of the inputs happens at every individual operator based on the logical timestamps.

In order to compare the different approaches, all generated monitors must adhere to the semantics of the specification. However, the different approaches realise very different evaluation principles: The software program synchronously processes a single timestamp before moving on to the next, the EPUs process event streams synchronously in a pipeline, and the FPGA synthesis processes the events asynchronously. This diversity of approaches leads to the first main research question answered in this thesis:

Q1. How can the different implementations and their relationships to the TeSSLa semantics be formally described?

Different variations of the TeSSLa semantics are discussed and related to each other to answer this question. First, the TeSSLa *semantics* is given on *streams* without any consideration of progress. This semantics relates fully known input streams to output streams. In the setting of online monitoring of asynchronous streams, however, we only get partial information about the input streams, i. e. the streams are revealed step-wise to the monitor with every additional observed event. The notion of *monitoring streams* is introduced to encode this form of partially known streams. The TeSSLa semantics' natural extension to monitoring streams leads to the *monitoring semantics*. It describes how much information can already be obtained from incomplete input streams, and this serves as the basis for all implementations.

As an intermediate step toward the actual implementations, two representations of streams are derived that are very close to the actual implementations: The *synchronised streams* are used in the software monitor and the EPUs, and the asynchronous *abstract monitoring streams* are used in the FPGA synthesis. They are described together with their corresponding semantics, the *synchronous monitoring* on the synchronised streams and the *abstract monitoring semantics* on abstract monitoring streams. The methodology of abstractions [CC92, CC77] is used to relate these two semantics to the monitoring semantics. The quality of the abstractions is further characterised by showing that they differ from the monitoring semantics only concerning the handling of incomplete streams. These two formalisms are the central element of addressing the research question formulated above Q1: They formally describe the handling of incomplete streams during online monitoring for the synchronous and the asynchronous evaluation, and their relation to the semantics on fully known streams is characterised.

The actual implementations are then related to the semantics as follows: The software monitor is described using a generic imperative language and directly derived from the synchronised monitoring. A simplified formal model for the EPUs describes their behaviour, and the mapping of a TeSSLa specification to this model in accordance with the synchronised monitoring is given. The behaviour of the synthesised FPGA is described using a network of operators, and a TeSSLa specification is translated into such a network in accordance with the abstract monitoring semantics.

The relation between the different semantics and the implementations is also shown in Figure 1.1 which is discussed further in the next section.

The second part of the contribution is an empirical evaluation of the efficiency of the different implementations, which is carried out to answer the second main research question:

Q2. How efficient are the different implementations?

This question is addressed in this thesis with regard to the run-time efficiency of the implementations. The monitor's throughput, i. e. the number of processed events over time, is identified as an indicator of the monitor's performance.

This thesis discusses different measurement settings and establishes the following approach: The measurements entirely focus on the performance of the monitors trying to ignore as much of its environment as possible. The influence of the trace observation and the I/O operations for reading and writing the trace data are minimised as far as possible.

The throughput is measured on representative sets of specifications used to compare the qualitative and quantitative performance of the different approaches. Real-world specifications obtained from several research projects show the usability of all implementations in practice. Further, the effect of adjusting different parameters is measured using synthetic specifications: The influence on the throughput of the depth of the specification, the depth of the recursive cycles in the specifications and the number of parallel inputs of the specification is compared.

The evaluation carried out according to these considerations does not provide a clear ranking but shows that all three implementations have their justified area of application: The software monitors offer flexibility and efficiency. If the input streams are available in a software setting, the compiled software backend is sufficient in many cases, especially if only individual specifications are evaluated. However, the EPUs can be an almost equally flexible and, in some cases, even significantly more efficient solution. They are especially efficient if the specification consists of many small, mostly but not entirely independent parts. Finally, the FPGA synthesis is less flexible because adjusting the specification requires an entirely new synthesis, but it can be the most efficient solution for large specifications as its performance is independent of the size of the specification.

1.4. Outline

Figure 1.1 shows an overview of the contributions and the outline of this thesis. Chapter 3 starts with an informal introduction into TeSSLa and streams in Section 3.1. The formal TeSSLa semantics on streams are given in Section 3.2. A similar semantics is given in [Sch20], but only as a didactic introduction and without a formal relation to the other used semantics. This semantics is lightweight and easy to understand but not suited for online monitoring.

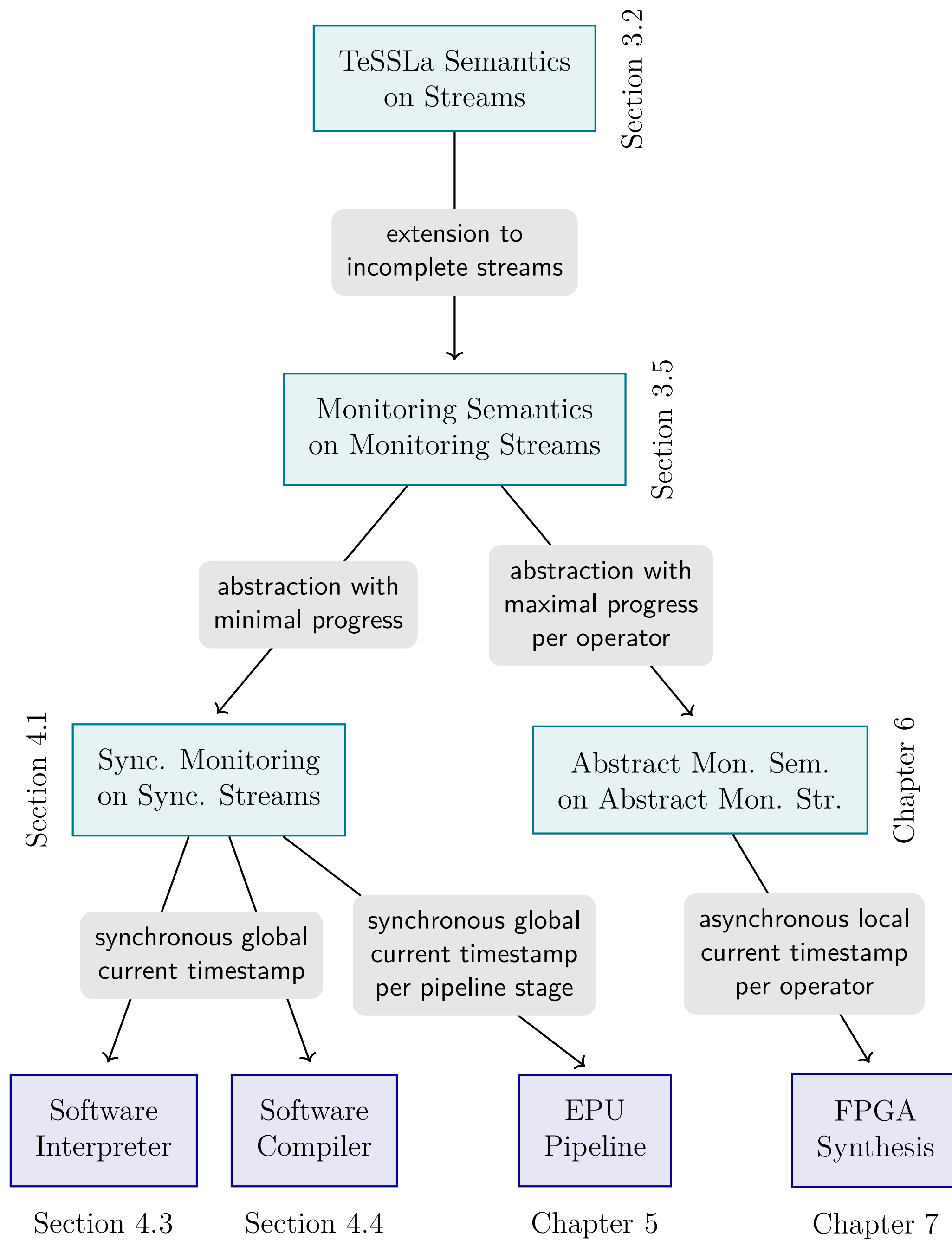


Figure 1.1.: Overview of the different □ semantics and □ implementations introduced in this thesis.

The semantics only define the basic operators. Further convenient operators which are used throughout the thesis are defined in Section 3.3 using equivalences. Section 3.4 continues the discussion of several design choices for TeSSLa, which were already motivated above, especially regarding the set of fundamental operators and the explicit memory usage.

In Section 3.5 the semantics are extended the monitoring streams which can represent incomplete streams. They represent a prefix of a complete stream as a set containing all possible continuations of the prefix. The TeSSLa semantics is extended to monitoring streams using the previously defined operators on streams.

Section 3.6 formally characterises the expressiveness of TeSSLa specifications using Scott continuity and future independence. This thesis does not formally cover the question of what can be expressed in TeSSLa with finite memory. Scheffel's thesis contains detailed discussions on expressiveness and memory usage of TeSSLa fragments [Sch20].

As motivated in the previous section, the monitoring semantics is not directly suited for actual implementations, but it instead serves as a joint base for synchronous and asynchronous evaluations. Thus, the synchronous monitoring and the abstract monitoring semantics are introduced and related to the monitoring semantics using the methodology of abstractions. Similar usage of abstraction was presented in [LSS+19] to handle input streams with partial information, i. e. known gaps in the input. Partial information is not explicitly considered in this thesis, but the concept of abstraction is used to relate the different implementations to the monitoring semantics.

This thesis does not cover the implementation of a parser, type checking, and macro expansion for TeSSLa. TeSSLa specifications are given as equation systems using mathematical notations, and a proper compiler frontend is assumed. The focus is on compiling TeSSLa specifications for different backends.

As shown in Figure 1.1 this thesis presents four different implementations of TeSSLa: A software interpreter serves as a simple reference implementation that is mainly used to verify the correctness of the other implementations. The software compiler uses the same synchronous evaluation as the interpreter but is much faster due to its ability to utilise compile-time and runtime optimisations. In the case of the EPUs, the TeSSLa specification's flow graph is mapped onto a pipeline of special-purpose processors, inspired by the concept of data flow processors. Finally, the FPGA synthesis uses an asynchronous evaluation. The main difference between the synchronous and the asynchronous evaluations is how the necessary synchronisation of events based on their timestamps is implemented: The synchronous evaluation contains a current global timestamp, and all traces are synchronised before entering the monitor. The EPU backend combines the synchronous evaluation with a pipelining

scheme such that instead of one global current timestamp, every EPU (i. e. every stage of the pipeline) has its own current timestamp. The asynchronous character of the monitoring semantics is preserved in the abstract monitoring semantics, which allows the FPGA synthesis to do the synchronisation locally for every translated operator individually, and hence maximal utilise the parallelisation throughout the space of the FPGA.

In Chapter 4 the synchronous semantics on synchronous streams is formally defined. While the monitoring semantics supports asynchronous evaluation and the monitoring streams natively encoded asynchronous event sources, both are synchronised in this approach. It is shown that the synchronous semantics is an abstraction of the monitoring semantics, and the quality of the abstraction is characterised. The implementations of the interpreter and software compiler are based on the same approach: The specification's flow graph is evaluated for the current timestamp. The next timestamp is determined from the input streams and additional derived events. The evaluation of the flow graph for a fixed timestamp is implemented in the interpreter using message passing. The interpreter uses dynamic message passing along the flow graph of the specification and is mainly included in this thesis to illustrate a simple implementation of the synchronous semantics, which is used as a reference implementation to verify the correctness of other implementations. The software compiler evaluates the flow graph using a linearisation. It compiles a TeSSLa specification into an imperative Rust or Java program, consisting of a single outer loop iterating over the timestamps included in the input traces in chronological order.

The compiler for the EPU pipeline is presented in Chapter 5. Each EPU is equipped with a control logic and an ALU. Following the idea of data flow processors, they perform computations based on the incoming messages and do not have a sequential program or a program state other than the incoming messages. The EPUs are organised in the form of a sequential pipeline: Every EPU is a pipeline stage. The TeSSLa specification is mapped onto this pipeline by splitting the flow graph into layers. The EPU pipeline evaluates the synchronous semantics by assigning every EPU a current timestamp, i. e. instead of one current global timestamp, the concept of pipelining allows EPUs earlier in the pipeline to advance further than EPUs later in the pipeline. However, there is no way for messages to travel faster than others. Everything is still synchronous in this single pipeline.

Section 5.3 defines a formal model for the EPUs and Sections 5.4 to 5.6 translates a TeSSLa specification for that formal model. Section 5.7 discusses implementation details of the actual hardware and Sections 5.8 and 5.9 presents optimisations to increase the efficiency of the EPU pipeline.

Chapter 6 presents the TeSSLa semantics used in [CHL+18, Sch20] and shows that they are an abstraction of the monitoring semantics. Hence it is called abstract monitoring semantics in this thesis. The corresponding abstract monitoring streams

no longer explicitly contain all possible continuations but encodes this using the explicit notion of a progress. In comparison to the synchronous streams, every stream can still have its individual progress, and the timestamps of the events are not synchronised across all input streams.

The FPGA synthesis is presented in Chapter 7. It utilises the asynchronicity of the abstract monitoring semantics to implement a local asynchronous evaluation. The specification's flow graph is synthesised as a network of operators communicating through channels. Every operator locally synchronises its input streams based on the event's timestamps. This approach computes every derived stream as far as possible. The translation scheme is compositional in the sense that every synthesised operator's input and output channels directly correspond to edges of the specification's flow graph, i. e. every subgraph of the synthesised network corresponds to a part of the specification's flow graph, which in turn corresponds to a part of the specifications. Section 7.2 defines operator networks as a simple formal model used to abstract the capabilities of an FPGA. Section 7.3 gives a translation of a TeSSLa specification into an asynchronous operator network and Section 7.4 shows how to realise such an operator network on actual FPGA hardware using Chisel [BVR+12] which compiles to Verilog and VHDL.

The empirical evaluation with the goal of the qualitative and quantitative comparison of the performance of the different backends is given in Chapter 8.

Finally, Chapter 9 concludes the thesis with a summary of its essential contents and findings, as well as an overview of open issues and possible extensions.

1.5. Related Work

The synchronous monitoring that is implemented by the software compiler is based on a widespread event-driven synchronous execution scheme which is described in [BCE+03]. This principle is used in many other synchronous stream languages like for example Lustre [HRR91], Esterel [PEB07], LOLA [DSS+05] and RTLola [FFS+19, BFST20, BFST19, FOPS20]. This scheme is extended in two ways to adapt it to TeSSLa: The handling of timestamps as first-level citizens and the possibility to insert events with additional timestamps that are not already present in the input streams. This is discussed in detail in Chapter 4.

Other implementations of the synchronous execution scheme are Copilot [PGMN10], which is a language strongly influenced by LOLA and an internal Haskell DSL that provides building blocks transforming streams into other streams, and HLola [GS21, CGS20] which is an internal Haskell DSL implementing LOLA.

The central idea of the EPUs is to provide reconfigurable hardware for the evaluation of stream transformations.

An approach towards reconfigurable hardware is high-level datapath merging introduced by [AMHM02] and applied to an early acyclic version of TeSSLa [LSS+18, LSS+20] in [DGH+17, GH17]. The main idea of datapath merging is to merge several flow graphs into a parameterised larger graph such that the original graphs can be obtained at runtime by setting the correct parameters. The main drawback of high-level datapath merging is that one needs to know all relevant data paths in advance in order to merge them. The EPUs provide a more generic approach because they do not rely on parameters selecting subgraphs but are freely configurable.

The most generic approach to reconfigurable FPGA designs is ZUMA [BL12] that is an embedded FPGA architecture on an FPGA, i. e. an FPGA synthesised on an FPGA. More specific to the execution of data flow graphs and related to the ideas of EPUs are approaches of processing elements or functional units that are interconnected by a switching network or routing [GHS11, GSM+99, CA13].

The EPUs combine this approach of realising pipelines through reconfigurable interconnection networks with the central idea of data flow processors, i. e. the incoming message determines how it is processed. The dataflow architecture was introduced in [DM74] and further extended in [Vee86, AC03, DG88, PC90]. For an overview and introduction into data flow processors see for example [HK08] and [SRU99, Chapter 2]. Further aspects of data flow processors and a detailed comparison with EPUs is given in Chapter 5.

The FPGA synthesis utilises the asynchronous monitoring semantics to implement an asynchronous evaluation. Evaluating TeSSLa specifications using asynchronous message passing along the specification's flow graph was already studied with an Erlang implementation, and an early acyclic version of TeSSLa in [LSS+18, LSS+20].

Synthesising Lustre on hardware in [RH91] follows the idea of a synchronous evaluation, i. e. the entire specification is evaluated for every time instant to compute the outputs for the current inputs. The same holds for Esterel's hardware synthesis [Ber16, PEB07]: Esterel can be used to program an FPGA in Esterel [HN03]: The user can specify the FPGA's behaviour down to clock cycles in Esterel. The TeSSLa synthesis has an additional abstraction due to the synchronisation of streams based on the event's logical timestamps. This allows writing more high-level specifications referring only to the logical timestamps of the streams. The user cannot refer to the FPGA's clock cycles in TeSSLa.

There are compilations of LOLA to FPGAs [BFST20, BFST19, BFS+20], too. They split up the monitoring synthesis on the FPGA into two sequential parts: A deadline detection happens first and based on its results, the actual event processing takes

place in the form of a synthesised automaton. As mentioned before, the TeSSLa synthesis is entirely compositional.

There are also several approaches to execute monitors on FPGAs for various logics which are not stream processing:

There has been work on the synthesis of STL to FPGAs in different ways [JBG+15, JBGN16, SJN+17] and a synthesis for past time LTL for the observation of hardware buses is described in [PMCR08]. The synthesis for mission-time bound LTL is compositional similarly to the TeSSLa synthesis [SMR15, RRS14]. However, they reduce the need for synchronisation by avoiding any recursive cycles in the data flow path.

Approaches also allowing for reconfiguration are described in [MRS17] for past-time LTL and in [RFB14] for past-time MTL with a microcomputer synthesized onto an FPGA. The dynamic evaluation of LTL_3 monitors [BLS11] on FPGAs with micro-programmed FSM is presented in [BHW+13].

2 Preliminaries

This chapter establishes names and notations for common mathematical concepts used throughout this thesis.

Sets

We use the notation $\mathbb{U} = \{\square\}$ for the unit type whose only value is $\square$, called unit.

We use the notation $\mathbb{B} := \{\mathit{true}, \mathit{false}\}$ for the set of Boolean values.

We use the notation $\mathbb{N} = \{0, 1, 2, \ldots\}$ for natural numbers, i. e. non-negative integers. For a comparison operator $\diamond \in \{<, \leq, >, \geq\}$ and a constant $c \in \mathbb{N}$ we use the shorthand $\mathbb{N}^{\diamond c} := \{n \in \mathbb{N} \mid n \diamond c\}$ for restricted numerical sets, e. g. $\mathbb{N}^{>0}$ for natural numbers without zero, i. e. positive integers. The same notation is used for the set $\mathbb{Z}$ of integers and the set $\mathbb{R}$ of real numbers.

We use the notation $\mathbb{D}_\perp := \mathbb{D} \cup \{\perp\}$ to indicate sets extended with the additional symbol $\perp$ called *bottom*.

Functions

Let $R \subseteq X \times Y$ be a binary relation over two sets X and Y. R is *functional* if

$$\forall\, x \in X \colon \forall\, y, z \in Y \colon (x, y) \in R \wedge (x, z) \in R \Longrightarrow y = z.$$

R is *serial* if

$$\forall\, x \in X \colon \exists\, y \in Y \colon (x, y) \in R.$$

A *partial function* $f \colon A \hookrightarrow B$ is a binary relation $f \subseteq A \times B$ which is functional. A *function* $f \colon A \to B$ is a partial function which is serial.

Every partial function $f \colon A \hookrightarrow B$ can be transformed into the equivalent function $f \colon A \to B_\perp$ by adding $(a, \perp)$ to the relation for all $a \in A$ which are not in the relation. We use this notation to specify partial functions.

We use the notation $A_1, A_2, \ldots, A_n \rightarrowtail B$ to indicate the function $A_{1\perp} \times A_{2\perp} \times \ldots \times A_{n\perp} \to B_\perp$, called $\perp$-function. Such a $\perp$-function f is called

- *well-formed* iff $\forall\, 1 \le i \le n\colon a_i = \perp \implies f(a_1, a_2, \ldots a_n) = \perp$,
- *conservative* iff $\exists\, 1 \le i \le n\colon a_i = \perp \implies f(a_1, a_2, \ldots a_n) = \perp$, and
- *ideal* iff $\exists\, 1 \le i \le n\colon a_i = \perp \iff f(a_1, a_2, \ldots a_n) = \perp$.

A partial function $f\colon A_1, A_2, \ldots, A_n \hookrightarrow B$ can be extended to a conservative $\perp$-function $f_\perp\colon A_1, A_2, \ldots, A_n \rightarrowtail B$ as follows:

$$f_\perp(a_1, a_2, \ldots, a_n) = \begin{cases} f(a_1, a_2, \ldots, a_n) & \text{if } \forall\, 1 \le i \le n\colon a_i \ne \perp \\ & \text{and } f(a_1, a_2, \ldots, a_n) \text{ is defined,} \\ \perp & \text{otherwise.} \end{cases}$$

A function $f\colon A_1 \times A_2 \ldots A_n \to B$ can be extended to an ideal $\perp$-function $f_\perp\colon A_1 \times A_2 \ldots A_n \rightarrowtail B$ as follows:

$$f_\perp(a_1, a_2, \ldots, a_n) = \begin{cases} f(a_1, a_2, \ldots, a_n) & \text{if } \forall\, 1 \le i \le n\colon a_i \ne \perp, \\ \perp & \text{otherwise.} \end{cases}$$

We assume (partial) functions to be implicitly extended to $\perp$-functions if needed.

Let $f\colon A \to B$ and $g\colon B \to C$ be two functions over the sets A, B and C. Then the notation $f \circ g$ indicates the functional composition given by

$$f \circ g(x) = g(f(x)).$$

Let $f\colon (A \to B) \to (C \to D)$ be a function that maps functions to functions, $g\colon A \to B$ a function and $c \in C$ a value. We use the notation $f(g)\colon C \to D$ to denote the function retrieved by applying f to g. We further use the notation $f(g)(c)\colon D$ to denote the value retrieved by applying c to the function retrieved by applying f to g.

We use the notation $id\colon A \to A$ with $id(x) = x$ for the identity function.

For a Boolean value $b \in \mathbb{B}$ and two values $d, e \in \mathbb{D}$ we use the *conditional operator* as an infix notation for the *conditional function* $ite\colon \mathbb{B} \times \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ with

$$b \,?\, d : e := ite(b, d, e) := \begin{cases} d & \text{if } b, \\ e & \text{otherwise.} \end{cases}$$

Semirings

A set R with an addition $+$ and a multiplication $\cdot$ together with the identity elements 1 and 0, respectively, form a *semiring* $(R, 0, 1, +, \cdot)$ if the following conditions are met for any $a, b, c \in R$ [BPR10]:

- $(R, 0, +)$ is a commutative monoid, i. e. $(a+b)+c = a+(b+c)$ (associativity), $0 + a = a + 0 = a$ (identity element) and $a + b = b + a$ (commutativity).
- $(R, 1, \cdot)$ is a monoid, i. e. $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ (associativity) and $1 \cdot a = a \cdot 1 = a$ (identity element).
- $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ (left distributivity) and $(a + b) \cdot c = (a \cdot c) + (b \cdot c)$ (right distributivity).
- $0 \cdot a = a \cdot 0 = 0$ (zero element).

Orders

A set P and a relation $\leq$ form a *partial order* $(P, \leq)$ if the following conditions are met for any $a, b, c \in P$ [AJ94]:

- $a \leq a$ (reflexivity).
- If $a \leq b$ and $b \leq a$ then $a = b$ (antisymmetry).
- If $a \leq b$ and $b \leq c$ then $a \leq c$ (transitivity).

Let $(P, \leq)$ be a partial order and $S \subseteq P$ be a subset of P. An element $a \in S$ is called the *minimum* of S if it is the least element of S, i. e.

$$\forall\, s \in S \colon a \neq s \Rightarrow a < s.$$

An element $a \in S$ is called the *maximum* of S if it is the greatest element of S, i. e.

$$\forall\, s \in S \colon a \neq s \Rightarrow s < a.$$

An element $a \in P$ is called the *infimum* of S if it is the greatest lower bound for S [AJ94], i. e. the following maximum exists:

$$a = \max\{p \in P \mid \forall\, s \in S \colon p \leq s\}$$

An element $a \in P$ is called the *supremum* of S if it is the least upper bound for S [AJ94], i. e. the following minimum exists:

$$a = \min\{p \in P \mid \forall\, s \in S \colon s \leq p\}$$

A partial order $(P, \leq)$ is a *total order* if the reflexivity can be extended to the condition that we have $a \leq b$ or $b \leq a$ for all $a, b \in P$ (connexity).

Kleene Fixed-Point Theorem

Given a partial order $(A, \leq)$, a subset $D \subseteq A$ is called *directed* [AJ94] if it is not empty and for each pair of elements in D there exists an upper bound in D, i. e.

$$\forall\, a, b \in D \;\; \exists\, c \in D \colon a \leq c \;\wedge\; b \leq c.$$

A partial order $(A, \leq)$ is called *directed-complete partial order (dcpo)* [AJ94] if there exists a supremum $\bigvee D$ for every directed subset $D \subseteq A$.

Let $f \colon A \to B$ be a function and $(A, \leq)$ and $(B, \leq)$ two partial orders.

- The function f is called *monotonic* [AJ94] if it preserves the order, i. e.

  $$\forall\, a_1, a_2 \in A \colon a_1 \leq a_2 \Rightarrow f(a_1) \leq f(a_2).$$

- The function f is called *Scott-continuous* [AJ94] if it preserves all directed suprema, i. e. for all directed subsets $D \subseteq A$ with supremum $\bigvee D$ in A the supremum of the image of D under f exists in A and is equal:

  $$\bigvee f(D) = f\left(\bigvee D\right).$$

  Every Scott-continuous function is monotonic.

By the *Kleene fixed-point theorem* [Tar55, SLG94], every Scott-continuous function $f \colon A \to A$ has a least fixed point μf if $(A, \leq)$ is a dcpo with a least element 0. The least fixed point μf is the supremum of the chain iterating f starting with the least element:

$$\mu f = \bigvee \{f^n(0) \mid n \in \mathbb{N}\}.$$

This chain is called the *Kleene chain*.

Abstractions

Let $(A, \preceq)$ and $(B, \preceq)$ be two partial orders. A pair of monotone functions $\alpha \colon A \to B$ and $\gamma \colon B \to A$ is called a *Galois connection* [EKMS93, Ore44] if

$$\forall\, a \in A, b \in B \colon \alpha(a) \preceq b \iff a \preceq \gamma(b).$$

Let $(A, \preceq)$ be a partial order, $f \colon A \to A$ a monotone function and $\gamma \colon B \to A$ a function. Then the function $f^{\#} \colon B \to B$ is an *abstraction* [CC92, CC77] of f if

$$\forall\, b \in B \colon f(\gamma(b)) \preceq \gamma(f^{\#}(b)).$$

If (α, γ) is a Galois connection between A and B, we call the function $f^{\#}\colon B \to B$ given by

$$f^{\#}(b) := \alpha(f(\gamma(b))$$

the *perfect abstraction* of f.

Sequences

A *sequence* is a list of elements with a particular order. All elements are from a common (finite or infinite) base set. For a set A we denote the set of all finite sequences over A with A^*. We write

$$s = \langle s_0, s_1, \ldots, s_{\ell-1} \rangle \in A^*$$

and say, the sequence s is of length $|s| = \ell$. For a set A we denote the set of all infinite sequences over A with A^ω. We write

$$s = \langle s_0, s_1, \ldots \rangle \in A^\omega$$

and say, the sequence s is of length $|s| = \infty$. The set $A^\infty := A^* \cup A^\omega$ is the set of all finite or infinite sequences over A.

Let $s \in A^*$ be a finite sequence of length ℓ and $s' \in A^\infty$ a finite or infinite sequence. We use the following notation to denote their concatenation:

$$s \mathbin{\&} s' := \langle s_0, s_1, \ldots, s_{\ell-1}, s'_0, s'_1, \ldots \rangle.$$

We use the term sequence and not word to distinguish sequences from words which are elements of formal languages. Our base set A is allowed to be infinite which would be uncommon for alphabets of typical formal languages.

Graphs

A *directed graph* $G = (V, E)$ consists of a set V of *vertices* and a set $E \subseteq V \times V$ of *edges*. A *directed multi-graph* $G = (V, E)$ has a multi-set E of edges. A *path* in G is a finite or infinite sequence $v_1, v_2, \ldots \in V^\infty$ of vertices such that every vertex is at most contained once in the sequence and for any two consecutive vertices v_i and v_{i+1} there is an edge $(v_i, v_{i+1}) \in E$. A subset of a graph's vertices is called *strongly connected* iff there is a path from each vertex to all others in the subset. Such a subset is called *strongly connected component (SCC)* if it is maximal, i. e. all strictly larger subsets of the graph are not strongly connected. An SCC is called *non-trivial* if it contains at least two nodes [BM76].

3 TeSSLa

This chapter discusses the TeSSLa semantics as a theoretical foundation for the different implementations discussed in the next chapter. We will start in Section 3.1 with an informal introduction into streams, the usage of TeSSLa and the available TeSSLa operators. Formal definitions for streams and the TeSSLa semantics are given in Section 3.2.

Informally a *stream* is a sequence of events, and each *event* consists of a timestamp and a data value. A *timestamp* is a number indicating when the event happened. There is no theoretical limit to the timestamp's precision, i. e. there can always be another timestamp between two timestamps. We require all events of a stream to be linearly ordered by their timestamp, i. e. two events on the same stream cannot have the same timestamp. We refer to a set of timestamps and available operators on timestamps as *time domain*. The data value is an arbitrary value attached to the event. Streams are typed by the set of possible data values. We refer to a set of possible data values and available operators on these values as *data domain*. The data domain is not restricted and can range from primitive data types like Booleans or integers to complex data structures such as lists or sets.

While the data domain of streams might differ, we require all streams to have the same time domain. Two streams might have events with the same timestamp, although it is not required that all streams have their events at identical timestamps. We assume all timestamps to be comparable, i. e. events of different streams can be ordered by their timestamps. This concept was introduced as the assumption of a common global clock in the introduction.

We do not directly define a prefix relation on streams, i. e. a stream cannot be extended into larger streams. The length of a stream might differ from its number of events and requires additional consideration because the events are timestamped, and the absence of events is not explicitly encoded. We thus do not introduce an explicit length of streams. Instead, we consider every stream to be of infinite length. This approach makes the definition very simple but has the drawback that we do not immediately get an operational version of the semantics: A TeSSLa specification is given in the form of an equation system, and we can only check for a given solution, i. e. a set of streams if they fulfil the specification. The TeSSLa semantics do not provide a systematic way to compute a solution for a given TeSSLa specification.

In Section 3.5 we introduce monitoring streams as an extension of streams: We define a prefix relation for monitoring streams and introduce the concept of a monitoring stream's progress, which generalises the concept of a stream's length. The progress of a monitoring stream encodes what we already know about the monitoring stream and what information can be added in possible future extensions of the monitoring stream. Monitoring streams are defined as a set of streams, which can be seen as the most general way to express progress. The TeSSLa monitoring semantics is introduced as TeSSLa semantics over monitoring streams. Using the monitoring streams' prefix relation the monitoring semantics provides an operational way to compute solutions for a given TeSSLa specification: We will show that the Kleene fixed-point theorem renders us an iterative approach starting with the empty chain and step-wise extending it until the solution is reached. Further, the monitoring semantics supports online monitoring, i. e. the online evaluation of streams extended over time. The monitoring semantics extends the TeSSLa semantics: It uses the same semantics for the individual TeSSLa operators.

In the following chapters, we introduce additional semantics with advantages for implementing the different TeSSLa engines. As motivated in the introductions, these semantics are shown to be abstractions of the TeSSLa monitoring semantics.

The formal semantics is only given for a minimal set of operators. Using these operators directly can be cumbersome when writing real-world specifications. Further derived operators are therefore introduced as syntactic sugar based on the basic operators in Section 3.3. These derived operators are defined to be equivalent to a composition of existing TeSSLa operators. We will also show that equivalences on the TeSSLa semantics hold on the TeSSLa monitoring semantics.

Before the formal introduction of the monitoring semantics, we discuss several design choices and basic principles of TeSSLa in Section 3.4. Several other sets of operators are defined in this section, which could have also been chosen as basic operators. We discuss the drawbacks of these operators and why the TeSSLa basic operators are defined the way they are.

Finally, in Section 3.6 we show several results on the expressiveness of TeSSLa. Using the tools defined in the previous section on the monitoring semantics, we can conclude this chapter with a precise characterisation of the stream transformation functions that can be expressed with TeSSLa. As already motivated in the introduction, TeSSLa is designed to express stream transformation functions that can be realised in an online monitoring setting with finite memory.

3.1. Motivating Example

A TeSSLa specification is a set of equations that describes the relationship between streams. A formal definition of a stream is given in Section 3.2.1. For the motivating examples in this section, we consider a stream to be a sequence of events, each consisting of a timestamp and a data value. We distinguish between *input streams* and *derived streams*. Applying a stream transformation to a set of input streams derives new streams from the input streams. Input streams are given streams. They are not defined by the specification but used as inputs of the specification. Derived streams are derived from other streams (input streams and derived streams) using TeSSLa operators. In theoretical considerations, we treat all derived streams as the output of the specification. In practical applications, we distinguish further between intermediate streams and output streams. Intermediate streams are only used to derive further streams, and output streams are the actual output of the specification.

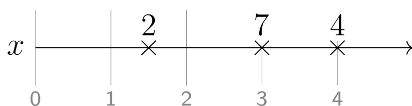
This section aims at an informal introduction into the usage of TeSSLa and the available TeSSLa operators. The formal semantics is given later in this chapter. Basic TeSSLa operators are printed in **bold** and are defined in Section 3.2. Derived operators are printed in sans serif and are defined in Section 3.3.

In the PDF version of this document, one can click on the operators to jump to their formal definition. In the printed version, the formal definitions of the operators are all listed in the list of theorems and definitions in Appendix A.4.

3.1.1. Lifting Functions on the Data Domain to Streams

The **lift** operator lifts a function on the data domain to a function on streams by applying the function to every event of the stream.

Example 3.1 (Lifting Unary Functions to Streams)**.** As a first example consider the following stream x over the time domain $\mathbb{R}$ of real numbers and the data domain $\mathbb{Z}$ of integers:



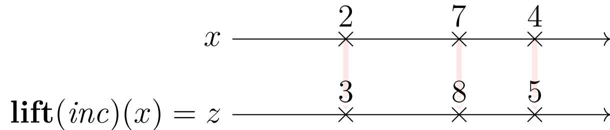
The stream x consists of three events at the timestamps $1.5, 3, 4 \in \mathbb{R}$ with the values $2, 7, 4 \in \mathbb{Z}$. The time flows from the left to the right, indicated by the timeline in the graphical representation. A cross indicates every event. The position on the x-axis is related to the event's timestamp, and the event's data is written above the cross.

The grey lines are meant to give an orientation about the timestamps. We will add these lines in later diagrams only if the concrete timestamps are relevant.

Consider the following TeSSLa specification:

$$z = \textbf{lift}(inc)(x)$$

The function $inc\colon \mathbb{Z} \to \mathbb{Z}$ increments an integer, i. e. $inc(i) = i + 1$ for any $i \in \mathbb{Z}$. The **lift** operator lifts the function inc to streams by applying it to every event's data individually:



The light red arrows indicate the data flow and dependency between the events: An arrow connects two events if the second event's data value is derived from the first event's data value. ⌟

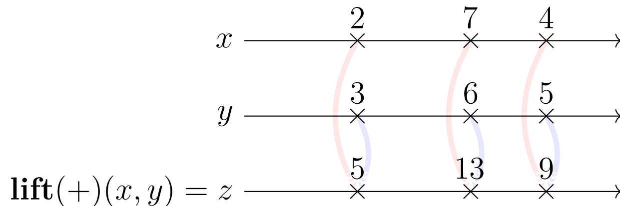
3.1.2. Synchronisation of Events

We use streams over a continuous time domain that explicitly specify the event's timestamps. The continuous time domain is the main difference between TeSSLa and synchronous stream processing languages over a discrete time domain like, e. g. LOLA [DSS+05]. Our timestamps serve multiple purposes: They are used to indicate the order of the events, and they can represent the exact time when the event happened. See the next section on a detailed discussion of timestamps.

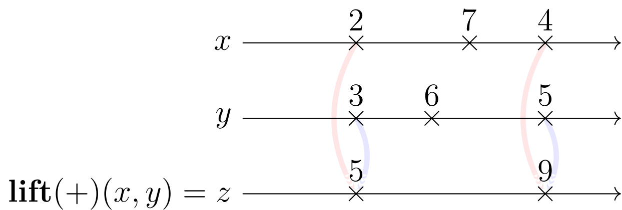
Example 3.2 (Lifting Binary Functions to Streams)**.** We can extend the previous example to a binary function $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, which takes two elements from the data domain and sums them up, i. e. returns another element from the data domain.

$$z = \textbf{lift}(+)(x, y)$$

We can lift such a function to streams over this data domain by applying it to the corresponding events of two streams.

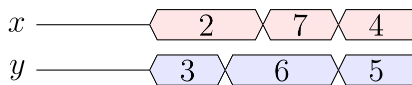


In the above example, the two input streams x and y are synchronous in the sense that their events have identical timestamps. Next, we consider the case of events without corresponding events with the exact timestamp on the other input streams. Streams do not explicitly encode the absence of events. As a result, the **lift** operator cannot use every i-th event of the input streams to compute the i-th event of the output stream. It uses the event's timestamps instead of their sequence number to synchronise the input streams, i. e. use those events of the input streams that have the same timestamp to compute an event with that timestamp on the output stream. A total function like $+$ on the data domain is lifted to a function on streams over the data domain which only considers events with the same timestamp and ignores the others:



Some events on x and y are ignored because of the missing simultaneous event on the other stream. ⌟

Ignoring those events makes sense if we interpret an event as a momentary action that happens at a particular point in time. We call this the *event view* of a stream. One can also interpret the events as points in a stream where a piece-wise constant signal changes its value. We call this the *signal view* of a stream. This is illustrated using the following digital signal notation:



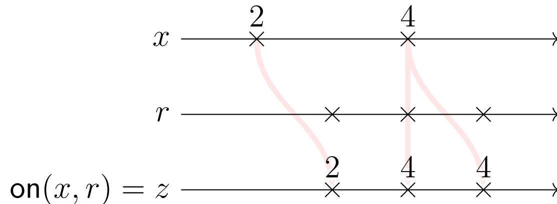
With this interpretation, one wants to lift functions on the data domain to functions on the streams so that the piece-wise constant signals' current values for every timestamp fulfil the operation. We refer to the current value of a stream if it is seen as a piece-wise constant signal as the *last known value* of a stream. So we want to consider the last known value on both input streams for every event on either of the two input streams to compute new events on a derived stream. The necessary control over asynchronous streams is provided by sampling one stream's event using another stream:

Example 3.3 (Sampling Events)**.** In the following specification the `on` operator takes two streams x and r and repeats the last event's value on the stream x for

every event on the stream r:

$$z = \mathsf{on}(x, r)$$

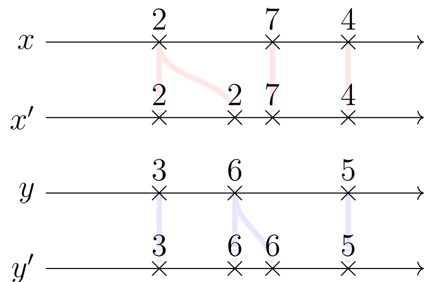
Note how data dependencies indicated by the light red arrows are no longer exclusively between events with the same timestamp. The on refers to previous events on x:



Example 3.4 (Synchronising Events)**.** The sync operator is a special case of sampling: It synchronises events between two streams, i. e. samples each stream with the events of the other stream:

$$x' = \mathsf{sync}(x, y)$$
$$y' = \mathsf{sync}(y, x)$$

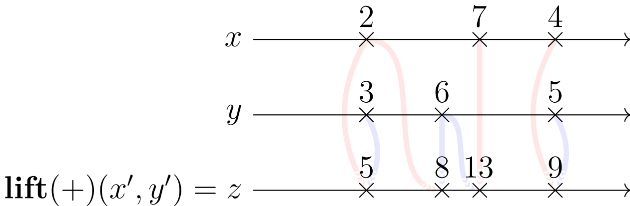
A stream cannot have two events with identical timestamps. So even if both streams have an event at the same timestamp, we end up only with one event with that particular timestamp. In the corresponding visualisation we can see how the derived streams x' and y' contain the same events as x and y and additional sampled events repeating the same value:



Now we can use the synchronised x' and y' as inputs to a **lift** to use the last known value of a stream instead of ignoring events without a simultaneous event on the other stream:

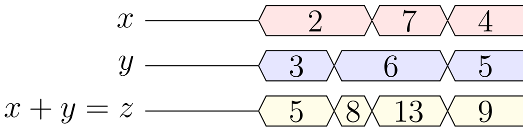
$$z = \textbf{lift}(+)(x', y')$$

The visualisation shows how events without a simultaneous event on the other stream are no longer ignored. Instead the last known event on the other stream is used:



Since this combination of synchronising streams and then applying a function on the data domain to the streams with the **lift** operator is an essential operation, we call this *signal lift* or slift. Further, if functions (and operators) defined on the data are used on streams, we assume an implicit signal lift. With this conventions we can write $z = x + y$ instead of $z = \mathbf{lift}(+)(\mathsf{sync}(x, y), \mathsf{sync}(y, x))$.

Coming back to the digital signal notation we now have:



Although the above visualisation conveys the idea of the signal view better, for the rest of the thesis, we primarily use the stream diagrams in which crosses indicate the events because these are closer to the actual implementations of the streams. ⌟

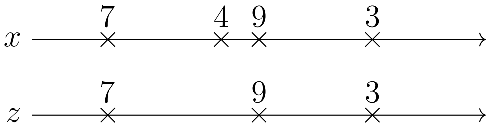
3.1.3. Filtering Events and Explicitly Handling the Absence of Events

If we lift a partial function, we can use the **lift** operator to filter streams.

Example 3.5 (Lifting Partial Functions)**.** Consider the partial function $f : \mathbb{N} \hookrightarrow \mathbb{N}$ mapping a number $n \in \mathbb{N}$ to itself if its odd. Then the specification

$$z = \mathbf{lift}(f)(x)$$

filters the stream $x \in \mathcal{S}_\mathbb{N}$ and removes all events with even data value:

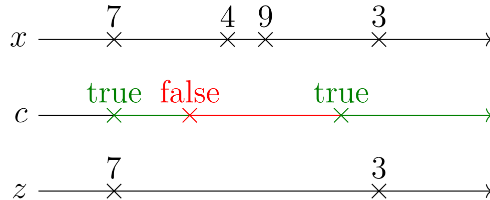


This approach can be extended to define a Boolean partial function $f : \mathbb{D} \times \mathbb{B} \hookrightarrow \mathbb{D}$ taking a data value and a condition. It maps the data value to itself if the condition is true: $f(d, \mathit{true}) = d$. If we lift such a function to streams, we get a function that takes a stream of data values and a stream of conditions. By synchronising

the condition stream with the data stream we can filter a stream based on the last known value on a condition stream:

$$z = \textbf{lift}(f)(x, \mathsf{sync}(c, x))$$

Note how we combined event and signal view in this example. The condition stream is synchronised with the data stream, but the data stream is considered as a stream of individual events which are not synchronised with the condition stream.

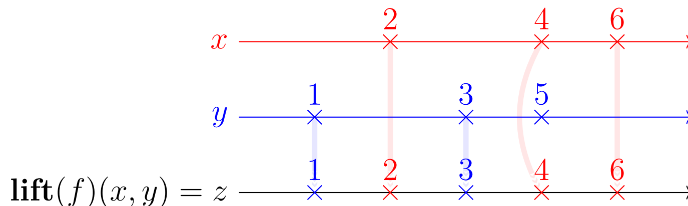


We lifted a partial function to filter a stream in the above example. We can extend this approach further and lift a ⊥-function, i. e. a function whose domain and co-domain are extended with the special symbol ⊥ representing the absence of a value. (See Chapter 2 for the formal definition of ⊥-functions.) The **lift** operator passes ⊥ to this function for streams without a corresponding event. Thus, the function can explicitly handle the absence of a corresponding event on the input streams. The function is called for every timestamp for which there is at least one event on any of the input streams.

Example 3.6 (Handling Absence of Events)**.** With this approach we can define a ⊥-function $f\colon \mathbb{D} \times \mathbb{D} \rightarrowtail \mathbb{D}$ which passes through any input with a precedence to the first argument if both arguments are present:

$$f(a, b) = \begin{cases} a & \text{if } a \neq \bot, \\ b & \text{otherwise.} \end{cases}$$

By lifting such a function we can merge the events of two streams with a precedence to the first stream if both streams have a simultaneous event:



The operator **lift**(f) merges the events of multiple streams with precedence to earlier arguments in those cases where multiple streams contain simultaneous events. This common operation is defined as the derived operator merge in Section 3.3.

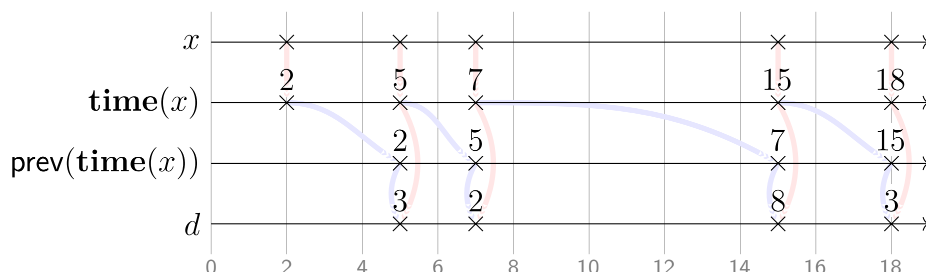
3.1.4. Timestamps, Previous Events and Event Creation

So far, we have discussed how to lift functions and synchronise events of different streams. Another important aspect of stream processing is providing access to previous events. The following examples combine this with explicitly accessing the timestamps of events. In TeSSLa, there are no operators to check properties over the event's timestamps. Instead, the **time** operator can be used to retrieve an event where the event's value is set to its timestamp. Thus we can reuse the same operators which were already introduced to manipulate event's values to perform computations on event's timestamps:

Example 3.7 (Accessing Timestamps And Previous Events)**.** The following specification computes for every incoming event the time passed since the last event. The prev operator gives access to the previous event on the given stream by shifting the event's values all by one event and removing the first event on the stream.

$$d = \mathbf{time}(x) - \mathsf{prev}(\mathbf{time}(x))$$

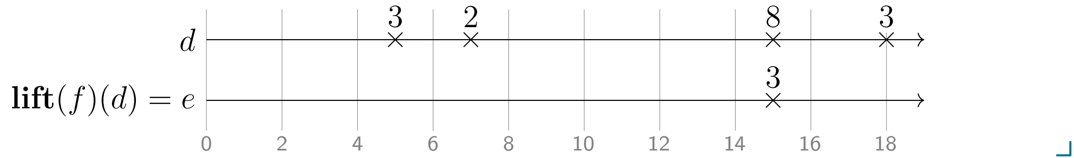
In the visualisation, we can see how prev accesses the previous event. The light blue and light red arrows again indicate the data flow. The different colours are only used to increase the readability.



Assume that we want to check that at least every 5 time units that there is an event on x. Such a check can be realised by taking the stream d and filtering it for events with a value greater than 5. If we use a function $f\colon \mathbb{T} \rightarrowtail \mathbb{T}$ which maps every value greater than 5 to the amount how much it exceeds 5 we get the time this event is too late to meet the deadline:

$$f(x) = \begin{cases} x - 5 & \text{if } x > 5, \\ \bot & \text{otherwise.} \end{cases}$$

In the visualisation we can see how the event on x at timestamp 15 is 3 time units too late to meet the criterion:



We have seen in the previous examples that timestamps in a stream serve three different purposes:

1. Timestamps of the events in a stream indicate the order of the events in that stream. They are strictly increasing and can be accessed and compared to retrieve the relative order of the events in that stream.
2. Timestamps of different streams indicate the relative order of events across all streams. They are used to synchronise the events of different streams.
3. The timestamps may represent the order of events and the exact time the event occurred. In those cases, one can use the timestamp to compute the relative time passed between events or check that absolute time lies within certain boundaries.

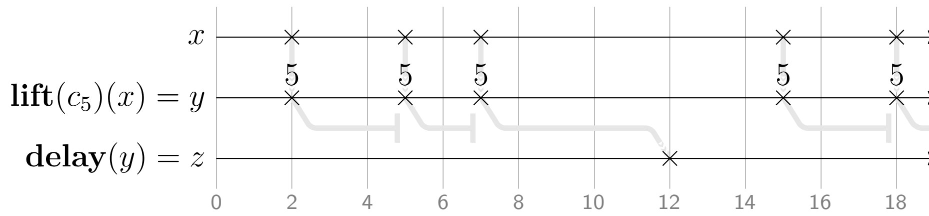
All our TeSSLa examples so far have been *timestamp conservative*, i. e. the derived streams contained only timestamps which already occurred in the input streams. See Definition 3.95 in Section 3.6 later in this chapter for a formal definition. In the previous example, we detected events that violated the deadline of 5 time units. Because of the timestamp conservatism, we can only detect this violation when the next event appears. In general, we already know 5 time units after the event if the deadline was met or not. The **delay** operator is used to introduce events at arbitrary new timestamps. It allows us to formulate a similar specification that reports the error earlier.

Example 3.8 (Delaying Events)**.** Consider the following specification

$$y = \mathbf{lift}(c_5)(x)$$
$$z = \mathbf{delay}(y)$$

with the function $c_5 \colon \mathbb{D} \to \{5\}$ on the data domain mapping every input to the output 5.

Using the same input stream x as in the previous example we get the following visualisation:



The **delay** operator takes a delay and generates an event without data after the given delay has passed.

The light grey arrows do not indicate data flow in this case because the event's data values are ignored. It still represents a causal relationship between the events because the events on the delayed stream only occur because of the events on x.

The operator does not keep track of arbitrarily many delays. Instead, every new delay resets previous delays. This behaviour keeps the semantics of the basic operators as simple as possible. Other possible definitions and their drawbacks are discussed in Section 3.4.5.

The diagram shows not all events of the streams: The stream z must continue with more events after timestamp 18. The cut light-grey arrow indicates this continuation in the diagram. Either an event on x cancels the currently active delay, or there will be an event on z at timestamp $18 + 5 = 23$. ⌟

3.1.5. Aggregating Data Along the Streams

Finally, we discuss TeSSLa's ability to aggregate data along a stream. So far, only examples referring to previous events using prev were considered. The prev operator can be nested, but it can only refer back to a finite number of events. It cannot aggregate data over all the events that occurred so far on a stream.

Such aggregations can be expressed with recursive definitions, i. e. a stream defined in terms of itself.

Example 3.9 (Recursive Aggregating Sum)**.** The following specification sums up the values of all events which occurred so far on the input stream x:

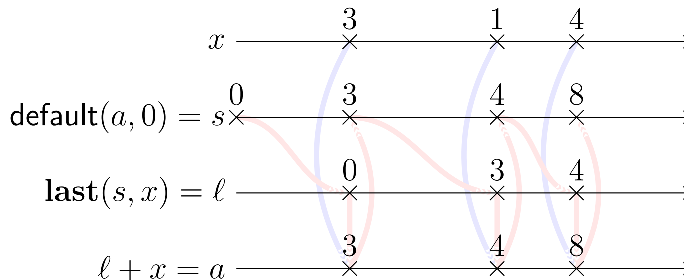
$$\ell = \mathbf{last}(\mathsf{default}(\ell + x, 0), x)$$

In order to name the subexpressions we rewrite this specification as follows:

$$\ell = \mathbf{last}(s, x)$$
$$a = \ell + x$$
$$s = \mathsf{default}(a, 0)$$

The recursive definition uses the operator **last** to refer to a previous event on a stream. Compared to prev this operator takes an additional parameter which is required to define where the events are supposed to be. In this example, we derive a new stream s with a recursive equation saying that the next event of s should be based on the last event of s. The prev operator cannot be used in such a recursive fashion because it lacks additional parameters. Without it, there would be no unique solution to the equation because there is no minimal distance between events on a stream.

The addition $\ell + x$ takes the last value on s and adds the current value on x to it. With ℓ having an event for every event on x, this sums up the event's values on x. The default operator is needed to define a base case for the recursion. Otherwise, there would never be a previous event on s to whose value the **last** operator could refer. The default adds an event with value 0 at the timestamp 0.



The light red data flow arrows show how the aggregation starts with 0 at timestamp 0. Then the **last** is triggered with every event on x. It reproduces the last known value on s at the timestamp of the event on x. The current value of x – shown as a light blue data flow arrow – is added to this last value of s. The default has no effect for later timestamps. It is only included to provide the default value for timestamp 0.

To conclude this set of motivating examples, we will consider two final examples, which combine recursive equations and event generation using the **delay** operator. The following example creates an event every 2 time units.

Example 3.10 (Period)**.** The following specification does not use any input streams. The stream d is only defined in terms of its own:

$$d = \textbf{delay}(\mathsf{const}(2, \mathsf{merge}(d, \textbf{unit})))$$

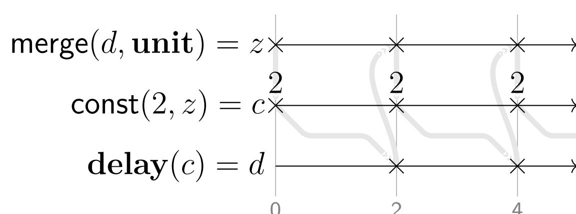
As before, we rewrite the specification in order to name the subexpressions. We get the three streams c, d and z that are derived from each other:

$$c = \mathsf{const}(2, z)$$

$$d = \mathbf{delay}(c)$$
$$z = \mathsf{merge}(d, \mathbf{unit})$$

The above specification takes every event on z and maps its value to a constant 2. The events with value 2 are then used as input for a **delay**. So with any first event on z, it will be repeated every 2 time units forever. Such a first event is introduced into the recursion using the merge with the stream **unit** which contains a single event at timestamp 0 without a value.

Since there is no input stream, the following stream visualisation is not only an exemplary solution but the only possible solution for this equation system:



The stream continues with this pattern infinitely because every event causes another event 2 time units later. ⌟

The following example extends the event generation by making the period configurable. Instead of using a fixed period of 2, an input stream specifies the period:

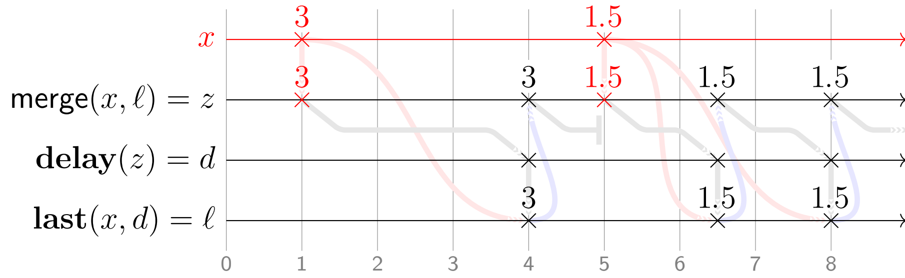
Example 3.11 (Variable Frequency Period)**.** The input stream x contains events whose values specify the period from that point on. The stream z contains an event with that value repeated with the given period:

$$z = \mathsf{merge}(x, \mathbf{last}(x, \mathbf{delay}(z)))$$

As before we rewrite the specification in order to name the subexpressions:

$$d = \mathbf{delay}(z)$$
$$\ell = \mathbf{last}(x, d)$$
$$z = \mathsf{merge}(x, \ell)$$

The main difference in comparison to the previous example is that we use the **last** instead of the const to set a value to the events generated by the **delay**. The merge starts the recursion, but this time we do not start with a fixed event but with the first event on x. The following visualisation shows an exemplary execution of this specification:



The red 1.5 and the black 1.5 indicate the same value. They are only coloured differently to indicate the source of the event: The red ones are immediately merged into the stream from the input stream x, and the black ones are reproduced by the **last** after the delay. Note how new ones overwrite existing delays. The **delay** operator can only handle a single currently active delay.

Without any further event on the input stream x, the pattern of events with the value 1.5 every 1.5 time units would continue indefinitely. Other than the previous example, the periodic event pattern generated by this specification can be interrupted by events on the input stream x: An event with value 0 on x would produce a single output event with that value and stop the periodical output until a new event on x with a positive value starts it again. ⌟

3.2. Semantics

In this section, the formal syntax and semantics of TeSSLa are presented. A similar semantics is given in [Sch20] as a didactic intermediate step which is replaced with the actual semantics later on. We reproduce the TeSSLa semantics in this thesis with some adjustments regarding the presentation and use it as a basis for the following semantics introduced later. Note that the TeSSLa semantics given in [CHL+18] are based on an extended stream model and will be shown in Chapter 6 to be an abstraction of the monitoring semantics introduced in Section 3.5.

We start by formally defining the concept of streams and discuss the Zenoness of streams. Next, we will give definitions for the syntax of TeSSLa as an equation system over streams. Then the semantics of TeSSLa will be defined as the solution of such an equation system. Only the basic operators are defined explicitly. The other operators are given as syntactic sugar in the next section. Finally, some properties regarding the equivalence of TeSSLa specifications, their well-definedness and their dependency graphs are discussed.

3.2.1. Streams

As discussed in the introduction of this thesis, there are many different ways to represent streams. This thesis considers streams with explicitly timestamped events over a continuous time domain. First, we formalise the idea of a time and data domain. The notation is based on [CHL+18]:

Definition 3.12 (Time Domain)**.** We call a totally ordered semiring $(\mathbb{T}, 0, 1, +, \cdot, \leq)$ a *time domain* if it is positive, i. e. , $\forall t \in \mathbb{T} \colon 0 \leq t$. ⌟

In the following, we always assume $\mathbb{T}$ to be the time domain if not explicitly stated otherwise.

Definition 3.13 (Data Domain)**.** A *data domain* is a set. ⌟

Any arbitrary set can be a data domain. However, it is often convenient for actual applications to use data domains that are time domains, too. A more detailed discussion about time and data domains can be found in Section 3.4.

We use the symbol $\mathbb{D}$ to denote an arbitrary data domain.

We formalise streams as finite or infinite sequences of events. Each event is represented as a tuple of a timestamp and a data value. The notation for streams is based on [Sch20]:

Definition 3.14 (Stream)**.** The set $\mathcal{S}_{\mathbb{D}} \subset (\mathbb{T} \times \mathbb{D})^{\infty}$ of all *streams* over a time domain $\mathbb{T}$ and a data domain $\mathbb{D}$ consists of finite or infinite sequences

$$s = \langle (t_0, a_0), (t_1, a_1), \ldots \rangle \in \mathcal{S}_{\mathbb{D}},$$

where the timestamps are strictly increasing, i. e.

$$\forall\, 0 < i < |s| \colon t_{i-1} < t_i.$$

A tuple $(t, d) \in \mathbb{T} \times \mathbb{D}$ is called an *event* and we say s *contains* (t, d) if the tuple occurs in the sequence s. ⌟

In the following, we abuse notation and indicate the Cartesian product $\mathbb{D}_1 \times \mathbb{D}_2 \times \ldots \times \mathbb{D}_n$ of n potentially different data domains with the notation $\mathbb{D}^n$. In the same way, we indicate the Cartesian product $\mathcal{S}_{\mathbb{D}_1} \times \mathcal{S}_{\mathbb{D}_2} \times \ldots \times \mathcal{S}_{\mathbb{D}_n}$ of streams over these data domains with the notation $\mathcal{S}^n_{\mathbb{D}}$.

Definition 3.15 (Timestamps of a Stream)**.** For a stream $s \in \mathcal{S}_\mathbb{D}$ we define $T(s) \subseteq \mathbb{T}$ to be the set of timestamps carrying events in s:

$$T(s) := \{t \in \mathbb{T} \mid s \text{ contains } (t, d) \text{ with } d \in \mathbb{D}\}.$$

We further define $T(\boldsymbol{s}) := \bigcup_{1 \leq i \leq k} T(s_i)$ for the timestamps of all events occurring in a tuple $\boldsymbol{s} \in \mathcal{S}_\mathbb{D}^n$ of streams. ⌟

As discussed in the introduction of this chapter, we do not define a prefix or refinement relation on streams. The following example illustrates this:

Example 3.16 (Stream)**.** Let the time domain be $\mathbb{R}^{\geq 0}$. Then consider the stream $s = \langle(2, 17)\rangle \in \mathcal{S}_\mathbb{N}$ which consists of one event at timestamp 2 with the value 17. Intuitively this stream explicitly covers the entire time domain $\mathbb{R}^{\geq 0}$, i. e. s does not only encode the event at timestamp 2 but also the absence of any events at other timestamps $t \in \mathbb{R}^{\geq 0} \setminus \{2\}$. If we use s in $\mathbf{last}(s, r)$ then for every event on r with a timestamp greater than 2, we get the value 17 because there is no other event following that first event on s. ⌟

We will introduce the concept of monitoring streams in Section 3.5 which introduces concepts on how to extend monitoring streams and defines a refinement relation on them.

There are three ways in which streams can be considered infinite:

1. Streams are always considered fully known, i. e. they are infinite in the sense that there is no possible extension of a stream.
2. Streams can have infinitely large timestamps, i. e. they have infinitely many events without a maximal timestamp.
3. Streams can have infinitely many events whose timestamps are not becoming infinitely large.

The third case is known in the literature as *Zeno behaviour* [Lam02, Mos07, ZJLS00], named after the Greek philosopher Zeno who came up with the paradox that an arrow will never reach its target if it first flies half of the distance to the target, then an additional quarter, then an additional eight and so on:

Definition 3.17 (Zeno [Lam02, Mos07, ZJLS00])**.** An stream $s \in \mathcal{S}_\mathbb{D}$ with infinitely many events is called to be *Zeno* if the supremum of its timestamps $\sup(T(s))$ exists. We say s has *Zeno behaviour* at $\sup(T(s))$. ⌟

Example 3.18 (Zeno Stream)**.** As an example for a Zeno stream consider the stream

$$\left\langle (1, \square), \left(1 + \frac{1}{2}, \square\right), \left(1 + \frac{2}{3}, \square\right), \left(1 + \frac{3}{4}, \square\right), \ldots \right\rangle \in \mathcal{S}_{\mathbb{D}}.$$

Its n-th timestamp is given by

$$t_n = 2 - \frac{1}{1+n}.$$

All timestamps in this infinite sequence of timestamps are strictly lower than 2 because every timestamp in the sequence is exclusively between the previous timestamp and 2. ⌟

All finite streams are, by definition, non-Zeno. Infinite streams with a smallest delta between the timestamps are also non-Zeno, but there are non-Zeno streams without a smallest timestamp delta:

Example 3.19 (Non-Zeno Stream)**.** As an example for a non-Zeno stream consider the stream

$$\left\langle (1, \square), \left(1 + \frac{1}{2}, \square\right), (2, \square), \left(2 + \frac{1}{3}, \square\right), (3, \square), \left(3 + \frac{1}{4}, \square\right), \ldots \right\rangle \in \mathcal{S}_{\mathbb{D}}.$$

Its n-th timestamp is given by

$$t_n = \begin{cases} m & \text{if } n \text{ is even,} \\ m + \frac{1}{1+m} & \text{otherwise.} \end{cases}$$

with

$$m = 1 + \left\lfloor \frac{n}{2} \right\rfloor.$$

The timestamps in this infinite sequence rise beyond any boundary, so this stream does not show Zeno behaviour. However, the delta between a timestamp at an even position and the next timestamp decreases below every boundary. ⌟

Note that a stream's timestamps can at most converge towards one limit. Without further restrictions, a set of timestamps could contain multiple intervals with finite boundaries, containing infinitely many timestamps. Even an arbitrary sequence of timestamps could diverge but stay below a finite boundary, e. g. by alternating between two converging sequences. However, a sequence of strictly increasing timestamps can either converge towards a limit or rise beyond any limit. For example, the Zeno stream shown in the examples above converges towards (but never reaches) timestamp 2.

Definition 3.20 (Limit of a Stream)**.** Let $s \in \mathcal{S}_{\mathbb{D}}$ be a stream and $\langle t_0, t_1, \ldots \rangle$ the ordered sequence of its timestamps $T(s)$. If this sequence converges towards a finite limit we call

$$\operatorname{limT}(s) = \lim_{i \to \infty} t_i$$

the *limit* of s. Otherwise we define

$$\operatorname{limT}(s) = \infty.$$

Although defined as a sequence, a stream can also be seen as a function, too:

Definition 3.21 (Functional View of a Stream)**.** Let $s \in \mathcal{S}_{\mathbb{D}}$ be a stream. Then $f_s \colon \mathbb{T} \to \mathbb{D}_\perp$ is the streams' *functional view* with

$$f_s(t) = \begin{cases} d & \text{if } s \text{ contains } (t, d), \\ \perp & \text{otherwise.} \end{cases}$$

The function f_s maps a timestamp t to a value d if s has an event with value d at time t. All timestamps t without an event at t are mapped to $\perp$. The domain of the function is always the time domain $\mathbb{T}$, which reflects the fact that there is no possible extension of a stream.

If the usage is clear from the context, we use s to refer to f_s.

This functional view is beneficial for the definition of the TeSSLa operators below: They are given by specifying their generated stream as a function mapping timestamps to values. For the formal definition of a stream, we rely on sequences in order to implicitly derive the following property: There is one first event, and every event except the first has a unique predecessor. As a result, a stream's timestamps can at most converge towards one limit, with all timestamps being strictly lower than the limit.

3.2.2. Syntax

Syntactically a TeSSLa specification is an equation system that derives new streams from the given input streams:

Definition 3.22 (TeSSLa Syntax)**.** TeSSLa has the *basic operators* **unit**, **time**, **lift**, **last** and **delay**. A *TeSSLa expression* e over these operators is given by the following grammar:

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

with x being variables representing streams and f being identifiers representing arbitrary functions on data domains.

A *TeSSLa specification* φ consists of equations of the form

$$x := e$$

with x being variables representing streams and e being TeSSLa expressions.

We distinguish between *free* and *bound stream variables*. Bound variables are used on the left-hand side of precisely one equation in φ, and free variables are only used in TeSSLa expressions. ⌟

The free streams are the specification's input streams, and the bound streams are derived streams.

3.2.3. Semantics

The semantics of a TeSSLa specification φ are given as a semantics function $\boldsymbol{f}_\varphi$ mapping a tuple of free input streams to a tuple of derived bound streams. The bound streams are derived as a solution of the equations in φ:

Definition 3.23 (TeSSLa Semantics [Sch20])**.** Let φ be a TeSSLa specification with k free stream variables and n bound stream variables. The semantics interprets the equations of φ as functions $f_i \colon \mathcal{S}_{\mathbb{D}}^k \times \mathcal{S}_{\mathbb{D}'}^n \to \mathcal{S}_{\mathbb{D}'_i}$ mapping all free streams $\boldsymbol{y} \in \mathcal{S}_{\mathbb{D}}^k$ and all bound streams $\boldsymbol{z} \in \mathcal{S}_{\mathbb{D}'}^n$ to a stream $z_i \in \mathcal{S}_{\mathbb{D}'_i}$:

$$z_i = \boldsymbol{f}_i(\boldsymbol{y})(\boldsymbol{z})$$

The functions f_i are compositions of the TeSSLa operators whose semantics are given below. In combination we get $\boldsymbol{f} \colon \mathcal{S}_{\mathbb{D}}^k \times \mathcal{S}_{\mathbb{D}'}^n \to \mathcal{S}_{\mathbb{D}'}^n$ being the tuple of all f_i and thus the following equation:

$$\boldsymbol{z} = \boldsymbol{f}(\boldsymbol{y})(\boldsymbol{z})$$

The semantics of φ is a function $\boldsymbol{f}_\varphi \colon \mathcal{S}_{\mathbb{D}}^k \to \mathcal{S}_{\mathbb{D}'}^n$ given as the fixed point $\boldsymbol{z}$ of $\boldsymbol{f}$:

$$\boldsymbol{f}_\varphi(\boldsymbol{y}) := \boldsymbol{z} \text{ such that } \boldsymbol{f}(\boldsymbol{y})(\boldsymbol{z}) = \boldsymbol{z}.$$ ⌟

In general, this fixed point is not unique, and hence the semantics is not well-defined. We will introduce the concept of well-formed TeSSLa specifications in Section 3.2.4 and show in Section 3.5 that the fixed point is unique for those specifications.

In the following, we define the semantics of the TeSSLa operators by defining the resulting stream $z \in \mathcal{S}_{\mathbb{D}}$ using its view as a function. We implicitly quantify the timestamp $t \in \mathbb{T}$ in the defining equations over all $t < \mathrm{limT}(z)$. Although the stream z is given as a function, one always has to keep in mind that it is a stream, i. e. we do not consider arbitrary functions from the time domain to the data domain, but only those functions that represent streams. This restriction becomes especially important in the case of compositions $o \circ p$ of two TeSSLa operators o and p: The output of the first operator o is a stream that is then processed by the second operator p.

Definition 3.24 (Semantics of the Operator **unit** [Sch20])**.** The operator **unit** is defined as follows:

$$\mathbf{unit} \in \mathcal{S}_{\mathbb{U}} \text{ with } \mathbf{unit} := z,$$

where the stream z defined by

$$z(t) = \begin{cases} \square & \text{if } t = 0, \\ \bot & \text{otherwise.} \end{cases}$$

⌟

The **unit** operator is a pre-defined stream with a single event at timestamp 0. The operator is required to introduce streams with an event at timestamp 0 independently from the input streams. For a further discussion on the set of basic operators, see Section 3.4.2.

We use the unit type $\mathbb{U} = \{\square\}$ to encode streams of events without values as streams whose events all have the single value $\square$. Every stream $\mathcal{S}_{\mathbb{D}}$ over an arbitrary data domain $\mathbb{D}$ can be implicitly converted into a stream over $\mathcal{S}_{\mathbb{U}}$ by ignoring the event's values.

Not that the unit stream can be stated explicitly as $\mathbf{unit} = \langle(0, \square)\rangle$, too, but for the sake of a uniform presentation of all TeSSLa operators, the functional view was used in the above definition.

Definition 3.25 (Semantics of the Operator **time** [Sch20])**.** The operator **time** is defined as follows:

$$\mathbf{time} \colon \mathcal{S}_{\mathbb{D}} \to \mathcal{S}_{\mathbb{T}} \text{ with } \mathbf{time}(s) := z,$$

where the stream z defined by

$$z(t) = \begin{cases} t & \text{if } t \in T(s), \\ \bot & \text{otherwise.} \end{cases}$$

⌟

The **time** operator maps the event's values to their timestamps.

Definition 3.26 (Semantics of the Operator **lift** [Sch20])**.** The operator **lift** is defined for any well-formed $\bot$-function $f \colon \mathbb{D}^n \rightarrowtail \mathbb{D}$ as follows:

$$\textbf{lift} \colon (\mathbb{D}^n \rightarrowtail \mathbb{D}) \to (\mathcal{S}_{\mathbb{D}}^n \to \mathcal{S}_{\mathbb{D}}) \text{ with } \textbf{lift}(f)(s_1, \ldots, s_n) := z,$$

where the stream z is defined by

$$z(t) = f(s_1(t), \ldots, s_n(t)).$$

⌟

The **lift** operator lifts a function f on the data domain to a function on streams over these data domains by applying f to the stream's values for every timestamp. As defined in the preliminaries, a $\bot$-function's domain and co-domain is extended with $\bot$. The special value $\bot$ indicates the absence of an event. The $\bot$-function f must be well-formed, i. e. if all inputs are $\bot$, the output must be $\bot$, too. In other words: The function f is not allowed to produce additional events if its input does not contain any event. This approach was motivated in detail in Section 3.1.3.

Definition 3.27 (Semantics of the Operator **last** [Sch20])**.** The operator **last** is defined as follows:

$$\textbf{last} \colon \mathcal{S}_{\mathbb{D}} \times \mathcal{S}_{\mathbb{U}} \to \mathcal{S}_{\mathbb{D}} \text{ with } \textbf{last}(v, r) := z,$$

where the stream z defined by

$$z(t) = \begin{cases} d & \text{if } t \in T(r) \land \exists\, t' < t \colon \mathit{isLast}(t, t', v, d), \\ \bot & \text{otherwise,} \end{cases}$$

with the following auxiliary definition:

$$\mathit{isLast}(t, t', v, d) := v(t') = d \land \forall\, t'' \colon t' < t'' < t \Rightarrow v(t'') = \bot.$$

⌟

The **last** operator takes two streams v and r. We refer to v as its *value stream* and to r as its *trigger stream*. It outputs an event with the previous value on v for every event on r. The predicate $\mathit{isLast}(t, t', v, d)$ for two timestamps $t, t' \in \mathbb{T}$, a stream

$v \in \mathcal{S}_{\mathbb{D}}$ and a value $d \in \mathbb{D}$ holds if (t', d) is the last event on v until timestamp t (excluding t).

The following definition of the **delay** operator differs from the one given in [Sch20] because it only takes one argument. We show in Section 3.6 that we gain the full expressiveness with this simpler version. The need for additional arguments of the operator is discussed further in Section 6.2.1.

Definition 3.28 (Semantics of the Operator **delay**). The operator **delay** is defined as follows:

$$\mathbf{delay}\colon \mathcal{S}_{\mathbb{T}} \to \mathcal{S}_{\mathbb{U}} \text{ with } \mathbf{delay}(d) := z,$$

where the stream z defined by

$$z(t) = \begin{cases} \square & \text{if } \exists\, t' < t\colon \mathit{set}(d, t, t') \wedge \mathit{noreset}(d, t', t), \\ \bot & \text{otherwise,} \end{cases}$$

with the following auxiliary definitions:

$$\begin{aligned} \mathit{set}(d, t, t') &:= d(t') = t - t' \text{ and} \\ \mathit{noreset}(d, t, t') &:= \forall\, t''\colon t < t'' < t' \Rightarrow d(t'') = \bot. \end{aligned}$$

⌟

The predicate $\mathit{set}(d, t, t')$ for a stream $d \in \mathcal{S}_{\mathbb{T}}$ and two timestamps $t, t' \in \mathbb{T}$ holds if the stream d has an event at timestamp t' whose value is the delta until the current timestamp t. The predicate $\mathit{noreset}(d, t, t')$ for a stream $d \in \mathcal{S}_{\mathbb{T}}$ and two timestamps $t, t' \in \mathbb{T}$ holds if the stream d does not contain an event between the timestamps t' and the current timestamp t (both exclusive).

The **delay** operator takes one stream d, called its *delay stream*. It emits a unit event in the resulting stream after the delay passes. Delays are not accumulated. A new delay resets a currently active one. A delay of 0 only resets the currently active delay without setting a new one.

Now we can conclude: For non-recursive stream definitions, the semantics are immediately given by applying the operators to the streams according to the equations. For recursive stream definitions, the semantics is given as the fixed point. So far, we have no way to compute the fixed point or even to assure that a fixed point always exists. Both problems will be addressed in Section 3.5.

3.2.4. Properties

We now consider some simple properties of TeSSLa specifications starting with the equivalence of specifications defined as usual:

Definition 3.29 (Equivalence of TeSSLa Specifications)**.** Two TeSSLa specifications φ and ψ are equivalent if their semantics functions $\boldsymbol{f}_\varphi$ and $\boldsymbol{f}_\psi$ are equivalent:

$$\varphi \equiv \psi :\Longleftrightarrow \boldsymbol{f}_\varphi \equiv \boldsymbol{f}_\psi. \qquad \lrcorner$$

With this definition, two specifications are only equivalent if, for all free streams, all of their derived streams are equivalent. Although the TeSSLa semantics introduced in this thesis does not have a formal definition of output streams, a projection of the semantics function onto certain elements of the tuple is a straightforward extension. The fixed point must be computed over the tuple of all bound streams, but afterwards, one can remove internal streams from the tuple of streams.

In the following we sometimes use expressions over streams as specification. For example the notation

$$\mathbf{last}(x, y) \equiv \mathbf{last}(x, \mathbf{last}(\mathbf{unit}, y))$$

is short for

$$\varphi \equiv \psi$$

with the implicitly defined specifications φ and ψ: Both specifications have the free input streams $x, y \in \mathcal{S}_\mathbb{D}$ and the bound derived stream $z_\varphi \in \mathcal{S}_\mathbb{D}$ or $z_\psi \in \mathcal{S}_\mathbb{D}$, respectively, given by

$$\begin{aligned} z_\varphi &= \mathbf{last}(x, y) \\ z_\psi &= \mathbf{last}(x, \mathbf{last}(\mathbf{unit}, y)). \end{aligned}$$

It is important for the equivalence to hold that ψ has only one derived variable. Assume the slightly different specification ψ' with the internal derived stream $z_1 \in \mathcal{S}_\mathbb{D}$ and the output derived stream $z_{\psi'} \in \mathcal{S}_\mathbb{D}$ given by

$$\begin{aligned} z_1 &= \mathbf{last}(\mathbf{unit}, y) \\ z_{\psi'} &= \mathbf{last}(x, z_1). \end{aligned}$$

The equivalence $\psi \equiv \psi'$ only holds if we ignore the internal stream z_1 and implicitly project the semantics function $\boldsymbol{f}_{\psi'}$ to the output stream $z_{\psi'}$.

Next, we define the dependency graph and the related flow graph of a TeSSLa specification. For the dependency graph we require the specification to be flat:

Definition 3.30 (Flat [CHL+18]). We call a TeSSLa specification φ *flat* if for every equation $x := e$ of φ the TeSSLa expression e contains exactly one of the TeSSLa operators **unit**, **time**, **lift**, **last** or **delay**. ⌟

In a flat TeSSLa specification, every derived bound stream corresponds to a TeSSLa operator used to define this stream.

By adding additional bound variables, every TeSSLa specification can be transformed into an equivalent flat TeSSLa specification.

Definition 3.31 (Dependency and Flow Graph [CHL+18]). The *dependency graph* of a TeSSLa specification φ is the directed multi-graph $G = (V, E)$ of nodes $V = \{x_1, \ldots, x_n\}$. For every equation $x_i := f_i(\boldsymbol{x})$ in φ the graph contains the edge (x_i, x_j) iff f_i depends on x_j.

The dependency graph with reversed edges is called *flow graph*. ⌟

For a given TeSSLa specification φ, we refer to the dependency graph of an equivalent flat TeSSLa specification as the dependency graph of φ if that is clear from the context.

While an edge in the dependency graph represents a dependency between two operators, the reversed edge in the flow graph represents the data flow between the operators.

So far, no formal restrictions on cyclic definitions of bound streams in a TeSSLa specification are defined. However, the semantics function $\boldsymbol{f}_\varphi$ might not be well-defined on arbitrary specifications if the equation system has multiple solutions. For example, for the specification $x := x$ every stream $x \in \mathcal{S}_\mathbb{D}$ is a fixed point.

Definition 3.32 (Well-Formed TeSSLa Specification [CHL+18]). We call a TeSSLa specification φ *well-formed* if every cycle of the dependency graph contains at least one *delayed*-labelled edge. We label edges corresponding to the first argument of **last** or **delay** with *delayed*. ⌟

We will show in Section 3.5 on monitoring semantics that the semantics function $\boldsymbol{f}_\varphi(\boldsymbol{y}) = \boldsymbol{z}$ such that $\boldsymbol{z} = \boldsymbol{f}(\boldsymbol{y})(\boldsymbol{z})$ of a TeSSLa specification φ is well-defined if φ is well-formed because then the fixed point $\boldsymbol{z}$ is unique.

Example 3.33 (Dependency Graph of a Well-Formed TeSSLa Specification). Reconsider the TeSSLa specification φ from Example 3.11 (Variable Frequency Period)

in Section 3.1.5 with the free (input) stream $x \in \mathcal{S}_{\mathbb{R}\geq 0}$ and the bound (derived) streams $d \in \mathcal{S}_{\mathbb{U}}$ and $z, \ell \in \mathcal{S}_{\mathbb{R}\geq 0}$:

$$z = \mathsf{merge}(x, \ell)$$
$$\ell = \mathbf{last}(x, d)$$
$$d = \mathbf{delay}(z)$$

The dependency graph of φ can be drawn as shown in Figure 3.1. Bound streams are depicted as rectangles and free streams with rounded corners. The only cycle in the graph contains a delayed-labelled edge, making the specification well-formed. Informally one can say that the cycle is interrupted by the **delay** operator: An input event of the **delay** cannot produce an output event with the same timestamp. Remember that values of 0 are defined to reset the delay. The **last** operator does not interrupt the cycle in the same way in this example because its trigger input, i. e. its second argument, is used in the cycle. The delayed-labelled edge of the **last** is going to the input stream x. ⌟

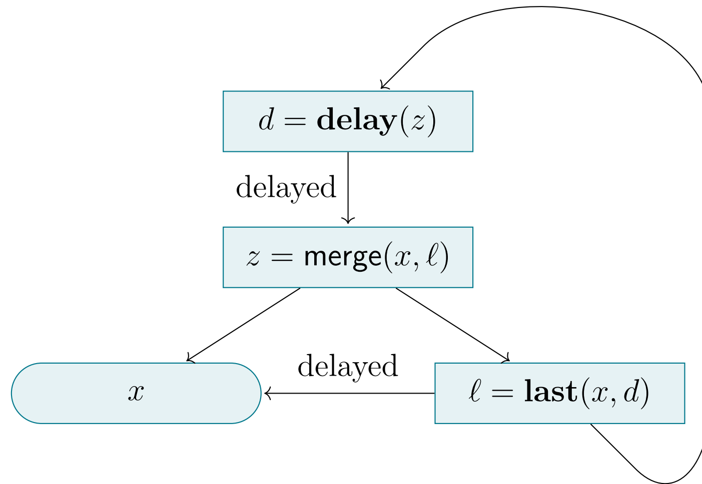


Figure 3.1.: Dependency graph of a well-formed specification. (Example 3.33)

Example 3.34 (Dependency Graph of a Not Well-Formed TeSSLa Specification)**.** As a counterexample for a well-formed specification consider the following specification with the streams $z, \ell, i \in \mathcal{S}_{\mathbb{Z}}$:

$$\ell = \mathbf{last}(z, z)$$
$$i = \mathbf{lift}(inc)(\ell)$$
$$z = \mathsf{merge}(i, 0)$$

Compared to the previous example, this specification lacks any free stream, i. e. there is no input stream. The dependency graph shown in Figure 3.2 contains two cycles: One cycle contains an edge corresponding to the first argument of **last** and hence is fine, but the other cycle (depicted in red in the diagram) contains only an edge corresponding to the second argument of **last** and thus no delayed-labelled edge.

The specification describes a stream z that has an event with value 0 at timestamp 0. With every further event of z, the event's value is incremented by 1. The specification, however, does not define the timestamps of these events because the **last** is triggered by z, which is itself again defined by the **last**. In synchronous stream processing languages over a discrete time domain, like e. g. [DSS+05], specifications like this are allowed and uniquely define a stream because for every timestamp, there is only one possible next timestamp. In TeSSLa, we need additional clarification on the timestamps of the events because the time domain is continuous. This can be an external trigger stream, as in Example 3.9 (Recursive Aggregating Sum) from Section 3.1.5, or a precise definition where additional events are created using a **delay**, as in the previous example.

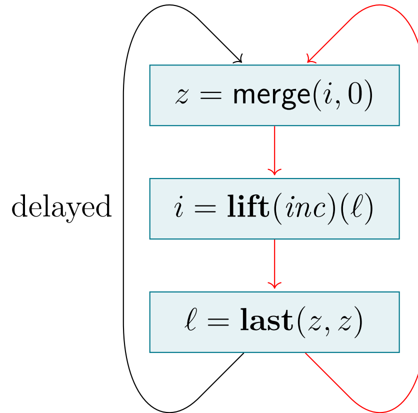


Figure 3.2.: Dependency graph of the specification given in Example 3.34 which is not well-formed.

The above examples illustrate that the semantics is, in general, not well-defined on specifications that are not well-formed. The equation systems can have multiple solutions, and so far, we have not introduced an ordering on these solutions because we have not yet introduced an ordering on the streams defined in this section. All streams are infinitely long, and we do not define any prefix relation on streams. Hence we cannot use, e. g. the least fixed point in the definition of the semantics discussed in this section.

The monitoring semantics introduced in Section 3.5 use monitoring streams that have a notion of progress, i. e. they can represent a prefix with multiple possible continuations. The monitoring semantics is defined as a least fixed point which guarantees a well-defined semantics for arbitrary specifications. The monitoring semantics function maps to a not entirely evaluated output in those cases, where the fixed point is not unique using the semantics defined in this section.

3.3. Common Derived Operators

This section defines some additional TeSSLa operators derived from the basic operators defined in the semantics. These basic operators are building blocks for the definition of more user-friendly and verbose TeSSLa operators. It is one of the fundamental ideas of TeSSLa to have a minimal set of basic operators which form the theoretic core of TeSSLa and many derived operators which simplify the practical usage of TeSSLa.

This section only demonstrates how to derive further TeSSLa operators with the example of some valuable operators. Practical implementations of TeSSLa may contain large standard libraries that can adapt to different application domains.

3.3.1. Operators Derived From lift

The following operators are directly derived from **lift** by passing it a specific function.

Definition 3.35 (Semantics of the Operator nil)**.** The stream $\mathsf{nil} \in \mathcal{S}_\emptyset$ is defined as

$$\mathsf{nil} \equiv \mathsf{lift}(f)(\mathbf{unit})$$

with the function $f\colon \mathbb{U} \rightarrowtail \emptyset$ given by $f(x) = \bot$ for all $x \in \mathbb{U}_\bot$. ⌟

The stream nil is the empty stream without any event.

Definition 3.36 (Semantics of the Operator const)**.** The binary operator $\mathsf{const}\colon \mathbb{D} \times \mathcal{S}_{\mathbb{D}'} \to \mathcal{S}_{\mathbb{D}}$ maps the values of the events on a stream to a constant value:

$$\mathsf{const}(d, r) = \mathbf{lift}(f)(r)$$

with the constant function $f\colon \mathbb{D}' \to \mathbb{D}$ with $f(d') = d$ for all $d' \in \mathbb{D}'$.

The unary operator $\mathsf{const}\colon \mathbb{D} \to \mathcal{S}_{\mathbb{D}}$ maps a value from the data domain to a stream with a single event at timestamp 0 carrying that value:

$$\mathsf{const}(d) \equiv \mathsf{const}(d)(\mathbf{unit})$$

In TeSSLa specifications, we use the convention that a value $d \in \mathbb{D}$ on a data domain $\mathbb{D}$ can be used to represent the stream $\mathsf{const}(d)$.

Definition 3.37 (Semantics of the Operator merge)**.** The n-ary operator $\mathsf{merge}\colon \mathcal{S}_{\mathbb{D}} \times \mathcal{S}_{\mathbb{D}} \times \ldots \times \mathcal{S}_{\mathbb{D}} \to \mathcal{S}_{\mathbb{D}}$ merges all the events from the given streams into a single new stream. In the case of simultaneous events, the operator uses the value of the event coming first in the sequence of operands.

$$\mathsf{merge}(s_1, s_2, \ldots, s_n) \equiv \mathbf{lift}(f_n)(s_1, s_2, \ldots, s_n)$$

with $f_n\colon \mathbb{D}^n \rightarrowtail \mathbb{D}$ given by

$$f_n(d_1, d_2, \ldots, d_n) = \begin{cases} d_1 & \text{if } d_1 \neq \bot, \\ f_{n-1}(d_2, \ldots, d_n) & \text{otherwise.} \end{cases}$$
$$f_1(d) = d$$

For the trivial case of $n = 1$ we get $\mathsf{merge}(s) \equiv s$.

3.3.2. Accessing Previous Values

The operators sync and on for the synchronisation of events were motivated in Section 3.1.2 and Section 3.1.4 introduced how previous events can be accessed directly with the operator prev. These operators are derived as simpler and hence more specific variants of the operator **last**:

Definition 3.38 (Semantics of the Operators prev, sync and on)**.** The operator $\mathsf{prev}\colon \mathcal{S}_{\mathbb{D}} \to \mathcal{S}_{\mathbb{D}}$ maps every event on the given stream to its predecessor on that stream. The operator $\mathsf{sync}\colon \mathcal{S}_{\mathbb{D}} \times \mathcal{S}_{\mathbb{U}}^n \to \mathcal{S}_{\mathbb{D}}$ synchronises the first stream with the events from the other streams. The operator $\mathsf{on}\colon \mathcal{S}_{\mathbb{U}} \times \mathcal{S}_{\mathbb{D}} \to \mathcal{S}_{\mathbb{D}}$ samples the second stream with the events from the first stream.

$$\begin{aligned} \mathsf{prev}(x) &\equiv \mathbf{last}(x, x) \\ \mathsf{sync}(x, \boldsymbol{y}) &\equiv \mathsf{merge}(x, \mathbf{last}(x, \mathsf{merge}(\boldsymbol{y}))) \\ \mathsf{on}(x, y) &\equiv \mathbf{lift}(f)(x, \mathsf{sync}(y, x)) \end{aligned}$$

with $f\colon \mathbb{D}' \times \mathbb{D} \rightarrowtail \mathbb{D}$ given by

$$f(a,b) = \begin{cases} b & \text{if } a \neq \bot, \\ \bot & \text{otherwise.} \end{cases}$$

Note that prev can be nested: We use the notation prev^n to indicate the composition of n prev operators. $\mathsf{prev}^n(x)$ shifts the values of the events of the stream x by n events to the right and removes the n first events from the stream. The parameter n is fixed for every operator prev^n used in a specification. See Section 3.4.5 for a discussion of the drawbacks of an operator prevn which allows access to the n-th previous event where n is dynamically given as stream and how such an operator can be expressed in TeSSLa.

With the above definitions, we have established different applications of the **last** operator:

- Accessing previous events on the same stream with prev,
- sampling events on a stream with triggers on another stream with on,
- synchronising events on a stream with triggers on another stream with sync, which can be seen as a special case of sampling, and
- accessing previous events at a given external trigger in recursions with **last**.

For every event on the trigger stream, the **last** operator produces the last known value of the value stream. The last known value at timestamp t only refers to events that happened strictly before t. This feature of the **last** operator is only needed for the last usage of **last**, i. e. as a way to access previous values in recursions. In fact, in the sampling and synchronising applications, we have to combine **last** and **lift** to get access to all events which happened at or before t. However, as one can see in the above definitions, it is rather straightforward to do so, and it makes theoretical analyses much easier if **last** and **lift** have strictly separated concerns which can then be combined later if needed.

3.3.3. Signal Lift

The signal view of a stream was motivated in Section 3.1.2. The corresponding signal lift operator slift is derived as a combination of **last** and **lift**:

Definition 3.39 (Semantics of the Operator slift)**.** The signal lift operator slift with the signature $(\mathbb{D}^n \hookrightarrow \mathbb{D}) \rightarrow (\mathcal{S}_\mathbb{D}^n \rightarrow \mathcal{S}_\mathbb{D})$ lifts a given partial function $f\colon \mathbb{D}^n \hookrightarrow \mathbb{D}$ on the data domain to streams:

$$\mathsf{slift}(f)(\boldsymbol{d}) \equiv \mathbf{lift}(f)(\boldsymbol{d}')$$

where every d'_i in $\boldsymbol{d}'$ is given by

$$d'_i = \mathsf{sync}(d_i, \boldsymbol{d}).$$

The tuple $\boldsymbol{d}'$ of streams is the synchronised version of $\boldsymbol{d}$ where all streams have events at identical timestamps. If streams in $\boldsymbol{d}$ are missing events at certain timestamps where other streams have events, then those events are added in $\boldsymbol{d}'$ by repeating the previous event on that stream. If we interpret a stream as a piece-wise constant signal, this operation does not change the signal but only its underlying encoding as an event stream. This synchronisation ensures that the function is never called with $\perp$ on any inputs.

The only case where additional events cannot be added is if a stream does not have any previous events, which could be repeated. The signal lift only generates an output event at timestamp t if all input events have an event at t or had an event prior to t. The above definition defines slift only on partial functions. By convention, such a function is then implicitly extended to a conservative $\perp$-function when passing it to the **lift**. This extension leads to the desired behaviour because conservative $\perp$-functions return $\perp$ if any of its inputs are $\perp$.

In TeSSLa specifications, we use the convention that all functions f, which are defined on the data domain, can implicitly be used on streams by replacing f with $\mathsf{slift}(f)$. This convention can be used together with the convention of automatic application of const in order to convert constants on the data domain to streams: By combining both conventions, we can write $y = x + 1$ to define the stream y to have the same events as x, with every event's value being incremented by 1. See Section 3.4.1 for a further discussion of this approach.

The filter operator is a special case of the signal view: It filters a stream of events based on the Boolean value of another stream. The event stream is not synchronised with the condition stream because we do not want to create new events. On the other hand, the condition stream is synchronised with the event stream because we want to interpret the condition as a piece-wise constant signal.

Definition 3.40 (Semantics of the Operator filter)**.** The operator filter with the signature $\mathcal{S}_\mathbb{D} \times \mathcal{S}_\mathbb{B} \rightarrow \mathcal{S}_\mathbb{D}$ filters a given event stream based on the current value of a Boolean stream:

$$\mathsf{filter}(x, c) \equiv \mathbf{lift}(f)(x, \mathsf{sync}(c, x))$$

with the function $f\colon \mathbb{D} \times \mathbb{B} \rightarrowtail \mathbb{D}$ given by

$$f(d,a) = \begin{cases} d & \text{if } a = \mathit{true}, \\ \bot & \text{otherwise.} \end{cases}$$

The lifted function f returns $\bot$ if either of its arguments is $\bot$ or if the second argument is false.

3.3.4. Default Values

If we see a stream as a piece-wise constant signal, the first event of such a stream is essential. Before that first event, the piece-wise constant signal does not exist. Starting with the first event, the value of the piece-wise constant signal is given by the last event on the stream. With the following operator, one can ensure that a stream starts with an event at timestamp 0. If the given stream does not already have an event at timestamp 0, such an event with a given default value is added:

Definition 3.41 (Semantics of the Operator default)**.** The derived TeSSLa operator default: $\mathcal{S}_{\mathbb{D}} \times \mathbb{D}_{\bot} \to \mathcal{S}_{\mathbb{D}}$ is given by:

$$\mathsf{default}(s,d) \equiv \begin{cases} \mathsf{merge}(s, \mathsf{const}(d, \mathbf{unit})) & \text{if } d \neq \bot, \\ s & \text{otherwise.} \end{cases}$$

When passed $\bot$ as default value d the default operator has no effect. This feature is used to simplify more complex compositions in later definitions. We encode this explicitly using the case differentiation for $d = \bot$ because the operator const only accepts actual values of the data domain $\mathbb{D}$.

With the convention of the automatic conversion from values on the data domain to corresponding constant streams, the expression $\mathsf{default}(s,d)$ could be directly written as $\mathsf{merge}(s,d)$, too. The default operator is included here as a simple operator which adds exactly the missing feature to create full expressiveness in Section 3.4.2.

3.3.5. Recursive Equations

A typical recursive equation in TeSSLa requires an initialisation as a base case to get the recursion started, accessing previous values at certain timestamps and a computation that derives a new value from an old value and some current input. Example 3.9 (Recursive Aggregating Sum) from Section 3.1.5 shows such an aggregation that sums up the values of all events on a stream: The initialisation is an event at timestamp 0 with the value 0. We want to access the old value with every event on the input stream, and the computation takes the old value and adds the value of the current event.

The operators fold and reduce simplify specifying such aggregations. They are known from functional programming [Hug89, Hut99], where they are used to express recursions that aggregate over data structures such as lists. In TeSSLa, these operators express recursive specifications that aggregate over all events of a stream. The operator fold takes an initial value and a function combining two values from possibly different data domains. The operator reduce uses the first event's value of the stream as the initial value and thus takes a function combining two values from the same data domain. They are given below as special cases of the more generic aggregation foldn operator, which generalises the operators to multiple input streams:

Definition 3.42 (Semantics of the Operator foldn). The operator foldn with the signature $\mathbb{D}'_\bot \times (\mathbb{D}' \times \prod_{i=1}^{n} \mathbb{D}_i \rightarrowtail \mathbb{D}') \to \prod_{i=1}^{n} \mathcal{S}_{\mathbb{D}_i} \to \mathcal{S}_{\mathbb{D}'}$ folds a well-formed $\bot$-function $f\colon \mathbb{D}' \times \prod_{i=1}^{n} \mathbb{D}_i \rightarrowtail \mathbb{D}'$ over all the event's values of the input streams starting with a default value $a \in \mathbb{D}'_\bot$:

$$\mathsf{foldn}(a, f)(\boldsymbol{x}) = y$$

where y is given by the following equation

$$y = \mathbf{lift}(f)(\mathsf{default}(\mathbf{last}(y, \mathsf{merge}(x)), a), \boldsymbol{x})$$ ⌟

The semantics of the foldn operator is not directly given as a composition of other operators but as a TeSSLa equation. The operator can be seen as a macro that needs to be instantiated before the fixed point used to define the TeSSLa semantics can be determined. Practical implementations of TeSSLa need a way to specify such macros to make full use of user-defined TeSSLa operators.

The fold operator is a special case of the foldn operator. It is restricted to a single input stream and total functions. It always produces an event at timestamp 0. If there is no input event at timestamp 0, then the given start value is used as output at timestamp 0:

Definition 3.43 (Semantics of the Operator fold)**.** The derived TeSSLa operator $\mathsf{fold}\colon \mathbb{D}' \times (\mathbb{D}' \times \mathbb{D} \to \mathbb{D}') \to \mathcal{S}_{\mathbb{D}} \to \mathcal{S}_{\mathbb{D}'}$ folds a function $f\colon \mathbb{D}' \times \mathbb{D} \to \mathbb{D}'$ over all the event's values of a stream $x \in \mathcal{S}_{\mathbb{D}}$ starting with a default value $a \in \mathbb{D}'$:

$$\mathsf{fold}(a, f)(x) \equiv \mathsf{foldn}(a, g)(x)$$

with $g\colon \mathbb{D}' \times \mathbb{D} \rightarrowtail \mathbb{D}'$ given by

$$g(a, x) = \begin{cases} a & \text{if } x = \bot, \\ f(a, x) & \text{otherwise.} \end{cases}$$

⌟

As a direct consequence of the above definitions, we get the following relation: The expression $\mathsf{fold}(a, f)(x)$ defines a stream y which is equivalent to the equation

$$y = \mathsf{default}(f(\mathsf{default}(\mathbf{last}(y, x), a), x), a).$$

For some operations, there is no neutral value a. For example, for a sum or a count, the value 0 is a good neutral value indicating that no data was processed so far, but there is no such value for a minimum or a maximum. One could introduce an artificial neutral value indicating the absence of any actual data or use TeSSLa's ability to express the absence of events to encode this.

The following operator folds a function over all the event's values of a stream and uses the first event on that stream as the initial value. Consequently, the folded function must take two operands of the same type and return a value of that same type.

Definition 3.44 (Semantics of the Operator reduce)**.** The operator reduce with the signature $(\mathbb{D} \times \mathbb{D} \to \mathbb{D}) \to \mathcal{S}_{\mathbb{D}} \to \mathcal{S}_{\mathbb{D}}$ folds a function $f\colon \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ over all the event's values of a stream:

$$\mathsf{reduce}(f)(x) \equiv \mathsf{foldn}(\bot, g)(x)$$

with $g\colon \mathbb{D}' \times \mathbb{D} \rightarrowtail \mathbb{D}'$ given by

$$g(a, x) = \begin{cases} x & \text{if } a = \bot, \\ f(a, x) & \text{otherwise.} \end{cases}$$

⌟

With the operators fold and reduce we can now derive several common aggregation operators as further examples how to use fold and reduce:

Definition 3.45 (Semantics of the Aggregation Operators count, sum, $\mathsf{minimum}$ and $\mathsf{maximum}$)**.** The operator $\mathsf{count}\colon \mathcal{S}_{\mathbb{U}} \to \mathcal{S}_{\mathbb{N}}$ counts the events in the input stream. The operator $\mathsf{sum}\colon \mathcal{S}_{\mathbb{R}} \to \mathcal{S}_{\mathbb{R}}$ sums up the event's values in the input stream. The operator $\mathsf{minimum}\colon \mathcal{S}_{\mathbb{R}} \to \mathcal{S}_{\mathbb{R}}$ computes the minimum of all event's values in the input stream. The operator $\mathsf{maximum}\colon \mathcal{S}_{\mathbb{R}} \to \mathcal{S}_{\mathbb{R}}$ computes the maximum of all event's values in the input stream.

$$\begin{aligned}
\mathsf{count}(x) &\equiv \mathsf{fold}(0, \mathit{inc})(x) \\
\mathsf{sum}(x) &\equiv \mathsf{fold}(0, +)(x) \\
\mathsf{minimum}(x) &\equiv \mathsf{reduce}(\min)(x) \\
\mathsf{maximum}(x) &\equiv \mathsf{reduce}(\max)(x)
\end{aligned}$$

The function $\mathit{inc}\colon \mathbb{Z} \to \mathbb{Z}$ increments an integer, i. e. $\mathit{inc}(i) = i + 1$ for any $i \in \mathbb{Z}$. ⌟

As a final example of how the foldn operator can be used to define arbitrary aggregations over multiple streams, we define a $\mathsf{resetCount}$ operator which extends the count operator with an additional resetting input stream: Every event on that resetting input stream resets the counting output to 0. For simultaneous events on both input streams, the counter is reset to 0 and immediately incremented to 1.

Definition 3.46 (Semantics of the Operator $\mathsf{resetCount}$)**.** The derived TeSSLa operator $\mathsf{resetCount}\colon \mathcal{S}_{\mathbb{U}} \times \mathcal{S}_{\mathbb{U}} \to \mathcal{S}_{\mathbb{N}}$ is given by

$$\mathsf{resetCount}(x, r) \equiv \mathsf{foldn}(0, f)(x, r)$$

with the function $f\colon \mathbb{N} \times \mathbb{U} \times \mathbb{U} \rightarrowtail \mathbb{N}$ given by

$$f(a, x, r) = \begin{cases} 1 & \text{if } x \neq \bot \wedge r \neq \bot, \\ 0 & \text{if } x = \bot \wedge r \neq \bot, \\ a + 1 & \text{if } x \neq \bot \wedge r = \bot, \\ c & \text{if } x = \bot \wedge r = \bot. \end{cases}$$

⌟

Note how $f(0, \bot, \bot) = 0$ causes the foldn to output 0 at timestamp 0 if there is no input event at timestamp 0 on the input streams x or r.

Similarly, the operators sum, $\mathsf{minimum}$ and $\mathsf{maximum}$ could be equipped with a resetting input stream, too.

3.3.6. Generating New Timestamps

In order to delay events with their values by an arbitrary amount of time, one can use the expression $\mathbf{last}(x, \mathbf{delay}(d))$. While this is sufficient in most cases, it might not provide the desired semantics if x contains more events than d. In those cases, the expression does not delay the last known value at the time when the delay was set, but it produces the last known value at the time when the delay is over. This problem can be overcome by removing those additional events on x first:

Definition 3.47 (Semantics of the Operator delayedLast)**.** The derived TeSSLa operator $\mathsf{delayedLast}\colon \mathcal{S}_{\mathbb{D}} \times \mathcal{S}_{\mathbb{T}} \to \mathcal{S}_{\mathbb{D}}$ delays the value of the first stream for the amount of time specified by the second stream:

$$\mathsf{delayedLast}(x, d) \equiv \mathbf{last}(\mathsf{on}(d, x), \mathbf{delay}(d))$$ ⌟

After the delay passes, the operator emits an event carrying the delayed value in the resulting stream. Like with the **delay** operator, delays are not accumulated. A new delay resets a currently active one. A delay of 0 only resets the currently active delay without setting a new one.

3.3.7. Implicit Type Conversions and Type Checking

The following list sums up all the implicit type conversions introduced for TeSSLa specification:

- A partial function $\mathbb{D}_1 \times \mathbb{D}_2 \times \ldots \times \mathbb{D}_n \hookrightarrow \mathbb{D}$ (or a function $\mathbb{D}_1 \times \mathbb{D}_2 \times \ldots \times \mathbb{D}_n \to \mathbb{D}$) on the data domains can be extended to a conservative (or ideal) $\perp$-function $\mathbb{D}_1 \times \mathbb{D}_2 \times \ldots \times \mathbb{D}_n \rightarrowtail \mathbb{D}$ on the data domains.
- A value $d \in \mathbb{D}$ of the data domain can be converted to the stream $\mathsf{const}(d) \in \mathcal{S}_{\mathbb{D}}$ representing a constant signal.
- A partial function $f\colon \mathbb{D}_1 \times \mathbb{D}_2 \times \ldots \times \mathbb{D}_n \hookrightarrow \mathbb{D}$ on the data domains can be converted to the function $\mathsf{slift}(f)\colon \mathcal{S}_{\mathbb{D}_1} \times \mathcal{S}_{\mathbb{D}_2} \times \ldots \times \mathcal{S}_{\mathbb{D}_n} \to \mathcal{S}_{\mathbb{D}}$ on streams.
- A stream $s \in \mathcal{S}_{\mathbb{D}}$ can be converted to the unit stream $\mathsf{const}(\Box, s) \in \mathcal{S}_{\mathbb{U}}$ by ignoring the event's values.

The type-checking problem is straightforward because every function and stream is typed, and hence new functions derived by composing existing functions are automatically typed. Which implicit type conversion should be applied can be easily determined if everything is fully typed. There is always only one conversion applicable, and if it is applicable, it must be applied. In some cases, the conversions can

be chained, but since there is always only one conversion applicable, the application of such a conversion chain is still straightforward.

In practical TeSSLa implementations, one usually wants to add some additional type inference to overcome the requirement to add type annotations to every stream explicitly. Explicit type annotations are then only needed for recursive equations if the type of the stream cannot be inferred from the used operators.

3.4. Design Choices

TeSSLa is designed as a specification language based on stream transformation and well-suited for online monitoring in hardware settings with limited memory. The following design goals for TeSSLa are introduced in [CHS+18, Section 4.2] and elaborated below:

- *Specification.* A specification has a different and usually more abstract view than the actual implementation. While an implementation contains all the details and edge cases, a specification can focus on particular aspects. Implementations are usually organised in units containing related aspects. Specifications can express global invariants across the entire implementation. As a stream processing language, TeSSLa provides a different perspective that can help avoid making the same mistakes in the specification and the implementation. However, TeSSLa does not require previous knowledge in mathematical logic: TeSSLa specifications can be seen more as a monitor description than a description of valid runs of a system.
- *Data.* TeSSLa supports arbitrary operations on the data domain. It thus allows quantitative and statistical analysis in addition to correctness checks based on timestamps or event ordering constraints. TeSSLa has no restrictions on the data domain because it is defined independently from the available operations on the data domain. TeSSLa only provides mechanisms that lift these operations to streams.
- *Time.* TeSSLa has built-in support for timestamps from a continuous time domain. The full power of timed event streams with arbitrary precision comes with all benefits and drawbacks. TeSSLa has no specific operations to check timing constraints. Instead, timestamps can be used as data values and processed with the same operations. The time domain must be a totally-ordered semi-ring because we need to be able to check orders of events, and for the generation of additional events with the **delay** operator, we need calculations on timestamps.

- *Formal Semantics.* A TeSSLa specification is an equation system over streams. The streams are considered immutable because the specification contains every derived stream exactly ones on the left-hand side of an equation. The right-hand side of the equation defines the stream by applying operators to other streams. TeSSLa has a small set of basic operators with an explicit semantics. The formal semantics is given as a fixed point over the potentially recursive equations, which allows expressing complex computations. For better usability, new operators can be defined as compositions of the basic operators. Practical implementations facilitate this in the form of a macro system.
- *Explicit Memory Usage.* TeSSLa follows the idea of explicit memory usage. As discussed in the introduction, the targeted scenario is online monitoring without access to the entire input stream. The input events arrive in order and are processed sequentially. The TeSSLa operators are designed such that the number of data values that TeSSLa implementations must store is independent of the number of processed events, i. e. the implementations do not need to keep track of the entire trace processed so far. They only need to store the data values of certain events. However, if specifications require unbound memory, then complex data structures like queues can be used in TeSSLa because the TeSSLa operators are agnostic towards the data domain. In that case, the implementation still only keeps track of a fixed number of data values, but a value from the data domain can be of arbitrary size. The explicit memory usage of TeSSLa makes it easy to check whether a given specification can be compiled into software with fixed memory usage or hardware configurations.

This section discusses properties of the TeSSLa operators and introduces additional unusual operators, which can be derived from the basic operators. On the one hand, they demonstrate the expressiveness of TeSSLa, and on the other hand, they are used to explain why the actual basic TeSSLa operators are not defined this way.

Although this section only refers to the TeSSLa semantics defined in the last section, all the results naturally carry over to the monitoring semantics introduced in the next section.

3.4.1. Lifting Nested Functions

We have established the convention that functions on the data domain are automatically lifted to streams using the signal lift operator slift. While this makes it very convenient to write TeSSLa specifications, it might seem unclear how nested applications of functions defined on the data domain are applied to the stream domain. We can show that lifting a composed function using a single **lift** has the same effect as composing the individually lifted functions. The same holds for slift, too.

Lemma 3.48 (Associativity of **lift**)**.** *Let* $f\colon A \times B \rightarrowtail X$ *and* $g\colon X \times C \rightarrowtail Y$ *be two well-formed* $\bot$*-functions on data domains. Then* $h\colon A \times B \times C \rightarrowtail Y$ *with* $h(a,b) = g(f(a,b),c)$ *is the composition of the two functions. We then have for corresponding streams* $a \in \mathcal{S}_A, b \in \mathcal{S}_B, c \in \mathcal{S}_C$ *the following equality:*

$$\mathbf{lift}(h)(a,b,c) \equiv \mathbf{lift}(g)(\mathbf{lift}(f)(a,b),c)$$

Proof. From the definition we get that the stream $\ell_f = \mathbf{lift}(f)(a,b)$ is given by

$$\ell_f(t) = f(a(t), b(t)).$$

Consequently $\ell = \mathbf{lift}(g)(\ell_f, c)$ is given by

$$\ell(t) = g(\ell_f(t), c(t)) = g(f(a(t), b(t)), c(t)),$$

which is exactly how $\mathbf{lift}(h)(a,b,c)$ is given. □

As a direct consequence of the definition of the sync operator, we get the following statement:

Lemma 3.49 (Oversampling of Signals)**.** *Let* $f\colon A \times B \hookrightarrow Z$ *be a partial function and* $a \in \mathcal{S}_A$*,* $b \in \mathcal{S}_B$ *and* $c \in \mathcal{S}_C$ *streams. We then have*

$$\begin{aligned}\mathsf{sync}(f(a,b),c) &\equiv f(\mathsf{sync}(a,c), \mathsf{sync}(b,c)) \\ &\equiv f(\mathsf{sync}(a,c), b) \equiv f(a, \mathsf{sync}(b,c))\end{aligned}$$

This lemma is called oversampling of signals because it does not matter if we synchronise a and b individually before applying the function f (implicitly using slift) or if we synchronise the result of applying the function f with c.

By combining the Lemmas 3.48 and 3.49 from above, we can show the same associativity result for slift:

Lemma 3.50 (Associativity of slift)**.** *Let* $f\colon A \times B \hookrightarrow X$ *and* $g\colon X \times C \hookrightarrow Y$ *be two partial functions on data domains. Then* $h\colon A \times B \times C \hookrightarrow Y$ *with* $h(a,b,c) = g(f(a,b),c)$ *for all* a, b *on which* $f(a,b)$ *is defined, is the composition of the two functions. We then have for corresponding streams* $a \in \mathcal{S}_A, b \in \mathcal{S}_B, c \in \mathcal{S}_C$ *the following equality:*

$$\mathsf{slift}(h)(a,b,c) \equiv \mathsf{slift}(g)(\mathsf{slift}(f)(a,b),c)$$

Proof. Let $\ell_f = \mathsf{slift}(f)(a,b)$ and $\ell_g = \mathsf{slift}(g)(\ell_f, c)$. With the definition of slift we get

$$\ell_g = \mathbf{lift}(g)(\mathsf{sync}(\ell_f, c), \mathsf{sync}(c, \ell_f)).$$

By applying Lemma 3.49 we get

$$\ell_g = \mathbf{lift}(g)(\mathsf{slift}(f)(\mathsf{sync}(a,c), \mathsf{sync}(b,c)), \mathsf{sync}(c, \ell_f)).$$

Because f and g are implicitly extended into conservative $\bot$-functions we know: If any input is $\bot$ the output is $\bot$. Thus, we can replace $\mathsf{sync}(\mathsf{sync}(a,c), \mathsf{sync}(b,c))$ with $\mathsf{sync}(a,b,c)$ which might contain additional events in the beginning which are then removed by $\mathbf{lift}(f)$ and $\mathbf{lift}(g)$:

$$\ell_g = \mathbf{lift}(g)(\mathbf{lift}(f)(\mathsf{sync}(a,b,c), \mathsf{sync}(b,a,c)), \mathsf{sync}(c,a,b))$$

Now we can apply Lemma 3.48 and get

$$\ell_g = \mathbf{lift}(h)(\mathsf{sync}(a,b,c), \mathsf{sync}(b,a,c), \mathsf{sync}(c,a,b)),$$

which is the definition of $\mathsf{slift}(h)(a,b,c)$. □

The proof relies on slift taking a partial function which is extended to a conservative $\bot$-function when passed to **lift**. If slift would take an arbitrary well-formed $\bot$-function instead, which could individually handle $\bot$ on its inputs, this could break the associativity of slift.

To summarise the above results, consider the expression $x+y+z$ for three streams $x, y, z \in \mathcal{S}_\mathbb{R}$. With the implicit conversion this expression will be expanded to $\mathsf{slift}(+)(x, \mathsf{slift}(+)(y,z))$ under the assumption that the conversion is applied from left to right. Since this is the same as $\mathsf{slift}(+)(\mathsf{slift}(+)(x,y), z)$ or even $\mathsf{slift}(f)(x,y,z)$ for a function

$$f\colon \mathbb{R}^3 \to \mathbb{R} \text{ with } f(a,b,c) = a+b+c$$

it is not needed to clarify the order of the application of the implicit conversion further. This equivalence allows many powerful performance optimisations for different backends. Depending on the implementation of a TeSSLa backend, it might be more efficient to lift complex expressions on the data domain into a single function or multiple functions on streams.

As a final example, consider $x+3$ for a stream $x \in \mathcal{S}_\mathbb{N}$ and a constant 3 on the data domain. This expression is interpreted as $\mathsf{slift}(+)(x, \mathsf{const}(3))$, which is the same as $\mathbf{lift}(f)(x)$ for a function $f\colon \mathbb{N} \to \mathbb{N}$ with $f(i) = i+3$.

3.4.2. Basic Operators

We defined the TeSSLa semantics by defining a minimal set of operators and then derived further operators as compositions of the basic operators. As with many languages, there are different possible sets of basic operators. The set of operators shown in the following theorem would also be a possible set of basic operators:

Theorem 3.51 (Signal Lift and Default)**.** *The operators* default*,* slift*,* **last** *and* **delay** *together with the accessor function* **time** *and the basic stream* nil *are sufficient to gain the expressiveness of TeSSLa.*

Proof. We proof the statement by deriving the missing operators **lift** and **unit** from the given ones:

$$\mathbf{unit} = \mathsf{default}(\mathsf{nil}, \square)$$

The helper function $\overline{\mathsf{sync}} \colon \mathcal{S}_{\mathbb{A}} \times \mathcal{S}_{\mathbb{B}} \to \mathcal{S}_{\mathbb{A}_\perp}$ synchronises two streams by adding additional events with the value $\perp$ whenever b has an event, but a does not. This transforms the stream from the data domain $\mathbb{A}$ to a stream over the data domain $\mathbb{A}_\perp$ which allows us to pass a $\perp$-function to slift:

$$\overline{\mathsf{sync}}(a, b) = \mathsf{default}(\mathbf{time}(a), 0) \geq \mathsf{default}(\mathbf{time}(b), 0) \mathrel{?} \mathsf{default}(a, \perp) : \perp$$

Now we get

$$\mathbf{lift}(f)(a, b) = \mathsf{slift}(f)(\overline{\mathsf{sync}}(a, b), \overline{\mathsf{sync}}(b, a)).$$

□

This set of operators might seem to be a more natural choice because one immediately gets the operators slift and default, which are helpful when writing specifications. However, these operators are more complex: The slift operator is a combination of **lift** and **last** because it has to remember the last known values, and it applies a function. The definition of **lift** is much simpler. Further, defining slift using **lift** and **last** as done in Definition 3.39 in Section 3.3.3 is more straight forward than defining **lift** using slift as done in the proof above.

In this thesis the basic operators **unit**, **time**, **lift**, **last** and **delay** are defined to gain a formal semantics for TeSSLa. For the different implementations, however, different sets of operators are considered. For example, we will discuss in Chapter 5 that the slift operator is a very natural operator for the compilation towards EPUs.

3.4.3. Events and Signals

As already motivated in Section 3.1.2 lifting functions with **lift** is more useful when thinking in terms of events and lifting functions with slift is more useful when thinking in terms of piece-wise constant signals introduced as the signal view. In order to write readable and intuitive TeSSLa specifications, one needs both operators. In terms of expressiveness, they can be converted into each other, as shown above.

As an example of the important difference between signals and events, we introduce a slight variant of the filter operator, which uses the synchronisation from the signal view on both input streams:

Definition 3.52 (Semantics of the Operator sfilter)**.** The derived TeSSLa operator sfilter: $\mathcal{S}_{\mathbb{D}} \times \mathcal{S}_{\mathbb{B}} \to \mathcal{S}_{\mathbb{D}}$ filters a given event stream based on the current value of a Boolean stream:

$$\mathsf{sfilter}(x, c) \equiv \mathsf{slift}(f)(x, c)$$

with the partial function $f\colon \mathbb{D} \times \mathbb{B} \hookrightarrow \mathbb{D}$ given by

$$f(d, a) = \begin{cases} d & \text{if } a = \mathit{true}, \\ \bot & \text{otherwise.} \end{cases}$$

The filter operator was defined in Definition 3.40 in Section 3.3.3 as

$$\mathsf{filter}(x, z) \equiv \mathbf{lift}(f)(x, \mathsf{sync}(z, x))$$

using the same function f as in the above definition. The main difference is that in case of sfilter both streams x and z are synchronised with each other and not only the condition stream z. This additional synchronisation causes additional events, which can be interpreted as oversampling if the filtered stream can be seen as a piece-wise constant signal. If not, these additional events might be unwanted. As an example consider traces in Figure 3.3: The stream $\mathsf{sfilter}(x, z)$ contains two additional events in comparison with $\mathsf{filter}(x, z)$.

- The event with value 2 is repeated on $\mathsf{sfilter}(x, z)$ because of the repeated event with value true on z.
- The stream $\mathsf{sfilter}(x, z)$ contains an additional event with value 4 in the end because the last known value on x is used.

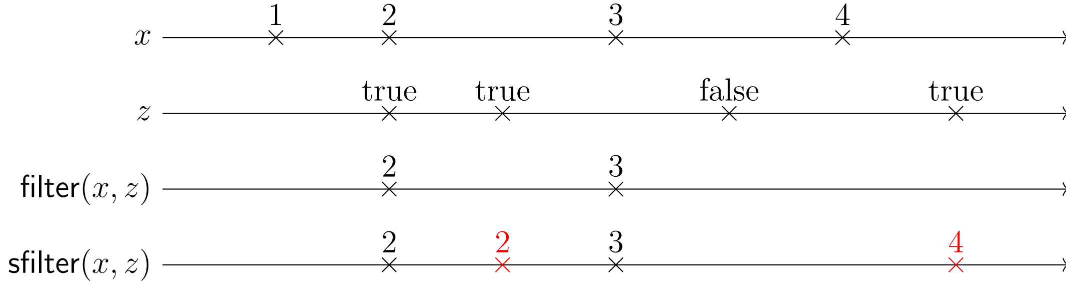


Figure 3.3.: Comparison of the filter operator which considers x as stream of individual events and z as signal and the sfilter operator which treats both inputs using the signal view. The additional events generated by sfilter are highlighted in red.

It is possible to derive the original filter semantics from slift, too:

$$\mathsf{filter}(x, z) \equiv \mathsf{slift}(f)(x, \mathsf{on}(x, z))$$

with the function f being the same as before. The additional events are removed by sampling the condition stream with the events on x.

These examples show the importance of carefully specifying which events are used in a computation. Properly synchronising events is a central aspect of writing TeSSLa specifications.

In this thesis, the distinction between streams of individual events and streams representing piece-wise constant signals is not made explicitly on the type level. We have only introduced one type for streams: The set $\mathcal{S}_{\mathbb{D}}$ of all streams over the data domain $\mathbb{D}$. The typing does not distinguish if a stream represents events or piece-wise constant signals.

For the theoretical results and the implementation of the different backends, it is easier not to make this distinction on the level of the basic TeSSLa operators. Derived operators, however, might distinguish between different stream types and even introduce additional stream types together with libraries of derived operators suitable to manipulate those streams, e. g. streams representing piece-wise linear signals or even streams representing signals constructed from sine waves. TeSSLa directly supports such operators as long as those streams are represented as a discrete sequence of timed events. The discrete events always indicate the points in time where the signal switches from one representation to the next, i. e. in the case of the piece-wise constant signal, the points where the signal changes its value. TeSSLa frontends could provide static type checking for different stream types to help users write meaningful specifications.

3.4.4. Generating Zeno-Streams

As already discussed in Section 3.2.1 a stream can represent Zeno behaviour. The main problem with Zeno behaviour is that it prevents any monitor which processes events in their order in the stream from advancing beyond the limit towards the stream's timestamps are converging. Infinitely many events in a finite time window cannot all be processed in finite time. There are two ways how the outputs of a specification can become Zeno:

1. If an input stream is Zeno and the specification derives a new stream with identical timestamps from that input stream, the derived stream becomes Zeno, too. This scenario is problematic for making progress in online monitoring but is not considered harmful because the problem originates from the input. Even transmitting the infinitely many input events to the monitor will take infinitely long, so in this scenario, the event source should be considered flawed. Especially since any measurements of events from real-world sources should not be Zeno because "real physical systems are not Zeno" [ZJLS00].

2. TeSSLa can create Zeno streams from non-Zeno streams. TeSSLa can create new timestamps using the **delay** operator and nothing prevents one from generating for example the stream shown in Example 3.18 (Zeno Stream) from Section 3.2.1 where the n-th timestamp is given by $t_n = 2 - 1/(1+n)$. That is considered harmful because the monitor stops processing events from the input beyond this limit. The monitor will be busy forever generating these events and can be considered essentially stuck in an endless loop.

   Whether a TeSSLa specification generates a Zeno stream is a semantic property of the specification. Hence TeSSLa allows the user to write such harmful specifications similar to how every Turing-complete programming language can express an endless loop.

   Example 3.19 (Non-Zeno Stream) from Section 3.2.1 demonstrates that requiring specifications to have a smallest timestamp delta is not the same as requiring specifications to generate only non-Zeno streams: The stream in the example does not have Zeno behaviour, although there is no smallest timestamp delta.

3.4.5. Memory Usage

TeSSLa has explicit memory usage, i. e. without the explicit usage of an unbound data structure, the TeSSLa operators support implementations with a fixed amount of memory. TeSSLa still supports specifications that require unbound memory. If

required, the user can explicitly use unbound data structures such as lists, sets or queues. This section illustrates this design choice: We will demonstrate how powerful TeSSLa operators can be defined as compositions of the basic operators that utilise unbound data structures. These operators serve two main purposes in this thesis:

1. They demonstrate the expressiveness of the TeSSLa specification language and
2. they illustrate why the basic TeSSLa operators are defined as they are: If they would natively support the behaviour of the operators introduced below, then every implementation of TeSSLa would need unbound memory.

With the principle of explicit memory usage, the usage of unbound data structures is made explicitly by the user and not hidden inside the semantics of the basic operators.

As a first example, we consider the prevn operator. It is a variation of prev operator that allows unbound dynamic access to previous events, i. e. at runtime based on input data, the prevn operator provides access to any previous event.

In the case of the prev operator, online monitoring implementations need to store one event's value for every prev operator in the specification. For the composed prev^n, this would be n values. In this case, the n is statically known at compile time.

Instead, if the n is determined dynamically, i. e. via events on another stream, there is no upper bound for n. Then an online monitor implementation would need to store all events seen on the stream so far. This unbound storage requires a dynamic data structure, e. g. a list:

Definition 3.53 (Semantics of the Operator prevn)**.** The derived TeSSLa operator $\mathsf{prevn}\colon \mathcal{S}_{\mathbb{D}} \times \mathcal{S}_{\mathbb{N}>0} \to \mathcal{S}_{\mathbb{D}}$ allows access to the n-th previous event where n is dynamically given as stream:

$$\mathsf{prevn}(x, n) = \mathsf{slift}(f)(v, \mathsf{on}(v, n))$$

with the function $f\colon \mathbb{D}^* \times \mathbb{N} \to \mathbb{D}$ given by

$$f(l, i) = \begin{cases} l[i] & \text{if } i < |l|, \\ \bot & \text{otherwise,} \end{cases}$$

and the stream $v \in \mathcal{S}_{\mathbb{D}^*}$ given by the following equation

$$v = \mathsf{default}(a :: \mathbf{last}(v, a), \langle\rangle).$$ ⌟

The above definition uses a stream over the data domain $\mathbb{D}^*$, i. e. a stream of finite lists over $\mathbb{D}$. So every event of the stream can contain a finite list as its value.

Over $\mathbb{D}^*$ the operator $::$ prepends an element to the list. The notation $l[i]$ for a list $l \in \mathbb{D}^*$ and $i \in \mathbb{N}$ accesses the $(i+1)$-th element of the list. All complex data structures in TeSSLa are always considered immutable. The immutability makes it possible to use data domains consisting of complex data structures in the same way as data domains consisting of primitive data types.

The prevn operator is not well-suited for online monitoring: It converts the entire so far processed stream into a list in order to simulate random access to all events seen so far. While this is possible and can be expressed in TeSSLa, it is not efficient because it breaks with the fundamental idea of online monitoring for streams: We only want to store a finite amount of data independently of the stream length. The memory consumption of prevn, however, increases with the length of the stream.

A similar situation arises if we adjust the **delay** operator so that it no longer resets the currently active delay with every new delay but instead keeps track of all open delays in a set:

Definition 3.54 (Semantics of the Operator mdelay)**.** The derived TeSSLa operator mdelay: $\mathcal{S}_\mathbb{T} \to \mathcal{S}_\mathbb{U}$ produces a unit event at timestamp $t + a$ for every event at timestamp t with value a. The expression $\mathsf{mdelay}(x) = z$ is defined in terms of the internal streams $\ell, o \in \mathcal{S}_{2^\mathbb{T}}$ of sets of timestamps, $m \in \mathcal{S}_\mathbb{T}$ of timestamps and $z \in \mathcal{S}_\mathbb{U}$ of unit events given below:

$$\begin{aligned}
\ell &= \mathsf{default}(\mathbf{last}(o, \mathsf{merge}(x, z)), \emptyset) \\
o &= \mathbf{lift}(f_o)(\ell, \mathbf{time}(\ell), x) \\
m &= \mathbf{lift}(f_m)(o) \\
z &= \mathbf{delay}(m - \mathbf{time}(m))
\end{aligned}$$

with the lifted functions $f_o \colon 2^\mathbb{T} \times \mathbb{T} \times \mathbb{T} \to 2^\mathbb{T}$ given by

$$f_o(S, t, a) = \begin{cases} \{s \in S \mid s > t\} & \text{if } a = \bot, \\ \{s \in S \mid s > t\} \cup t + a & \text{otherwise,} \end{cases}$$

and $f_m \colon 2^\mathbb{T} \to \mathbb{T}$ given by

$$f_m(s) = \begin{cases} \min(s) & \text{if } |s| > 0, \\ \bot & \text{otherwise.} \end{cases}$$

⌟

We represent maps over a key domain $\mathbb{K}$ and a value domain $\mathbb{V}$ as partial functions $\mathbb{K} \hookrightarrow \mathbb{V}$. For such a partial function $f\colon \mathbb{K} \hookrightarrow \mathbb{V}$ we define $|f|$ to be the number of different keys $k \in \mathbb{K}$ for which $f(k)$ is defined, i. e. the size of the map. In the above definition, we use timestamps as keys of the map and further define $\min(f)$ to be the minimal key $k \in \mathbb{K}$ such that $f(k)$ is defined, i. e. the minimum of the key-set of the map.

The above definition uses streams over a data domain of unbound sets. Again these sets might grow indefinitely during the execution of an online monitor for this specification. However, in this case, it depends on the input stream how much memory will be consumed because the set only stores the open delays.

With some slight modifications a non-resetting variant of delayedLast can be given as well:

Definition 3.55 (Semantics of the Operator mdelayedLast)**.** The derived TeSSLa operator $\mathsf{mdelayedLast}\colon \mathcal{S}_{\mathbb{D}} \times \mathcal{S}_{\mathbb{T}} \to \mathcal{S}_{\mathbb{D}}$ takes a value stream and a delay stream. It produces an event at timestamp $t+a$ with the last known value on the value stream at timestamp t for every event on the delay stream at timestamp t with value a. The expression $\mathsf{mdelayedLast}(v, x) = w$ is defined in terms of the internal streams $\ell, o, y \in \mathcal{S}_{\mathbb{T} \hookrightarrow \mathbb{D}}$ of maps of timestamps to values, $m \in \mathcal{S}_{\mathbb{T}}$ of timestamps, $z \in \mathcal{S}_{\mathbb{U}}$ of unit events and $w \in \mathcal{S}_{\mathbb{D}}$ given below:

$$\ell = \mathsf{default}(\mathbf{last}(o, \mathsf{merge}(x, z)), \emptyset)$$
$$o = \mathbf{lift}(f_o)(\ell, \mathbf{time}(\ell), x, \mathsf{on}(x, v))$$

with the lifted function $f_o\colon (\mathbb{T} \hookrightarrow \mathbb{D}) \times \mathbb{T} \times \mathbb{T} \times \mathbb{D} \to (\mathbb{T} \hookrightarrow \mathbb{D})$ given by

$$f_o(S, t, a, d) = \begin{cases} \{(s, e) \in S \mid s > t\} & \text{if } a = \bot, \\ \{(s, e) \in S \mid s > t\} \cup (t + a, d) & \text{otherwise.} \end{cases}$$

$$m = \mathbf{lift}(f_m)(o)$$

with the lifted function $f_m\colon 2^{\mathbb{T}} \to \mathbb{T}$ given by

$$f_m(s) = \begin{cases} \min(s) & \text{if } |s| > 0, \\ \bot & \text{otherwise.} \end{cases}$$

$$z = \mathbf{delay}(m - \mathbf{time}(m))$$
$$y = \mathsf{on}(z, \mathsf{prev}(o)))$$
$$w = \mathbf{lift}(f_w)(y, \mathbf{time}(z))$$

with the lifted function $f_w\colon (\mathbb{T} \hookrightarrow \mathbb{D}) \times \mathbb{T} \to \mathbb{D}$ given by

$$f_w(y, t) = y(t).$$ ⌟

This operator can be used to derive the following convenient operator:

Definition 3.56 (Semantics of the Operator shift). The derived TeSSLa operator $\mathsf{shift}\colon \mathcal{S}_{\mathbb{D}} \times \mathcal{S}_{\mathbb{T}^+} \to \mathcal{S}_{\mathbb{D}}$ takes a stream of events and a stream of timestamps and delays every event by the amount of time given by the stream of timestamps:

$$\mathsf{shift}(x, s) = \mathsf{mdelayedLast}(x, \mathsf{on}(x, s))$$ ⌟

The shift operator can be used with constants being automatically converted to signals, too. For example $\mathsf{shift}(x, 2)$ delays every event on x by 2 time units. As with all delaying operators, negative delays are not allowed because that would no longer be future independent.

Again without any limit on how many events can occur in the range of 2 time units, there is no upper limit on how big the set storing all these events might become at runtime.

The main principle behind the definition of the basic TeSSLa operator is not to forbid the definition of these operators. We have shown in this section that they can be derived from the basic TeSSLa operators. However, unbound memory usage is made explicit because we always needed to introduce streams over unbound data domains.

3.5. Monitoring

In the setting of online monitoring, the monitor receives individual events step by step. The monitor does not have access to the entire trace, not even to previous events. The TeSSLa semantics on streams introduced in Section 3.2 only considers fully known streams that cannot be extended any further. As the next step towards implementations for online monitoring, this section introduces monitoring streams and the monitoring semantics over monitoring streams.

Monitoring streams encode additional information on how far the stream is known and possible continuations of the stream. We can then define a prefix relation on monitoring streams that describes how to extend a stream.

This thesis discusses different approaches for encoding streams: As illustrated in Figure 1.1 in Section 1.4, the synchronised streams presented in Chapter 4 and the abstract monitoring streams in Chapter 6 will be introduced as special cases of the more general approach of the monitoring streams introduced next.

3.5.1. Monitoring Streams

We define monitoring streams as sets of streams. This set represents what we already know about a stream and where multiple continuations are possible.

Definition 3.57 (Monitoring Stream)**.** A *monitoring stream* p over a time domain $\mathbb{T}$ and a data domain $\mathbb{D}$ is a set of streams over $\mathbb{T}$ and $\mathbb{D}$, i. e. $p \in \mathcal{P}_{\mathbb{D}} = 2^{\mathcal{S}_{\mathbb{D}}} \setminus \emptyset$. ⌟

$\mathcal{P}_{\mathbb{D}}$ is the power set of all possible streams without the empty set. This set is further restricted as follows:

Definition 3.58 (Monitoring Stream of Independent Events)**.** A monitoring stream $p \in \mathcal{P}_{\mathbb{D}}$ over a time-domain $\mathbb{T}$ *consists of independent events* iff for all streams $r, s \in p$ and for all timestamps $t \in \mathbb{T}$ there exists a stream $q \in p$ defined by

$$q(t') = \begin{cases} s(t') & \text{if } t' = t \\ r(t') & \text{otherwise.} \end{cases}$$

⌟

This property requires the events at a certain timestamp to be independent from their specific stream. Note that r, s and q do not necessarily have to be three different streams. In the following we implicitly assume that all monitoring streams consists of independent events.

Intuitively we consider the monitoring stream to be known as long as all the streams of the set are equal, i. e. they contain the same events with the same values at identical timestamps. From the point where any of the streams differs from the other streams in the set, we consider the different streams to represent all possible continuations.

This thesis uses sets of streams to encode monitoring streams for several reasons:

- Sets of streams are a simple extension of streams. This extension allows the further use of the existing TeSSLa operators. We will raise the operators defined on streams to monitoring streams by applying them to all streams in the set individually.
- Sets of streams can represent multiple regions of uncertainty. A region between two timestamps is considered to be of full knowledge if all streams in the set contain the same events in the region. A monitoring stream can contain multiple regions where the different streams in the set differ. Thus, monitoring streams can be seen as a unified theory that can integrate work on TeSSLa for timed event streams with partial information [LSS+19]. The rest of this section discusses these abilities of monitoring streams in general. However,

actual implementations which can handle inputs with partial information are beyond the scope of this thesis.

- The different implementations of TeSSLa shown in Figure 1.1 in Section 1.4 rely on different semantics, which are abstractions of the monitoring semantics. Thus, the semantics of the implementations can be shown to adhere to the common monitoring semantics, although not all its details can be realised. The corresponding chapters of the implementations introduce Galois connections (α, γ) between monitoring streams and the respective stream representation of the implementation. Representing monitoring streams as sets of streams makes the definition of the abstraction function α and the concretisation function γ straight-forward.

Following the notation $\mathcal{S}_{\mathbb{D}}^n$ (see Section 3.2.2) we use $\mathcal{P}_{\mathbb{D}}^n$ to indicate the Cartesian product $\mathcal{P}_{\mathbb{D}_1} \times \mathcal{P}_{\mathbb{D}_2} \times \ldots \times \mathcal{P}_{\mathbb{D}_n}$ of streams over the individual data domains $\mathbb{D}_1, \mathbb{D}_2, \ldots, \mathbb{D}_n$.

Similarly to Definition 3.15 on streams in Section 3.2.1, we define the following on monitoring streams:

Definition 3.59 (Timestamps of a Monitoring Stream)**.** For a monitoring stream $p \in \mathcal{P}_{\mathbb{D}}$ we define $T(p) \subseteq \mathbb{T}$ to be the set of timestamps carrying events in p:

$$T(p) := \bigcup_{s \in p} T(s).$$

We further define $T(\boldsymbol{p})$ for all timestamps occurring in $\boldsymbol{p} \in \mathcal{P}_{\mathbb{D}}^n$:

$$T(\boldsymbol{p}) := \bigcup_{1 \leq i \leq k} T(p_i).$$

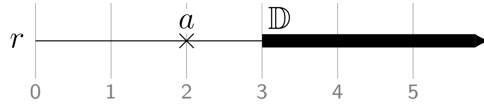
⌟

Example 3.60 (Fully Known and Fully Unknown Monitoring Stream)**.** The fully known stream $s \in \mathcal{S}_{\mathbb{D}}$ is represented as the singleton set $\{s\} \in \mathcal{P}_{\mathbb{D}}$. The fully unknown stream is represented as $\mathcal{S}_{\mathbb{D}} \in \mathcal{P}_{\mathbb{D}}$. ⌟

Example 3.61 (Monitoring Stream)**.** Let $r \in \mathcal{P}_{\mathbb{D}}$ be a monitoring stream given as follows:

$$r = \{s \in \mathcal{S}_{\mathbb{D}} \mid s(2) = a \wedge \forall\, t \neq 2 \wedge t < 3\colon s(t) = \bot\}.$$

We know about an event at timestamp 2 with value $a \in \mathbb{D}$, and we know that there are no other events until timestamp 3 (exclusively), but we know nothing after timestamp 3 (inclusively). In the stream visualisations we represent this monitoring stream r as follows:



The stream is fully known up to (but not including) timestamp 3, indicated by the thin bar used before. All streams in r have an event with value a at timestamp 2, drawn as regular events as before. Starting with timestamp 3, the monitoring stream r contains all possible continuations, indicated by the thick bar. The data domain $\mathbb{D}$ above the thick bar shows that the events in the region can have any value from $\mathbb{D}$. ⌟

A monitoring stream $p \in \mathcal{P}_{\mathbb{D}}$ can represent various levels of uncertainty for a particular timestamp t:

- In the definitive case, all streams in the set show the same behaviour at timestamp t: Either all streams have an event with the same value at t, or no stream has an event at t. This definitive case is visualised either as a regular event or as a thin timeline without any event, respectively.
- In the next weaker case, the event pattern is definitive, but the values are uncertain: Every stream in the set contains an event at timestamp t, but the values differ. This case is visualised as a regular event whose value is given as a set of possible values.
- In addition to streams with different values for events at timestamp t we can also have streams in the set without an event at timestamp t: Some streams contain an event at t with a limited set of values, and other streams contain no event at that timestamp. This case is visualised as a thick bar instead of an event. The possible values of events at t are written above the thick bar.
- We have full uncertainty at timestamp t: At least one stream has no event at t, and for every possible value in the data domain $\mathbb{D}$ there exists a stream with an event at t carrying that value. This case is visualised as a thick bar instead of an event. Above the thick bar, the data domain of the stream is written as a set of possible values for events at timestamp t.

For a stream $a \in \mathcal{S}_{\mathbb{D}}$ we abuse notation if the types are obvious from the context and write a to represent $\{a\} \in \mathcal{P}_{\mathbb{D}}$ and for a tuple of streams $\boldsymbol{a} = (a_1, a_2, \ldots, a_n) \in \mathcal{S}_{\mathbb{D}}^n$ we write $\boldsymbol{a}$ to represent $(\{a_1\}, \{a_2\}, \ldots, \{a_n\}) \in \mathcal{P}_{\mathbb{D}}^n$.

We extend Definition 3.20 from Section 3.2.1 to monitoring streams as follows:

Definition 3.62 (Limit of a Monitoring Stream)**.** For a monitoring stream $p \in \mathcal{P}_{\mathbb{D}}$ we define its *limit* to be

$$\mathrm{limT}(p) = \min_{s \in p} \mathrm{limT}(s).$$ ⌟

We can now define a prefix relation on monitoring streams. The fewer streams are in a monitoring stream, the more we know about the stream:

Definition 3.63 (Refinement Relation)**.** We define the *refinement relation* $\sqsubseteq \subseteq \mathcal{P}_{\mathbb{D}} \times \mathcal{P}_{\mathbb{D}}$ as inverse subset relation. For arbitrary $a, b \in \mathcal{P}_{\mathbb{D}}$ we define

$$a \sqsubseteq b \Leftrightarrow b \subseteq a$$

and say that b is a *refinement* of a. ⌟

Every stream in $\mathcal{P}_{\mathbb{D}}$ is a refinement of the fully unknown stream and hence the fully unknown stream the least element of $(\mathcal{P}_{\mathbb{D}}, \sqsubseteq)$:

$$\forall\, p \in \mathcal{P}_{\mathbb{D}} \colon p \sqsubseteq \mathcal{S}_{\mathbb{D}}.$$

There is no common greatest element on the other end because the fully known streams are all incomparable regarding the refinement relation.

From the above definitions we can derive the following statement, because every directed subset of $(\mathcal{P}_{\mathbb{D}}, \sqsubseteq)$ has a supremum:

Lemma 3.64 (Refinement Relation is a Dcpo)**.** $(\mathcal{P}_{\mathbb{D}}, \sqsubseteq)$ *is a dcpo.*

Intuitively, the directed subsets can be seen as sets of prefixes of a common monitoring stream, and for every such set of prefixes, there exists a longest prefix.

We extend the refinement relation to tuples of streams by applying it to every pair of streams in the tuples individually. Let $\boldsymbol{s}, \boldsymbol{u} \in \mathcal{P}_{\mathbb{D}}^{k}$ be two tuples of monitoring streams. We then have

$$\boldsymbol{s} \sqsubseteq \boldsymbol{u} \Leftrightarrow \forall\, i \in \{1, 2, \ldots, k\} \colon s_i \sqsubseteq u_i.$$

3.5.2. Monitoring Semantics

In order to define the monitoring semantics on monitoring streams, we first raise the TeSSLa operators defined on streams to monitoring streams:

Definition 3.65 (TeSSLa Operators on Monitoring Streams)**.** The TeSSLa operators defined on $\mathcal{S}_{\mathbb{D}}$ are raised to $\mathcal{P}_{\mathbb{D}}$ by applying them to all streams in the set individually. Let $o : \mathcal{S}_{\mathbb{D}_1} \times \mathcal{S}_{\mathbb{D}_2} \times \ldots \times \mathcal{S}_{\mathbb{D}_k} \to \mathcal{S}_{\mathbb{D}}$ be a TeSSLa operator. We then define $\hat{o} : \mathcal{P}_{\mathbb{D}_1} \times \mathcal{P}_{\mathbb{D}_2} \times \ldots \times \mathcal{P}_{\mathbb{D}_k} \to \mathcal{P}_{\mathbb{D}}$ as follows:

$$\hat{o}(p_1, p_2, \ldots, p_k) = \{o(s_1, s_2, \ldots, s_k) \mid s_i \in p_i, 1 \leq i \leq k\}.$$ ⌟

The operator $\hat{o}$ is derived by applying o to all possible combinations of input streams. The following examples demonstrate this for the case of a binary operator:

Example 3.66 (TeSSLa Operators on Monitoring Streams)**.** Let $a, b \in \mathcal{P}_\mathbb{N}$ be two monitoring streams given explicitly as sets of streams $a = \{a_1, a_2\}$ and $b = \{b_1, b_2\}$ over the streams $a_1, a_2, b_1, b_2 \in \mathcal{S}_\mathbb{N}$ given as follows:

$$a_1 = \langle(1,1),(3,2)\rangle \qquad b_1 = \langle(1,7),(3,1)\rangle$$
$$a_2 = \langle(1,1),(3,4)\rangle \qquad b_2 = \langle(1,7),(3,3)\rangle$$

Further, let the derived stream $c \in \mathcal{P}_\mathbb{N}$ be defined as

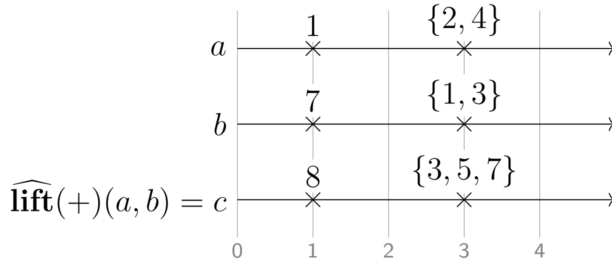
$$c = \widehat{\textbf{lift}}(+)(a,b).$$

The monitoring stream c can be computed explicitly by applying **slift**$(+)$ to all combinations of a and b individually. We get the streams $c_i \in \mathcal{S}_\mathbb{N}$ as follows

$$c_1 = \textbf{lift}(+)(a_1,b_1) = \langle(1,1+7),(3,2+1)\rangle = \langle(1,8),(3,3)\rangle$$
$$c_2 = \textbf{lift}(+)(a_1,b_2) = \langle(1,1+7),(3,2+3)\rangle = \langle(1,8),(3,5)\rangle$$
$$c_3 = \textbf{lift}(+)(a_2,b_1) = \langle(1,1+7),(3,4+1)\rangle = \langle(1,8),(3,5)\rangle$$
$$c_4 = \textbf{lift}(+)(a_2,b_2) = \langle(1,1+7),(3,4+3)\rangle = \langle(1,8),(3,7)\rangle$$

With $c_2 = c_3$ we get $c = \{c_1, c_2, c_4\}$.

The following visualisation depicts the streams a, b and c:



The TeSSLa monitoring semantics is defined analogous to the TeSSLa semantics in Definition 3.23 in Section 3.2.3 with the only difference that on monitoring streams the TeSSLa operators are raised to monitoring streams as defined above:

Definition 3.67 (TeSSLa Monitoring Semantics)**.** Let φ be a TeSSLa specification with k free streams $\boldsymbol{y} = (y_1, y_2, \ldots, y_k)$ and n bound streams $\boldsymbol{z} = (z_1, z_2, \ldots, z_n)$. The equations $z_i = f_i(\boldsymbol{y})(\boldsymbol{z})$ for $1 \le i \le n$ of the specification φ can be raised to monitoring streams by replacing f_i with $\hat{f}_i$. In combination we get

$$\boldsymbol{z} = \hat{\boldsymbol{f}}(\boldsymbol{y})(\boldsymbol{z})$$

with

$$\boldsymbol{f}\colon \mathcal{P}_{\mathbb{D}}^{k} \times \mathcal{P}_{\mathbb{D}'}^{n} \to \mathcal{P}_{\mathbb{D}'}^{n}.$$

The *monitoring semantics* of φ is a function $\hat{\boldsymbol{f}}_{\varphi}\colon \mathcal{P}_{\mathbb{D}}^{k} \to \mathcal{P}_{\mathbb{D}'}^{n}$ given as the least fixed point of $\hat{\boldsymbol{f}}$ over the bound streams $\boldsymbol{z}$:

$$\hat{\boldsymbol{f}}_{\varphi}(\boldsymbol{y}) := \mu \hat{\boldsymbol{f}}(\boldsymbol{y}).$$ ⌟

Two TeSSLa specifications are called monitoring-equivalent if they behave equivalently regarding the monitoring semantics:

Definition 3.68 (Monitoring-Equivalence of TeSSLa Specifications)**.** Two TeSSLa specifications φ and ψ are *monitoring-equivalent* if their semantics functions $\hat{\boldsymbol{f}}_{\varphi}$ and $\hat{\boldsymbol{f}}_{\psi}$ are equivalent:

$$\varphi \mathrel{\hat{\equiv}} \psi :\Longleftrightarrow \hat{\boldsymbol{f}}_{\varphi} \equiv \hat{\boldsymbol{f}}_{\psi}.$$ ⌟

We will show later that two TeSSLa specifications are monitoring-equivalent iff they are equivalent.

From the definitions of the basic TeSSLa operators we can derive:

Lemma 3.69 (Scott-Continuity of the TeSSLa Operators)**.** *The TeSSLa operators* $\widehat{\textbf{unit}}$, $\widehat{\textbf{time}}$, $\widehat{\textbf{lift}}$, $\widehat{\textbf{last}}$ *and* $\widehat{\textbf{delay}}$ *are Scott-continuous with regard to the partial order* $(\mathcal{P}_{\mathbb{D}}, \sqsubseteq)$ *and its extension to tuples of monitoring streams.*

When their extension to monitoring streams is clear from the context, we do not explicitly distinguish between the TeSSLa operators on streams and those on monitoring streams.

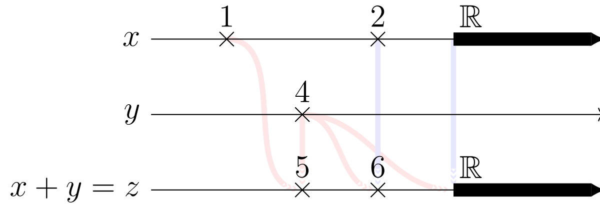
3.5.3. Examples

We start with a simple example demonstrating how an slift can only produce certain events if both input streams are in regions of full knowledge:

Example 3.70 (Signal Lift)**.** Let $x, y \in \mathcal{P}_{\mathbb{R}}$ be two free monitoring streams and let $z \in \mathcal{P}_{\mathbb{R}}$ be a derived monitoring stream given by:

$$z = x + y$$

The following visualisation shows the evaluation of this simple specification on two exemplary input streams:



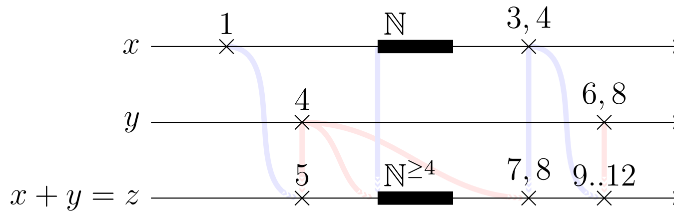
Events on the derived stream are derived from the input streams in the same way as before on streams. The data flow is indicated by the light red and light blue arrows. As soon as the first input stream reaches the region of full uncertainty, the output stream becomes fully uncertain, too. ⌟

Next, we consider a similar example with more complex refinement areas:

Example 3.71 (Signal Lift With More Complex Refinements)**.** Let $x, y \in \mathcal{P}_{\mathbb{N}}$ be two free monitoring streams and $z \in \mathcal{P}_{\mathbb{N}}$ a derived monitoring stream given by:

$$z = x + y$$

The following visualisation shows the evaluation of this specification on two exemplary input streams:



In the above diagram, comma-separated values represent sets, e. g. 3, 4 represents $\{3, 4\}$ and sequences such as 9..12 represent the interval $[9, 12]$ and hence on the data domain $\mathbb{N}$ the set $\{9, 10, 11, 12\}$.

Note how $\{3, 4\} + 4 = \{7, 8\}$ and $\{3, 4\} + \{6, 8\} = \{9, 10, 11, 12\}$ reflects the way how the TeSSLa operations are lifted to sets of streams by applying them to every possible combination of streams.

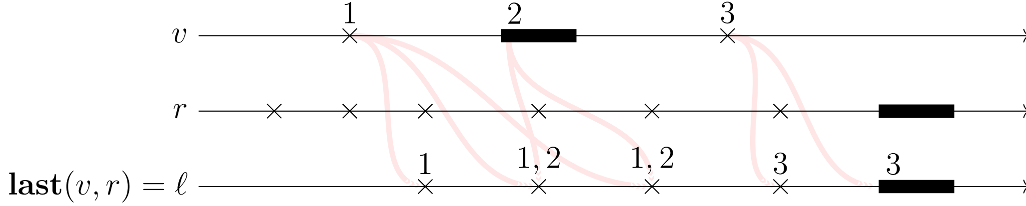
Note that $\mathbb{R} + 4 = \mathbb{R}$ as seen in the previous example, but in this case $\mathbb{N} + 4 = \mathbb{N}^{\geq 4}$. How much refinement happens here on the last part on z highly depends on the data domain. ⌟

The operators **lift** and **slift** studied in the previous examples can be considered symmetric regarding the influence of the input streams' refinement on the refinement of the output stream. The **last** operator is asymmetric in the sense that regions of uncertainty in the input only affects the output when the operator was triggered:

Example 3.72 (**last**). Let $v \in \mathcal{P}_{\mathbb{R}}$ and $r \in \mathcal{P}_{\mathbb{U}}$ be free monitoring streams and the output monitoring stream $\ell \in \mathcal{P}_{\mathbb{R}}$ be given by:

$$\ell = \textbf{last}(v, r)$$

The following visualisation demonstrates the evaluation of this specification on two exemplary inputs:



One can see how the output stream ℓ only has uncertainty regarding the event's values as long as the events on the triggering stream r are known. The possible values reflect which events on the value stream v come into question. As soon as the triggering stream r becomes uncertain, the output becomes uncertain, too, because we no longer know where the events might be. However, we still know that any event in that region can only carry the value 3 because that is the only event on the value stream v which comes into question. ⌟

So far, we only considered specifications, where the application of the operators immediately provides the output streams and hence the application of the fixed point is trivial. Next, we look at how monitoring streams allow us to compute the fixed point defined in the TeSSLa monitoring semantics. In the case of the TeSSLa semantics, we were only able to check if a given solution is a valid fixed point. In the case of the TeSSLa monitoring semantics, we are now using monitoring streams with their refinement relation. Hence we can compute the fixed point starting with the smallest element, i. e. the fully unknown monitoring stream $\mathcal{P}_{\mathbb{D}}$ which is the set of all possible streams. We will discuss the theory behind constructing this fixed point further after the examples in Lemma 3.81.

Example 3.73 (Counting). Let $x \in \mathcal{P}_{\mathbb{U}}$ be a free monitoring stream and $\ell, i, z \in \mathcal{P}_{\mathbb{N}}$ output monitoring streams given by the following recursive specification:

$$\ell = \textbf{last}(z, x)$$
$$i = \textbf{lift}(inc)(\ell)$$
$$z = \mathsf{merge}(i, 0)$$

The function $inc\colon \mathbb{Z} \to \mathbb{Z}$ increments an integer, i. e. $inc(i) = i+1$ for any $i \in \mathbb{Z}$. The visualisation in Figure 3.4 demonstrates how the fixed point of z can be computed starting with the smallest element $z_0 = \mathcal{S}_{\mathbb{N}}$.

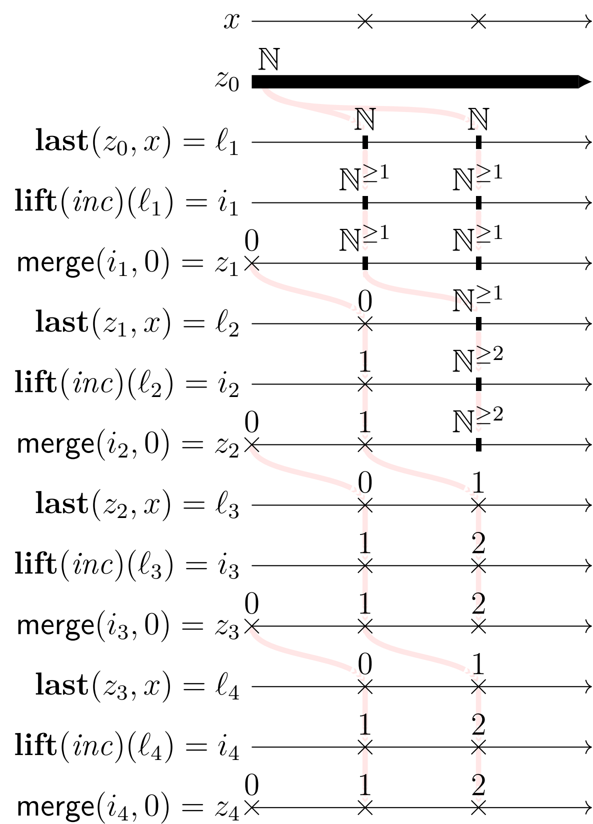


Figure 3.4.: Visualisation how the fixed point of z in Example 3.73 is computed.

With the initial application of the **last** with trigger x to z_0 we get ℓ_1: The monitoring stream ℓ_1 contains no event at those timestamps where x contains no event. When x contains an event, ℓ_1 might contain an event. The very short thick bars are supposed to represent individual timestamps at which the monitoring stream might contain an event or not. The possible values are denoted above as usual.

The merge introduced as a base case for the recursion and the subsequent application of the **last** now has a precise value for the first event.

After several iterations we reach the fixed point $z_4 = z_3$ and can conclude $z = z_3$. ⌟

The following example demonstrates what happens if we remove the merge from the specification in the previous example. Without the base case, the iterative computation of the fixed point step by step converges to the result that only the empty stream is a possible solution:

Example 3.74 (Recursion Without a Base Case)**.** As in the previous example let $x \in \mathcal{P}_{\mathbb{U}}$ be a free monitoring stream and $\ell, z \in \mathcal{P}_{\mathbb{N}}$ output monitoring streams given by the following recursive specification:

$$\ell = \mathbf{last}(z, x)$$
$$z = \mathbf{lift}(inc)(\ell)$$

The visualisation in Figure 3.5 demonstrates how the fixed point of z can be computed starting with the smallest element $z_0 = \mathcal{S}_{\mathbb{N}}$.

Because of $z_4 = z_3$ we reached the fixed point and can conclude $z = z_3$. ⌟

The TeSSLa semantics in Definition 3.23 in Section 3.2.3 is defined as solution of an equation system. However, there may be multiple solutions in general, and thus, this semantics might not be well-defined. In the case of the TeSSLa monitoring semantics, we are using monitoring streams with their refinement relations. Hence, we can order the monitoring streams and explicitly use the least fixed point in the monitoring semantics. We will show in Lemma 3.81 that the least fixed-point exists and can be computed. The following examples demonstrate how the monitoring semantics is well-defined even for not-well-formed specifications:

Example 3.75 (Not-Well-Formed Simple Recursion)**.** We start with a very simple not-well-formed recursion over a single stream $x \in \mathcal{S}_{\mathbb{D}}$:

$$x = x$$

As in the previous examples we start with the least stream:

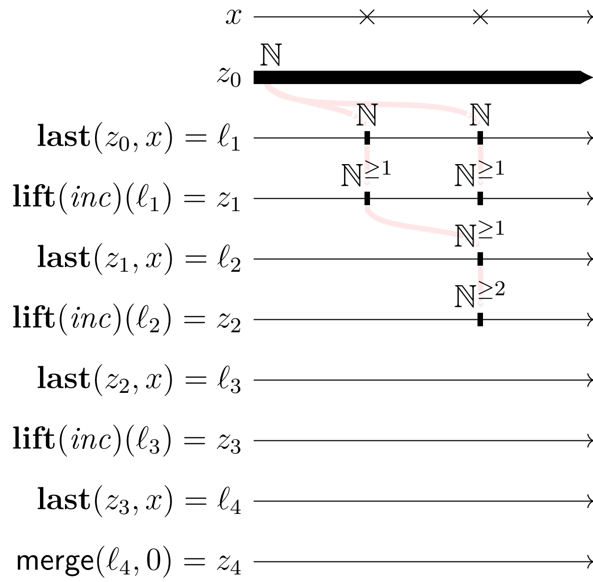


Figure 3.5.: Visualisation how the fixed point of z in Example 3.74 is computed.

Due to the simplicity of the specification, we immediately reached a fixed point. Note that every stream is a fixed point of this specification, but since we start computing fixed points with the least stream, i. e. the set of all possible streams, we get exactly that as our least fixed point.

If we adjust the specification slightly things get more complicated:

$$x = x + 1$$

Now it highly depends on the data domain which fixed point will be reached. For example for $x \in \mathcal{S}_\mathbb{Z}$ we have the same situation as above: The least stream x_0 is immediately our fixed point because $\mathbb{Z} + 1 = \mathbb{Z}$. For $x \in \mathcal{S}_\mathbb{N}$ the situation is slightly different. See Figure 3.6 for the visualisation.

By iteratively removing possible values from the data domain, we end up with the empty stream because, after infinitely many steps, we ruled out all possible values. ⌟

Next, we consider a more advanced example of a not-well-formed specification:

Example 3.76 (Not-Well-Formed Advanced Recursion)**.** Consider the following specification with the streams $z, \ell, i \in \mathcal{S}_\mathbb{Z}$:

$$\ell = \mathbf{last}(z, z)$$

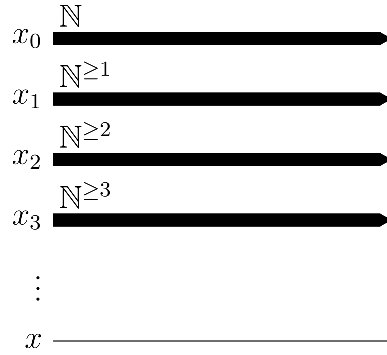


Figure 3.6.: Visualisation of the evaluation of the specification from Example 3.75 with the data domain $\mathbb{N}$.

$$i = \textbf{lift}(inc)(\ell)$$
$$z = \mathsf{merge}(i, 0)$$

As shown in Example 3.34 (Dependency Graph of a Not Well-Formed TeSSLa Specification) in Section 3.2.4 this specification lacks any free stream, i. e. there is no input stream. Its dependency graph contains a cycle without a delayed-labelled edge: This cycle contains only an edge corresponding to the second argument of **last** and thus no delayed-labelled edge. See Figure 3.7 for the visualisation.

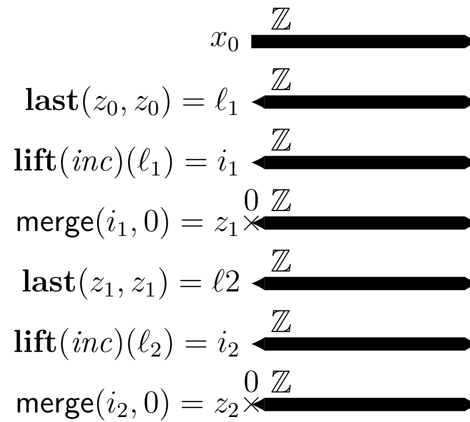


Figure 3.7.: Visualisation of the evaluation of the not-well-formed specification from Example 3.76.

We reached the least fixed point with $z_1 = z_2 = z$. Note how $\textbf{last}(x, x)$ returns a set of streams where every stream does not contain an event at timestamp 0. Fed with this stream, the $\mathsf{merge}(i, 0)$ returns a stream that contains an event with value 0 at timestamp 0 and any possible continuation immediately after the first event.

In the visualisation we introduce the additional notation ◀▬ which indicates a region of uncertainty not including its left boundary. Note the important difference between ▬ and ×◀▬. The cross indicates a definitive event, i. e. every stream in the set contains an event at that timestamp. The thick bar indicates a possible event, i. e. there are at least two streams in the set: One containing an event at that timestamp and one that does not contain an event at that timestamp. So ×◀▬ indicates a definitive event at the timestamp of the cross followed by infinitely many possible events with timestamps strictly greater than the timestamp of the cross. ⌟

If we adjust the previous example and change the data domain from $\mathcal{S}_\mathbb{Z}$ to $\mathcal{S}_\mathbb{N}$, we get an effect similar to the specification $x = x$ discussed in Example 3.75 above:

Example 3.77 (Not-Well-Formed Advanced Recursion Over Discrete Data Domain)**.** Consider again the specification from the previous example but this time with the streams $z, \ell, i \in \mathcal{S}_\mathbb{N}$:

$$\ell = \textbf{last}(z, z)$$
$$i = \textbf{lift}(inc)(\ell)$$
$$z = \mathsf{merge}(i, 0)$$

Every application of $\textbf{lift}(inc)$ removes another value from the set of possible values until after infinitely many steps, no possible values are left, and we get a stream without any events. The only difference is the $\mathsf{merge}(i, 0)$ which always adds an event with value 0 at timestamp 0. See Figure 3.8 for the visualisation.

⌟

We conclude this set of examples for the TeSSLa monitoring semantics with the replication of the final motivating examples from Section 3.1 with the monitoring semantics, which illustrates how additional events are generated with **delay** in recursive equations: The simple period specification from Example 3.10 and the advanced variable frequency period specification from Example 3.11.

Example 3.78 (Period)**.** Let $z, d \in \mathcal{P}_\mathbb{U}$ and $c \in \mathcal{P}_\mathbb{R}$ be derived monitoring streams given by the following specification:

$$c = \mathsf{const}(2, z)$$
$$d = \textbf{delay}(c)$$
$$z = \mathsf{merge}(d, \textbf{unit})$$

As already discussed in Example 3.10 this specification does not use any free streams. All streams are defined by the specification. Hence, the visualisation in Figure 3.9

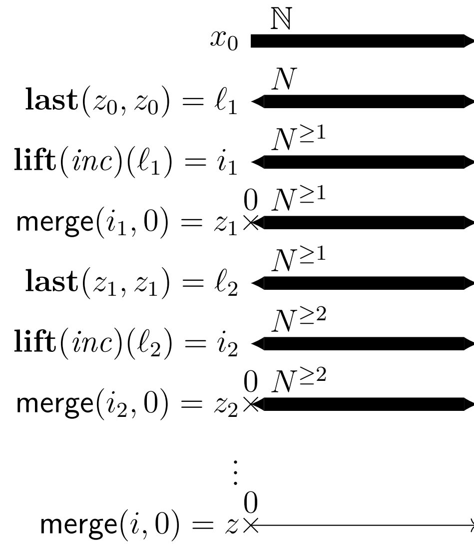


Figure 3.8.: Visualisation of the evaluation of the not-well-formed specification from Example 3.77.

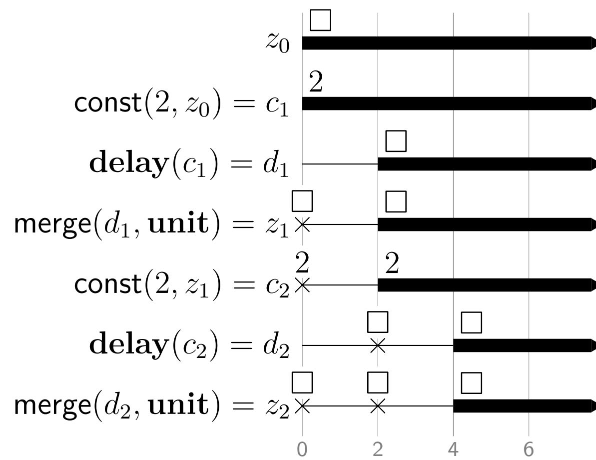


Figure 3.9.: Visualisation of the evaluation of the specification from Example 3.78.

shows not an exemplary evaluation with an arbitrary input but the only possible solution for the specification.

The fixed point computation starts again with the fully unknown stream $z_0 = \mathcal{S}_{\mathbb{U}}$. The **delay** operator produces events without values, so the imprecision is not about the events' values but purely about their existence. The const maps all events' values to 2, and after the first application of the **delay** we know that there cannot be any event before the timestamp 2 because all possible events on its input c_1 have a value of 2. The merge introduces an event at timestamp 0 as a base case similar to the previous examples on recursive specifications with **last**. Now we can iterate this procedure, and with every application of the **delay** we know about another event. In this example, we do not reach a fixed point after a finite number of steps. The fixed point is the monitoring stream which contains infinitely many events with a period of 2 time units. Because the TeSSLa monitoring semantics is Scott-continuous (see Section 3.5.4 below), we reach this fixed point after infinitely many steps. ⌟

For this example, we once more illustrate what happens if a recursive specification does not have a base case:

Example 3.79 (Period-Recursion Without Base Case)**.** By leaving out the merge with a default value at timestamp 0 in the previous example, we can achieve a similar effect as in the recursion with a **last** without a base case in Example 3.74: With every application of the **delay** operator, everything is delayed by 2 time units, but without the addition of a new event at timestamp 0 we end up with the empty stream after infinitely many steps.

Let $z \in \mathcal{P}_{\mathbb{U}}$ and $c \in \mathcal{P}_{\mathbb{R}}$ be monitoring streams given by the following specification:

$$c = \mathsf{const}(2, z)$$
$$z = \mathbf{delay}(c)$$

The visualisation now looks as shown in Figure 3.10.

As before, we only reach a fixed point after infinitely many steps, but the iteration leads to the empty stream this time. ⌟

As a final example we now consider the computation using the monitoring semantics of the variable frequency period specification from Example 3.11 in Section 3.1.5:

Example 3.80 (Variable Frequency Period)**.** Let $x \in \mathcal{P}_{\mathbb{R}+}$ be a free monitoring stream indicating the desired period and let $\ell, z \in \mathcal{P}_{\mathbb{R}+}$ and $d \in \mathcal{P}_{\mathbb{U}}$ be derived monitoring streams given by the following specification:

$$d = \mathbf{delay}(z)$$

$$\ell = \textbf{last}(x, d)$$
$$z = \mathsf{merge}(x, \ell)$$

The stream visualisation in Figure 3.11 shows the evaluation on an exemplary input stream x.

The red 1.5 and the black 1.5 are the same value. They are only coloured differently to indicate the origin of the event.

In this specification, the **delay** is applied directly to z_0 without a const mapping the events' values to a fixed value. Hence d_1 only excludes an event at timestamp 0. More significant progress is then generated by the **last** which is triggered by d_1 to reproduce values from x. Before the first event on x, the **last** does not produce any event. At timestamp 5, the stream ℓ_1 still may contain an event with value 3. For all timestamps greater than 5, the stream ℓ_1 may contain events with a value of 1.5. The merge provides a base case for the recursion by copying the events from x. With the subsequent application of **delay** the event pattern starts with a period of 3. The next event on x interrupts this pattern and adjusts the period to 1.5. Without further events, this pattern goes on forever, and the fixed point is the stream with infinitely many events with a value of 1.5 every 1.5 time units. ⌟

3.5.4. Fixed Points in the Monitoring Semantics

In the previous examples, we have seen how the fixed point used to define the monitoring semantics can be constructed by iteratively applying the function derived from the equation system starting with the fully unknown monitoring stream $z_0 = \mathcal{S}_{\mathbb{D}}$. The following lemma formally states that:

Lemma 3.81 (Construction of the Least Fixed Point)**.** *The least fixed point used in the definition of the TeSSLa monitoring semantics exists and can be constructed:*

$$\hat{\boldsymbol{f}}_\varphi(\boldsymbol{y}) = \mu \hat{\boldsymbol{f}}(\boldsymbol{y}) = \bigvee \{ (\hat{\boldsymbol{f}}(\boldsymbol{y}))^n (\mathcal{S}_{\mathbb{D}_1}, \mathcal{S}_{\mathbb{D}_2}, \ldots, \mathcal{S}_{\mathbb{D}_k}) \mid n \in \mathbb{N} \}$$

Proof. For a given tuple of input streams $\boldsymbol{y} \in \mathcal{P}_{\mathbb{D}}$ the function $\hat{\boldsymbol{f}}(\boldsymbol{y})$ is Scott-continuous because it is built from the TeSSLa operators, and the property is closed under function composition and Cartesian products. With Lemma 3.64 from Section 3.5.1 we know that $(\mathcal{P}_{\mathbb{D}}, \sqsubseteq)$ is a dcpo. By extension $(\mathcal{P}_{\mathbb{D}_1} \times \mathcal{P}_{\mathbb{D}_2} \times \ldots \times \mathcal{P}_{\mathbb{D}_k}, \sqsubseteq \times \sqsubseteq \ldots \sqsubseteq)$ is a dcpo, too. Now the statement follows directly from the Kleene fixed-point theorem. □

Based on that, we can now formulate one of the main theorems regarding the TeSSLa monitoring semantics. As we have already motivated in the previous examples, the TeSSLa monitoring semantics have several advantages over the TeSSLa semantics:

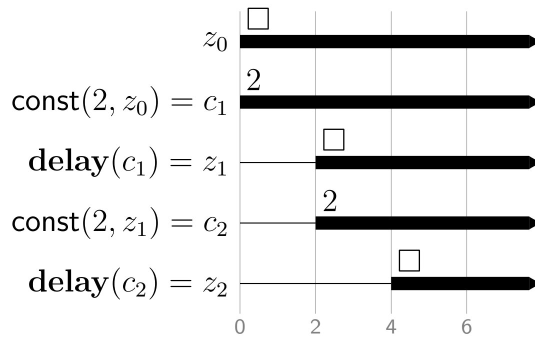


Figure 3.10.: Visualisation of the evaluation of the period-recursion without base case from Example 3.79.

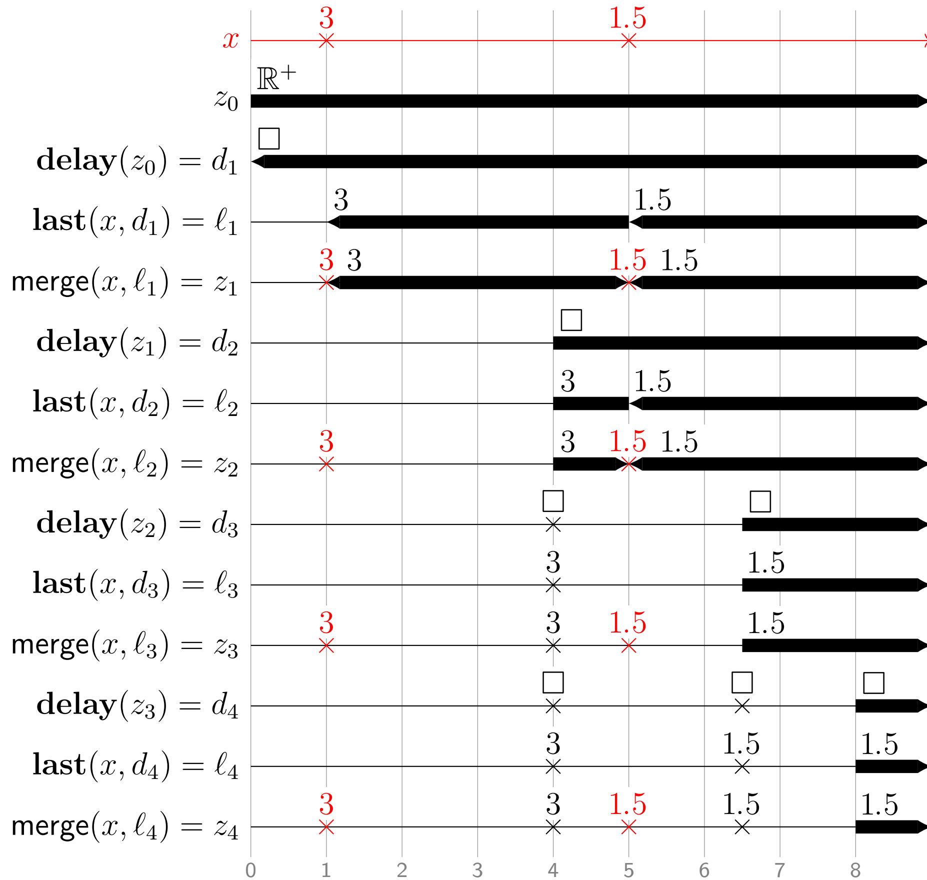


Figure 3.11.: Visualisation of the evaluation of the variable frequency period specification from Example 3.80 on an exemplary input stream x.

1. With the introduction of monitoring streams, we can express incomplete streams. The monitoring semantics can be applied to incomplete streams and thus enable us to do online monitoring.
2. The monitoring semantics are always well-defined because with the introduction of the refinement relation on monitoring streams, the monitoring semantics can be defined as the least fixed point. So for those cases where we had multiple fixed points in the TeSSLa semantics, we now still have a well-defined semantics.
3. As shown in the previous lemma, the fact that the monitoring semantics are defined over monitoring streams allows us to construct the least fixed point.
4. The refinement relation on monitoring streams and the monitoring semantics provides us with the tools we need to prove that the fixed point is unique if the TeSSLa specification is well-formed.

The following theorem states the last point: If the TeSSLa specification is well-formed, then the least fixed point is the only fixed point. The intuitive idea is as follows: We iteratively apply the function given by the equation system of the specification as in the above lemma, i. e. we iteratively build the Kleene chain. We assume that there is always enough new progress provided by the free input streams such that we never reach an early fixed point. The TeSSLa operators are built to reach the maximal fixed point in this case, and thus, the minimal and maximal fixed points are the same.

Theorem 3.82 (Uniqueness of the Fixed Point in the Monitoring Semantics)**.** *If a TeSSLa specification φ is well-formed then the fixed point $\mu\hat{\boldsymbol{f}}(\boldsymbol{y})$ used in the TeSSLa monitoring semantics is unique.*

Proof. By Lemma 3.81 we know that the least fixed point exists. We call that least fixed point $\boldsymbol{a}$. Now assume that another fixed point $\boldsymbol{b}$ exists with $\boldsymbol{a} \neq \boldsymbol{b}$. We now have two fixed points $\boldsymbol{a}, \boldsymbol{b} \in \mathcal{P}_{\mathbb{D}}^n$ with $\boldsymbol{a} \sqsubset \boldsymbol{b}$ because $\boldsymbol{b}$ must be greater than the least fixed point $\boldsymbol{a}$:

$$\hat{\boldsymbol{f}}(\boldsymbol{y})(\boldsymbol{a}) = \boldsymbol{a}$$
$$\hat{\boldsymbol{f}}(\boldsymbol{y})(\boldsymbol{b}) = \boldsymbol{b}$$

We further assume that $\boldsymbol{f}$ was derived from a flat TeSSLa specification because every TeSSLa specification can be rewritten into a monitoring-equivalent flat TeSSLa specification. Now every cycle in the dependency graph corresponds to a sequence of entries in $\hat{\boldsymbol{f}} = (\hat{f}_1, \hat{f}_2, \ldots, \hat{f}_k)$. At least one of these entries must be **last** or **delay** because φ is well-formed. We call that entry $\hat{f}_i \colon \mathcal{P}_{\mathbb{D}}^n \to \mathcal{P}_{\mathbb{D}_i}$. We now have

$$\hat{f}_i(\boldsymbol{a}) = a_i$$

$$\hat{f}_i(\boldsymbol{b}) = b_i$$

with $\boldsymbol{a} \sqsubset \boldsymbol{b}$ and $a_i \sqsubseteq b_i$. Under the assumption that f_i is the only **last** or **delay** forming a delayed-labelled edge on that cycle in the dependency graph, we even have $a_i \sqsubset b_i$. That is a contradiction to f_i being either **last** or **delay** because both operators are defined in a way that they refine their input further on pre-fixed points until the fixed point is reached: An output event at timestamp t is defined in both operators independent of their input streams at t. ☐

We conclude this section with the simple result that the TeSSLa monitoring semantics is Scott-continues. This result follows directly from the observation that the basic TeSSLa operators are defined such that they have this property.

Lemma 3.83 (TeSSLa Monitoring Semantics is Scott-Continuous)**.** *The monitoring semantics $\hat{\boldsymbol{f}}_\varphi$ for a TeSSLa specification φ is Scott-continuous.*

Proof. We already observed in Lemma 3.69 from Section 3.5.2 that the basic operators are Scott-continuous concerning the partial order $(\mathcal{P}_{\mathbb{D}}, \sqsubseteq)$ and its extension to tuples of monitoring streams. Scott-continuity is compositional, and $\hat{\boldsymbol{f}}_\varphi$ is defined as a fixed point given as the Kleene chain of a function that is composed of the basic operators. ☐

3.5.5. Relation to Semantics

In the previous section, we have shown that the fixed point used in the definition of the monitoring semantics is unique. Now we are going to establish some results regarding the relation of the monitoring semantics and the semantics from Section 3.2 which is defined on fully known streams. Finally, we can show that the fixed point in the TeSSLa semantics is unique if the TeSSLa specification is well-formed. The uniqueness of the fixed point is a significant result because it allows us to define the TeSSLa semantics in Definition 3.23 in Section 3.2.3 as this fixed point, and this is well-defined if the TeSSLa specification is well-formed.

First, we categorise functions on monitoring streams that preserve full knowledge. Recall that monitoring streams are called fully known if they are singleton sets because no further refinement is possible. In terms of the refinement relations, those streams are maximal.

Definition 3.84 (Preserving Full Knowledge)**.** We say a function $f\colon \mathcal{P}_{\mathbb{D}}^k \to \mathcal{P}_{\mathbb{D}}^n$ over tuples of monitoring streams *preserves full knowledge* if it maps tuples of fully known streams to tuples of fully known streams, i. e. if the input is a tuple of singleton sets, then the output is a tuple of singleton sets, too. ⌟

Lemma 3.85 (Relation Between TeSSLa Monitoring Semantics and TeSSLa Semantics)**.** *Let φ be a well-formed TeSSLa specification with the semantics $\boldsymbol{f}_\varphi$ and the monitoring semantics $\hat{\boldsymbol{f}}_\varphi$. We then have*

$$\forall\, \boldsymbol{a} \in \mathcal{S}_{\mathbb{D}}^n \colon \hat{\boldsymbol{f}}_\varphi(\boldsymbol{a}) = \boldsymbol{f}_\varphi(\boldsymbol{a}).$$

Proof. It follows from the definition of the TeSSLa monitoring semantics that every fixed point on $\boldsymbol{f}(\boldsymbol{y})$ in the TeSSLa semantics is a fixed point on $\hat{\boldsymbol{f}}(\boldsymbol{y})$ in the TeSSLa monitoring semantics, too. Since φ is well-formed, that fixed point is the only fixed point and, hence, the least. □

As a direct consequence of this lemma, the monitoring semantics preserve full knowledge.

3.5.6. Maximal Refinement

So far, we have established the relation between the monitoring semantics and the semantics on fully known monitoring streams. In order to analyse the case of non-maximal monitoring streams, we introduce a way to judge the quality of a monitoring function based on the refinement of its output: The refinement of its output is maximal if the set of streams does not contain any stream which could already be ruled out given the current input. We say functions on monitoring streams which fulfil this property produce maximal refinement:

Definition 3.86 (Maximal Refinement)**.** We say a function $f \colon \mathcal{P}_{\mathbb{D}}^k \to \mathcal{P}_{\mathbb{D}'}^n$ produces *maximal refinement* iff for every input $\boldsymbol{a} \in \mathcal{P}_{\mathbb{D}}^k$ we have

$$f(\boldsymbol{a}) = \bigcup_{\boldsymbol{s} \in \boldsymbol{a}} f(\boldsymbol{s}). \qquad ⌟$$

This definition relies on monitoring streams $\mathcal{P}_{\mathbb{D}} = 2^{\mathcal{S}_{\mathbb{D}}}$ being sets of streams. A function $f \colon \mathcal{P}_{\mathbb{D}}^k \to \mathcal{P}_{\mathbb{D}'}^n$ on monitoring streams transforms a tuple of monitoring streams into a tuple of monitoring streams. For a concrete tuple $\boldsymbol{a} \in \mathcal{P}_{\mathbb{D}}^k$ of sets of streams, we consider the union over f individually applied to all possible tuples $\boldsymbol{s} \in \boldsymbol{a}$ of streams. The element operator $\in$ is overloaded to tuples by applying it to every element of the tuples individually:

$$(s_1, s_2, \ldots, s_k) \in (a_1, a_2, \ldots, a_k) \Longleftrightarrow \forall\, 1 \leq i \leq k \colon s_1 \in a_i.$$

Since f is defined on tuples of monitoring streams, we can only apply f to a tuple of streams using the implicit conversions from a tuple of streams into a tuple of

monitoring streams which replaces every stream with a singleton set containing that stream.

In other words, a monitoring stream transformation is said to have maximal refinement if its output is the infimum of the function applied to all maximal extensions of its input. A monitoring stream is a common prefix of all its possible extensions. To judge the quality of a monitoring stream transformation, we compare its output on a prefix with its output on all possible extensions of that prefix. It is considered maximal if its output on the prefix is the largest common prefix of its outputs on all possible extensions of that prefix. The largest common prefix of a set of monitoring streams is their infimum.

With this definition being established, we can now show that the TeSSLa monitoring semantics fulfil this property:

Theorem 3.87 (TeSSLa Monitoring Semantics Produces Maximal Refinement)**.** *Let φ be a TeSSLa specification. Then the monitoring semantics $\hat{\boldsymbol{f}}_\varphi \colon \mathcal{P}^k_{\mathbb{D}} \to \mathcal{P}^n_{\mathbb{D}'}$ produce maximal refinement.*

Proof. For a proof by contradiction, we assume that $\hat{\boldsymbol{f}}_\varphi$ does not produce maximal refinement. Then there are tuples of streams $\boldsymbol{a} \in \mathcal{P}^k_{\mathbb{D}}$ and $\boldsymbol{c} \in \mathcal{P}^n_{\mathbb{D}'}$ such that $\boldsymbol{c} \sqsupset \hat{\boldsymbol{f}}_\varphi(\boldsymbol{a})$, i. e. $\boldsymbol{c}$ is a true extension of $\hat{\boldsymbol{f}}_\varphi(\boldsymbol{a})$. For $\hat{\boldsymbol{f}}_\varphi$ to not produce maximal refinement, we need

$$\forall\, \boldsymbol{a}' \sqsupset \boldsymbol{a} \colon \hat{\boldsymbol{f}}_\varphi(\boldsymbol{a}') \sqsupseteq \boldsymbol{c}.$$

So if we consider all outputs generated by applying $\hat{\boldsymbol{f}}_\varphi$ on all extensions of $\boldsymbol{a}$, then all these outputs have a common prefix $\boldsymbol{c}$ which is a true extension of $\hat{\boldsymbol{f}}_\varphi(\boldsymbol{a})$. In other words, $\hat{\boldsymbol{f}}_\varphi(\boldsymbol{a})$ could have been at least $\boldsymbol{c}$ because all extensions of this input lead to extensions of $\boldsymbol{c}$.

Let $\boldsymbol{b} := \hat{\boldsymbol{f}}_\varphi(\boldsymbol{a})$. Since $\boldsymbol{c}$ is a true extension of $\boldsymbol{b}$ there exists an index $1 \leq i \leq n$ and a stream $s \in \mathcal{S}_{\mathbb{D}'_i}$ such that $s \in b_i$ but $s \notin c_i$. So s is contained in $\hat{\boldsymbol{f}}_\varphi(\boldsymbol{a})$, but for every true extension $\boldsymbol{a}' \sqsupset \boldsymbol{a}$ this stream s is no longer present in $\hat{\boldsymbol{f}}_\varphi(\boldsymbol{a}')$ because $\hat{\boldsymbol{f}}_\varphi(\boldsymbol{a}') \sqsupseteq \boldsymbol{c}$ and s is not contained in $\boldsymbol{c}$. In other words, whichever stream is removed from $\boldsymbol{a}$ in order to create its extension $\boldsymbol{a}'$, the stream s is no longer present in $\hat{\boldsymbol{f}}_\varphi(\boldsymbol{a}')$. By the definition of $\hat{\boldsymbol{f}}_\varphi$, this is a contradiction to s being in $\hat{\boldsymbol{f}}_\varphi(\boldsymbol{a})$ because nothing in $\boldsymbol{a}$ can have caused its existence in $\hat{\boldsymbol{f}}_\varphi(\boldsymbol{a})$. □

The above proof is quite technical, but its primary essence is $\hat{\boldsymbol{f}}_\varphi$ being defined by applying the TeSSLa operators individually to all the (possible combinations of) streams contained in a monitoring stream. This construction is basically what

maximal refinement requires from a function on monitoring streams: The output must combine applying all streams in the input individually to the function.

As a final result regarding the relation of the TeSSLa semantics and the monitoring semantics, we can conclude:

Lemma 3.88 (Relation of Equivalence and Monitoring Equivalence)**.** *Two well-formed TeSSLa specifications are equivalent iff they are monitoring equivalent.*

Proof. As a consequence of Lemma 3.85 we get for two well-formed TeSSLa specifications: If they are monitoring-equivalent, then they are equivalent.

Let φ and ψ be two well-formed and equivalent TeSSLa specifications. Since $\hat{\boldsymbol{f}}_\varphi$ and $\hat{\boldsymbol{f}}_\psi$ are Scott-continuous and equivalent on maximal refined inputs, they can only differ in their amount of refinement, but they both produce maximal refinement and are thus equivalent. □

3.5.7. Fixed Points in the Semantics

With the previous results, we can now come back to our goal and finally show the uniqueness of the fixed point in the semantics:

Corollary 3.89 (Uniqueness of the Fixed Point in the Semantics)**.** *If a TeSSLa specification φ is well-formed, then the fixed point $\boldsymbol{z} = \boldsymbol{f}(\boldsymbol{y})(\boldsymbol{z})$ used in the TeSSLa semantics is unique.*

Proof. We have to show that (1) every fixed point of $\boldsymbol{f}$ is unique and (2) that $\boldsymbol{f}$ has a fixed point.

First, for (1), we assume that $\boldsymbol{x} \in \mathcal{S}_{\mathbb{D}}^n$ is a fixed point. By Lemma 3.85 we know that $\boldsymbol{x}$ is a fixed point on the function $\hat{\boldsymbol{f}}(\boldsymbol{y})$ used in TeSSLa monitoring semantics, too. Since that fixed point is unique on $\hat{\boldsymbol{f}}(\boldsymbol{y})$ because φ is well-formed, it must be unique on $\boldsymbol{f}$, too.

For (2), we know that there is a fixed point on $\hat{\boldsymbol{f}}(\boldsymbol{y})$ and because $\hat{\boldsymbol{f}}(\boldsymbol{y})$ preserves maximal refinement and $\boldsymbol{y}$ is maximal, that fixed point also exists on $\boldsymbol{f}(\boldsymbol{y})$. □

Note that we again used the conservative extension from $\boldsymbol{x} \in \mathcal{S}_{\mathbb{D}}^n$ to $\mathcal{P}_{\mathbb{D}}^n$ by replacing every element in the tuple with a singleton set.

3.6. Expressiveness of TeSSLa

In this section, we adopt the results regarding the expressiveness of TeSSLa from [CHL+18] to the more general concept of monitoring streams and the monitoring semantics. The semantics used in [CHL+18] is introduced as abstract monitoring semantics in Chapter 6 and shown to be an abstraction of the monitoring semantics.

In order to formally describe the expressiveness of TeSSLa, we first introduce the notion of future-independent functions on monitoring streams. The intuitive idea of future independence is that a function can produce outputs up to a particular timestamp without having access to events on the input stream located after that timestamp. We start with an auxiliary definition:

Definition 3.90 (Segments of a Monitoring Stream up to a Timestamp)**.** Let $s \in \mathcal{P}_{\mathbb{D}}$ be a monitoring stream. Then the segment of s up to the timestamp t is defined as

$$x|_{\leq t} := \{a \mathbin{\&_{\leq t}} b \mid a \in x \wedge b \in \mathcal{S}_{\mathbb{D}}\},$$

where $c = a \mathbin{\&_{\leq t}} b$ for streams $a, b, c \in \mathcal{S}_{\mathbb{D}}$ is the concatenation of the two streams at the timestamp t given by

$$c(t') = \begin{cases} a(t') & \text{if } t' \leq t, \\ b(t') & \text{otherwise.} \end{cases}$$

⌟

Note that for any monitoring stream $s \in \mathcal{P}_{\mathbb{D}}$ and any timestamp $t \in \mathbb{T}$ we have $s|_{\leq t} \sqsubseteq s$.

We extend this segmentation operator to tuples of streams by applying it individually to every stream in the tuple. Let $\boldsymbol{s} \in \mathcal{P}_{\mathbb{D}}^k$ be a tuple of monitoring streams. We then define for any timestamp $t \in \mathbb{T}$

$$\boldsymbol{s}|_{\leq t} := (s_1|_{\leq t}, s_2|_{\leq t}, \ldots, s_k|_{\leq t}).$$

The following definition is adopted from the corresponding definition in [CHL+18] to monitoring streams:

Definition 3.91 (Future Independence)**.** A function $f\colon \mathcal{P}_{\mathbb{D}}^k \to \mathcal{P}_{\mathbb{D}'}^n$ on monitoring streams is called *future independent* if for all inputs $\boldsymbol{s} \in \mathcal{P}_{\mathbb{D}}^k$ and all timestamps $t \in \mathbb{T}$ we have

$$f(\boldsymbol{s})|_{\leq t} \sqsubseteq f(\boldsymbol{s}|_{\leq t}).$$

⌟

Future independence forbids any causal dependencies between output events and later input events. Changes of input events after a timestamp t must not affect the output until timestamp t, i. e. all outputs must have the same prefix up to timestamp t.

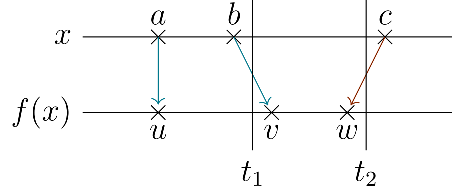


Figure 3.12.: Stream visualisation showing the ■ allowed and ■ forbidden causal dependencies for a future-independent function f on monitoring streams.

The diagram in Figure 3.12 depicts examples for allowed causal dependencies with green arrows and an example for a forbidden causal dependency with a red arrow. $x|_{\leq t_1}$ contains the events a and b and because of the causal dependencies $f(x|_{\leq t_1})$ contains the dependent events u and v, but $f(x)|_{\leq t_1}$ contains only u. This dependency is allowed because $f(x)|_{\leq t_1}$ is a prefix of $f(x|_{\leq t_1})$. In case of the causal dependency indicated with the red arrow $x|_{\leq t_2}$ does not contain the event c and hence $f(x|_{\leq t_2})$ cannot contain w, but $f(x)|_{\leq t_2}$ does contain w. This dependency is forbidden for a future-independent function because the output of such a function must not depend on future events.

The definition of future independence is even stronger because for any stream $s \in \mathcal{P}_{\mathbb{D}}^k$ and any timestamp t it requires the output of $\boldsymbol{s}|_{\leq t}$ to be at least as refined as $f(\boldsymbol{s})|_{\leq t}$. In other words, a function cannot ignore refinement made on its input but must use new inputs immediately to produce new outputs.

Example 3.92 (Future Independence and Refinement)**.** Consider a function that produces its output in chunks of three events. Its output stream is only extended if at least three additional events can be added. This function appears future independent if one only considers the causal dependencies of the events, but it is not future independent regarding the refinement. See Figure 3.13 for an example of such a function: The function f counts the input events and extends the output stream only if at least three additional events can be added. This behaviour is not future independent because, intuitively, the function has to look into the future to decide whether to extend its output stream.

In the diagram in Figure 3.13 one can see that for both depicted timestamps t_i for $i \in \{1, 2\}$ we have $f(x|_{\leq t_i}) \sqsubset f(x)|_{\leq t_i}$ which is the opposite of the required relation for future independence: The output derived from the cut input must be at least as refined as the cut output. ⌟

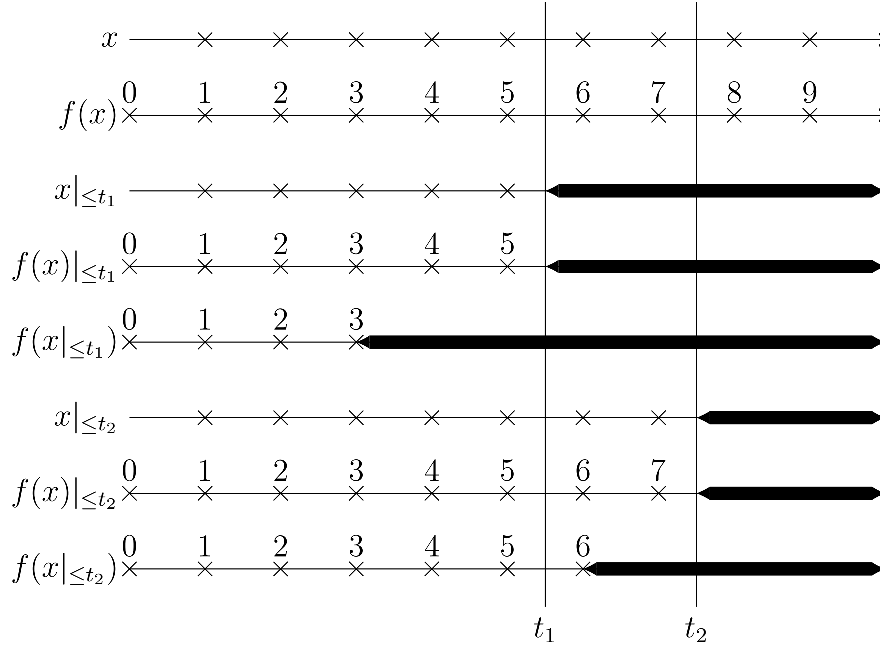


Figure 3.13.: Stream visualisation showing exemplary applications of the function described in Example 3.92, which is not future independent.

On the other hand, generating more refinement than required by future independence is fine.

Example 3.93 (Generating More Refinement is Future Independent)**.** We give an example for a future-independent function, which can generate more refinement on the output then what was present on the input. Consider the following TeSSLa specification with the free stream $x \in \mathcal{S}_{\mathbb{D}}$ and the bound stream $y \in \mathcal{S}_{\mathbb{D}}$:

$$y = \mathsf{filter}(x)(\mathit{false}).$$

The monitoring semantics immediately produce the fully known empty stream independently of the input because the output is the same for every possible input.

The visualisation in Figure 3.14 illustrates how this is future independent because cutting the input at t does not reduce the refinement of the output more than cutting the output at t. ⌟

The examples above only consider Scott-continuous functions because the intuition of future independence is easier to see that way. Nevertheless, this restriction is not required: The property is defined on arbitrary functions on monitoring streams.

To sum up the section on future independence, we can say that a future-independent function must output events based on the available input events. Waiting for future

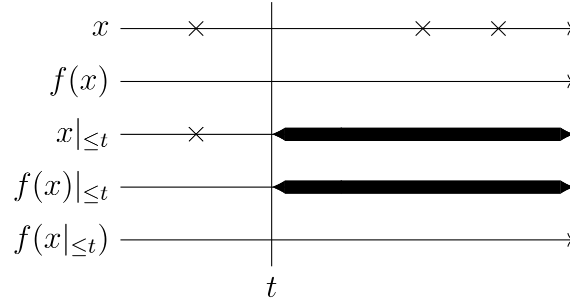


Figure 3.14.: Exemplary application of the future-independent function described in Example 3.93. The function generates more refinement on the output than what was provided in its input.

events is not allowed, but in cases where the function already knows more, it is allowed to output more.

Lemma 3.94 (TeSSLa Monitoring Semantics is Future Independent)**.** *The monitoring semantics* $\hat{\boldsymbol{f}}_\varphi$ *for a TeSSLa specification* φ *is future independent.*

Proof. The basic TeSSLa operators are future independent by definition: **last** and **delay** are the only operators referring to other timestamps, and they only refer to the past. For Scott-continuous functions, future independence is compositional. $\hat{\boldsymbol{f}}_\varphi$ is defined as a fixed point given as the Kleene chain of a function which is composed of the basic operators. □

We have only defined future independence on monitoring streams in this thesis. In order to properly define future independence on streams and show that the TeSSLa semantics are future independent, too, one would need to introduce a notion of cut streams. In this regard, one can see the monitoring streams as a way to prove the future independence of the TeSSLa semantics since they introduce a prefix relation for streams.

A second important property of functions on monitoring streams is timestamp conservatism. A function is called *timestamp conservative* if its output only contains events whose timestamps are either 0 or already appeared in the input. As shown in multiple examples, TeSSLa can generate additional events at arbitrary timestamps with the **delay** operator. We will show that TeSSLa without the **delay** operator can express all timestamp-conservative functions.

The following definition is adopted from the corresponding definition in [CHL+18] to monitoring streams:

Definition 3.95 (Timestamp Conservative)**.** A function $f\colon \mathcal{P}_{\mathbb{D}}^{k} \to \mathcal{P}_{\mathbb{D}'}^{n}$ on monitoring streams is called *timestamp conservative* if for all $\boldsymbol{s} \in \mathcal{P}_{\mathbb{D}}^{k}$ we have

$$T(f(\boldsymbol{s})) \subseteq T(\boldsymbol{s}) \cup \{0\}.$$

Definition 3.96 (Timestamp-Conservative TeSSLa)**.** We call a TeSSLa specification φ *timestamp conservative* if only the operators **lift**, **last**, **time** and the basic stream **unit**, as well as operators derived from these are used.

Lemma 3.97 (Timestamp-Conservative TeSSLa is Timestamp Conservative)**.** *The semantic function* $\hat{\boldsymbol{f}}_{\varphi}$ *of a timestamp-conservative TeSSLa specification* φ *is timestamp conservative.*

Proof. The proof follows directly from the definition of the basic TeSSLa operators: Only the **delay** operator can introduce new timestamps. □

Next, we adopt the corresponding statement in [CHL+18] to monitoring streams:

Lemma 3.98 (Expressiveness of Timestamp Conservative TeSSLa)**.** *For a function* $f\colon \mathcal{P}_{\mathbb{D}}^{k} \to \mathcal{P}_{\mathbb{D}}^{n}$ *on monitoring streams there exists a timestamp-conservative TeSSLa specification* φ *such that* $\hat{\boldsymbol{f}}_{\varphi} \equiv f$ *iff*

a) f *is Scott-continuous and preserves full knowledge,*
b) f *has maximal refinement,*
c) f *is future independent, and*
d) f *is timestamp conservative.*

Proof. Since f is continuous, a finite input prefix is sufficient to compute a finite output prefix. Using this, we can represent f as a step function $\tilde{f}$ which takes an aggregated memory value, the value of all input streams at the current timestamp t (or $\perp$ if there is no value at t on that stream) and that current timestamp t. For simplicity, the step function outputs only a new memory value fed into the subsequent application of the step function. The initial value for the memory is $\perp$. The functions $\tilde{o}_i$ map the current memory value to the output stream's values at the current timestamp. The following TeSSLa specification is equivalent to the function f mapping input streams x to output streams y:

$$\begin{aligned}
t &= \mathbf{time}(\mathsf{merge}(\mathbf{unit}, x_1, \dots, x_k)) \\
m &= \mathbf{lift}(\tilde{f})(\mathbf{last}(m, t), x_1, \dots, x_k, t) \\
y_i &= \mathbf{lift}(\tilde{o}_i)(m) \quad \forall\, i \leq n
\end{aligned}$$

Specifying the output event streams only based on this aggregated memory is sufficient because f is Scott-continuous. Further, because f preserves full knowledge, we know that this stepwise iteration can continue until all inputs are processed. The maximal refinement of f guarantees us that f and the monitoring semantics of the above TeSSLa specification are not only equivalent on fully known streams but behave equivalent on non-maximal prefixes, too, because the TeSSLa monitoring semantics are also of maximal refinement. Since the memory value domain is unbound, the step function $\tilde{f}$ can store arbitrary information. Because f is future independent, it is sufficient to provide $\tilde{f}$ only access to past events. Finally, it is sufficient to evaluate $\tilde{f}$ for the timestamp 0 and every following timestamp in the input events because f is timestamp conservative.

The other direction follows immediately because we already know that the monitoring semantics for every timestamp-conservative TeSSLa specification fulfils the properties a) to d). □

Note that the step function $\tilde{f}$ and the output mapping functions $\tilde{o}_i$ in the above proof are a way to describe the relationship between the input and the output streams. Since f and the monitoring semantics of the TeSSLa specification used in the proof are both of maximal refinement, they are not equivalent to executing the step function in a strictly synchronous way for one timestamp after another. A synchronous execution would require the input to be available synchronously, but the theorem and its proof consider arbitrary functions on monitoring streams. The step function works for not synchronised input streams, too, because it is applied to streams using the monitoring semantics of the **lift** operator.

Next, we extend the previous statement to include functions on monitoring streams that are not timestamp-conservative. For the construction, we now need the **delay** operator to generate events at additional timestamps.

The following definition is adopted from the corresponding definition in [CHL+18] to monitoring streams:

Lemma 3.99 (Expressiveness of TeSSLa)**.** *For a function $f\colon \mathcal{P}_{\mathbb{D}}^k \to \mathcal{P}_{\mathbb{D}}^n$ on monitoring streams there exists a TeSSLa specification φ such that $\hat{\boldsymbol{f}}_\varphi \equiv f$ iff*

a) f is Scott-continuous and preserves full knowledge,
b) f has maximal refinement, and
c) f is future independent.

Proof. We prove the statement by extending the construction presented in the previous proof for Lemma 3.98. The only difference is the omission of the restriction of the function f to be timestamp conservative. As a result, it is no longer sufficient to evaluate the step function $\tilde{f}$ for timestamp 0 and every timestamp which appeared

in the input streams. Additionally, we introduce a delay function $\tilde{u}$, which computes, based on the memory, a delay when the subsequent evaluation of the step function must take place. The delay computed by such a function can be fed directly into the **delay** operator whose output is then used as an additional trigger to compute all timestamps:

$$\begin{aligned} t &= \mathbf{time}(\mathsf{merge}(\mathbf{unit}, x_1, \dots, x_k, d)) \\ d &= \mathbf{delay}(\mathbf{lift}(\tilde{u})(m)) \\ m &= \mathbf{lift}(\tilde{f})(\mathbf{last}(m, t), x_1, \dots, x_k, t) \\ y_i &= \mathbf{lift}(\tilde{o}_i)(m) \quad \forall\, i \leq n \end{aligned}$$

Again the other direction follows immediately because we already know that the monitoring semantics for every TeSSLa specification fulfils the properties a) to c). □

Note how that proof utilises all of the basic TeSSLa operators.

In Lemma 3.99 (Expressiveness of TeSSLa) and Lemma 3.98 (Expressiveness of Timestamp Conservative TeSSLa) above, the function's requirement to have maximal refinement might seem like a relatively strong limitation. With the following definition of behavioural equivalence, we introduce a slightly weaker equivalence on functions on monitoring streams which is of practical relevance and allows us to lift this restriction from the expressiveness results. A similar concept is introduced in [Sch20] but only on the streams which are introduced as abstract monitoring streams in Chapter 6 and only on Scott-continuous functions.

Definition 3.100 (Behavioural Equivalence)**.** Let $f, g \colon \mathcal{P}_{\mathbb{D}}^k \to \mathcal{P}_{\mathbb{D}}^n$ be functions on monitoring streams. They are *behavioural equivalent* if

$$\forall\, \boldsymbol{a} \in \mathcal{P}_{\mathbb{D}}^k \colon f(\boldsymbol{a}) \sqsubseteq g(\boldsymbol{a}) \vee g(\boldsymbol{a}) \sqsubseteq f(\boldsymbol{a}).$$ ⌟

For Scott-continuous functions on monitoring streams, this can be simplified as follows: They are behavioural equivalent if they are equivalent on fully known streams, i. e. singleton sets.

Lemma 3.101 (Behavioural Equivalence of Scott-Continuous Functions)**.** *Let the functions $f, g \colon \mathcal{P}_{\mathbb{D}}^k \to \mathcal{P}_{\mathbb{D}}^n$ on monitoring streams be Scott-continuous and preserve full knowledge. They are behavioural equivalent iff*

$$\forall\, \boldsymbol{a} \in \mathcal{S}_{\mathbb{D}}^k \colon f(\boldsymbol{a}) = g(\boldsymbol{a})$$

Note that the above statements quantify over streams and not over monitoring streams. We implicitly convert streams into monitoring streams by inserting them into singleton sets, representing fully known streams.

Proof. The definition of behavioural equivalence directly implies the weaker statement in the lemma. On Scott-continuous functions, the statement in the lemma implies the definition of behavioural equivalence because, for fully known streams, the statements are equivalent, and, for prefixes of fully known streams, the functions' output always is a prefix of their output on fully known streams. □

With this simplification, one can see why the definition of behavioural equivalence provides a reasonable way to compare functions on monitoring streams: We only consider Scott-continuous functions, which are full knowledge preserving. On those functions, it is sufficient to compare their behaviour only on fully known streams. Their behaviour on other inputs is restricted by their behaviour on fully known streams: Every incomplete input is a prefix of a fully known stream. If the input is extended, then the output can only be an extension of the earlier output, too, and finally, the output must reach the fully known output. So behavioural equivalence states that eventually, the two functions produce the same output. They only differ in how early they provide the output, which can be argued to be an implementation detail.

By combining Lemma 3.98 (Expressiveness of Timestamp Conservative TeSSLa) and Lemma 3.99 (Expressiveness of TeSSLa) with Definition 3.100 (Behavioural Equivalence) we can conclude this section with the following theorem:

Theorem 3.102 (Expressiveness of TeSSLa)**.** *For a function $f\colon \mathcal{P}_{\mathbb{D}}^{k} \to \mathcal{P}_{\mathbb{D}}^{n}$ on monitoring streams there exists a TeSSLa specification φ such that $\hat{\boldsymbol{f}}_{\varphi}$ and f are behavioural equivalent iff*

a) f is Scott-continuous and preserves full knowledge, and
b) f is future independent.

The TeSSLa specification φ can be timestamp conservative iff f is timestamp conservative.

3.7. Conclusion

We have defined TeSSLa as an equation system over streams in this chapter. TeSSLa operators are applied to streams in the equations TeSSLa operators are either one of the very few basic operators or derived operators defined in terms of the basic operators. The semantics is given in terms of a fixed point over the equations.

The monitoring streams add the concept of incompletion to streams which is essential for online monitoring. We have a prefix relation on the monitoring streams called refinement relation because a monitoring stream is a prefix of a large stream if the latter is a refinement of the former. The monitoring semantics are defined in terms of the least fixed point over the equation because the refinement relation provides an order on the fixed points. The monitoring semantics is a semantics for incomplete input streams. It is a way to construct the fixed point, and it allows us to prove that the fixed point of the TeSSLa semantics is unique if the specification is well-formed.

We have established a relation between the monitoring semantics and the semantics and can conclude that the TeSSLa monitoring semantics $\hat{\boldsymbol{f}}_\varphi(\boldsymbol{y})$ is very well suited for online monitoring: It is Scott-continuous, preserves full knowledge, is related to the semantics function, and the output is as refined as possible on not fully known input streams.

Producing maximal refinement is a great property, but unfortunately it is unfeasible for efficient implementations: A monitoring stream is a potentially infinite set of streams, and we can lift arbitrary functions from the data domain to streams. Hence computing the output with the best refinement possible for an incomplete input can become very challenging. We will consider two abstractions of the monitoring semantics for actual implementations that no longer produce maximal refinement. The synchronous TeSSLa semantics introduced in Section 4.1 for the interpreter can be seen as the simplest possible abstraction of the monitoring semantics that still computes something. The abstract monitoring semantics introduces in Chapter 6 for the FPGA synthesis are on the opposite side of the spectrum: They have the maximal refinement per operator, which is the best one can do while staying compositional, i. e. the individual operators can be synthesised individually without the need for global consideration.

As motivated in the introduction, we only want to consider functions that can be implemented with finite memory for online monitoring with stream transformations. We precisely defined the set of functions on stream transformations which we consider as those which are monotonic, continuous and preserve full knowledge and are future independent. We have demonstrated in Section 3.4 on design choices that these functions are not in general implementable with finite memory, but the memory usage is made explicit by the data domain. These functions are precisely the functions that can be expressed in TeSSLa.

4 Interpreter and Software Compiler

This chapter presents a TeSSLa interpreter and software compiler. The interpreter is the first implementation considered in this thesis and the most straightforward implementation. Due to its simplicity, the interpreter will serve as a reference implementation used to test the more complex implementations following. Further, it will serve as a baseline for the efficiency benchmark. The interpreter does not contain any optimisations regarding the processing speed.

The software compiler is a faster approach to execute a TeSSLa specification in software. It compiles the TeSSLa specification into imperative code. Local variables represent the current value of the streams and the memory cells of the operators. This code is close to a manual implementation and can profit from compile-time optimisation performed by different backends, e. g. the LLVM compiler [Lat02, LA04, Lat12] or the Java VM JIT.

For simplicity, both solutions do not implement the monitoring semantics but a simpler abstraction of that semantics. This abstraction is synchronous, i. e. there is a global current timestamp. As already mentioned in the introduction, the reasonable abstractions of the monitoring semantics can be categorised by their progress. Due to its synchronous nature, the semantics introduced in this chapter has the least possible progress. The input streams are synchronised on their timestamps, and hence the monitors only perform the next step if data for all input streams for the next timestamp is available. In a single step, the monitors only compute the synchronous output stream for the current timestamp, too. So the output streams do not contain any information about future events. We will show in this chapter that this is the minimal progress that a function on monitoring streams must generate to be future independent.

The synchronisation of the input streams transforms a tuple of streams into a stream of tuples. The principle of using explicit timestamps to encode events from the asynchronous environment still applies naturally: Timestamps can be arbitrary precise, and the synchronised stream contains only those timestamps with an event on at least one of the individual streams.

The synchronous approaches evaluate the flow graph of the TeSSLa specification for the current global timestamp. Section 4.1 gives a formal semantics for this setting: It introduces synchronised streams and the synchronised monitoring function on those

streams as an abstraction of the monitoring semantics on monitoring streams. We define synchronous semantics for every TeSSLa operator. An operator is provided with the current timestamp and the current evaluation of their dependent streams for that timestamp. It provides its current output for that timestamp. Further, operators can store values in a memory cell. The TeSSLa specification is future independent, so it is sufficient to store past events. As discussed in Section 3.4.5 the TeSSLa operators are built such that it is sufficient to store a single data value for every operator.

Section 4.2 discusses the different approaches to implement the synchronous semantics in the interpreter and the compiler. The interpreter dynamically builds an object graph representing the flow graph of the TeSSLa specification at runtime. It evaluates the flow graph straightforwardly by sending actual messages along the edges of the flow graph. The compiler linearises the flow graph into sequential imperative code. Details of the interpreter are given in Section 4.3, and the compiler is discussed in Section 4.4.

The synchronous monitoring is based on a widespread event-driven synchronous execution scheme which is described in [BCE+03] using the pseudocode shown in Figure 4.1. This principle is used in many other synchronous stream languages like for example LOLA [DSS+05] and RTLola [FFS+19, BFST20, BFST19, FOPS20].

```
Initialise Memory
for each input event do
  Compute Outputs
  Update Memory
end
```

Figure 4.1.: The common event driven synchronous execution scheme as shown in [BCE+03, Figure 1].

The synchronous monitoring function for TeSSLa adjusts and extends this scheme in two aspects:

- TeSSLa has explicit timestamps, which is an adjustment to the step or instant based semantics: The TeSSLa semantics refer to the event's timestamps, and the synchronous execution provides access to a global current timestamp.
- With the notion of explicit timestamps of arbitrary precision comes the possibility to generate events at additional timestamps. The loop body is executed for every timestamp in the input stream and for every additional timestamp generated by the specification.

It is sufficient to evaluate the flow graph for every timestamp occurring in the input stream for timestamp-conservative specifications. In order to evaluate specifications that can generate events at additional timestamps, the operators can give feedback when they should be evaluated next. This feedback of the operators is then used in the global evaluation loop to determine for which global timestamps the flow graph should be evaluated.

4.1. Semantics

This section gives formal semantics for the synchronised setting used by the interpreter and the software compiler. First, we formalise the concept of the progress of a monitoring stream. The amount of progress generated is used to compare different semantics. Next, we define synchronised streams, followed by the actual semantics on these streams. The semantics is given in the form of operator functions for every basic TeSSLa operator and synchronised monitoring function combining these operator functions into a semantics for a given TeSSLa specification.

From this chapter on, we use a common data domain $\mathbb{D}$ for the theory. The typing discussed in detail in the previous chapters can still be applied, but it simplifies the function signatures if we consider the typing and type checking already done in a separate previous step. For further simplification, we assume that the common data domain is a superset $\mathbb{D} \supseteq \mathbb{T}$ of the common time domain $\mathbb{T}$.

4.1.1. Progress

The progress of a monitoring stream is the timestamp up to which we already know everything about the stream. After that timestamp, we might still know something about the monitoring stream, but some information is missing that is not yet available to the monitor in the setting of online monitoring. We will use the formal definition of the progress later in this section to show that the synchronous monitoring yields exactly the minimal progress that a function on monitoring streams must generate to be future independent.

We extend the order on time domains to the set $\mathbb{T}_\infty = \mathbb{T} \cup \{\infty\}$ as follows:

$$\forall\, t \in \mathbb{T} \colon t < \infty.$$

Definition 4.1 (Progress of a Monitoring Stream)**.** Let $s \in \mathcal{P}_\mathbb{D}$ be a monitoring stream. Then let $t \in \mathbb{T}_\infty$ be the maximal timestamp such that for all $t' < t$ and any two streams $r, r' \in s$ we have $r(t') = r'(t')$. We call the minimum of t and $\mathrm{limT}(s)$

the *progress* of s. If the progress is t and $r(t) = r'(t)$ holds, too, we call the progress *inclusive*, otherwise *exclusive*. ⌟

Every monitoring stream has either an exclusive or an inclusive progress. Although every inclusive progress is technically also an exclusive progress, we only consider the maximal progress of a monitoring stream. See Example 4.8 (Abstraction and Concretisation for Synchronised Streams) in Section 4.1.2 for an elaborate example on exclusive and inclusive progress.

If a monitoring stream s has Zeno behaviour, its progress is capped by the limit $\mathrm{limT}(s)$ towards which the timestamps converge. Since this limit is never reached, this progress is an exclusive one.

For a tuple of monitoring streams $\boldsymbol{s} \in \mathcal{P}_{\mathbb{D}}^k$, we call a timestamp $t \in \mathbb{T}_\infty$ the *minimal progress* of $\boldsymbol{s}$ if it is the minimal progress of all streams in $\boldsymbol{s}$. An exclusive progress is considered smaller than an inclusive progress. The minimal progress is either inclusive or exclusive, depending on its origin.

The following relation between prefix and progress follows directly from the definitions: Let $s, s' \in \mathcal{P}_{\mathbb{D}}$ be two monitoring streams with progresses $t, t' \in \mathbb{T}_\infty$, respectively. Then the relation $s \sqsubseteq s'$ implies $t \leq t'$, and the strict relation $s \sqsubset s'$ implies strictly less progress, where again, an exclusive progress is considered smaller than an inclusive progress.

The following lemma states that future-independent functions at least preserve the minimal progress. They might generate more progress on their output than they had on their input but cannot produce less progress than they had on their input. If the function is defined on tuples of streams, then the statement applies to the minimal progress of the tuple.

Lemma 4.2 (Future Independence Preserves Progress)**.** *Let $f\colon \mathcal{P}_{\mathbb{D}}^k \to \mathcal{P}_{\mathbb{D}}^n$ be a function on monitoring streams with*

a) f being Scott-continuous and preserving full knowledge, and
b) f being future independent

then for any $\boldsymbol{s} \in \mathcal{P}_{\mathbb{D}}^k$ either

- *the minimal progress of $f(\boldsymbol{s})$ is at least the minimal progress of $\boldsymbol{s}$, or*
- *the timestamps of $f(\boldsymbol{s})$ converge to a limit* $\mathrm{limT}(\boldsymbol{s})$ *smaller than the minimal progress of $\boldsymbol{s}$.*

Proof. Let $f\colon \mathcal{P}_{\mathbb{D}} \to \mathcal{P}_{\mathbb{D}}$ be a function on monitoring streams fulfilling the properties a) and b) and $s \in \mathcal{P}_{\mathbb{D}}$ be a monitoring stream with inclusive progress $t \in \mathbb{T}$. For the sake of the contradiction let $t' \in \mathbb{T}$ with $t' < t$ be the inclusive progress of $f(s)$.

Let $u \in \mathcal{S}_{\mathbb{D}}$ be a fully known extension of s, i. e. $s \sqsubseteq u$. From a), we know that $f(u) \in \mathcal{S}_{\mathbb{D}}$ is fully known, too. We have $u|_{\leq t} \sqsubseteq s$ because $u|_{\leq t}$ is the smallest prefix of u with inclusive progress t. With a) it follows that $f(u|_{\leq t}) \sqsubseteq f(s)$. Hence, we know that the progress of $f(u|_{\leq t})$ is at most t'. The progress of $f(u)|_{\leq t}$ is t and with b) we know that $f(u)|_{\leq t} \sqsubseteq f(u|_{\leq t})$. That implies $t \leq t'$ which is a contradiction to the assumption. □

Intuitively the above proof states that f cannot be future independent if the progress of $f(s)$ is less than s because that would imply that the extension of $f(s)$ depends on events on s in the future. The only exception is a function f that generates a Zeno stream that converges before its input's minimal progress is reached.

4.1.2. Synchronised Streams

A synchronised stream is an abstraction of a tuple of monitoring streams. The synchronisation transforms a tuple of monitoring streams with individual events into a stream of tuples. Hence the synchronised stream contains all the timestamps at which any represented monitoring streams have an event. The synchronised stream cannot represent different possible continuations with the same expressiveness of the monitoring streams. It only contains events up to a single common timestamp for all the represented streams:

Definition 4.3 (Synchronised Stream)**.** A *synchronised stream* $s \in \mathcal{Q}_n$ of size n is a finite or infinite sequence over timestamped n-tuples:

$$\mathcal{Q}_n := (\mathbb{T}_\infty \times \mathbb{D}^n_\perp \cup \{_\})^\infty.$$

The timestamps in s must be strictly increasing, and the timestamp ∞ is only allowed with the value symbol $_$. ⌟

The symbol $\perp$ encodes the absence of an event on a particular stream at a particular timestamp. Consequently, the tuple $\perp = (\perp, \perp, \ldots, \perp)$ encodes the absence of any event at that timestamp. The symbol $_$ encodes explicit progress, i. e. the absence of events since the last timestamp up to but not including the current timestamp. In practical implementations, there is usually no need to encode progress explicitly. One can introduce an additional stream whose events are not used by the specification but whose timestamps encode inclusive progress. However, exclusive progress cannot be encoded this way, which is why it is explicitly included in the above definition in order to preserve the progress in the abstraction (see Definition 4.5 below). With these conventions, we can now formally define the progress of a synchronised stream:

Definition 4.4 (Progress of a Synchronised Stream)**.** Let $q \in \mathcal{Q}_k$ be a synchronised stream. Then the *progress* of q is the supremum of all timestamps used in q if it exists, or ∞ otherwise. If q ends in $(t, _)$ or if there is no maximal timestamp we call the progress *exclusive*, otherwise *inclusive*. ⌟

For the simple case of a synchronised stream with finitely many events, the progress of that synchronised stream is the timestamp of its last event. Since the timestamps of a synchronised stream are strictly increasing, its last timestamp is the largest one. For synchronised streams with infinitely many events, there is no last event. In that case, we use the least upper bound of the timestamps, i. e. their supremum, if the timestamps converge towards a limit. Otherwise, if the timestamps do not converge, the progress is defined as ∞.

The abstraction function α encodes a tuple of monitoring streams as a synchronised streams:

Definition 4.5 (Abstraction Function for Synchronised Streams)**.** Let $\boldsymbol{s} \in \mathcal{P}_{\mathbb{D}}^k$ be a tuple of monitoring streams. Then let $q \in \mathcal{Q}_k$ be given by

$$q = \langle (t_0, \boldsymbol{s}(t_0)), (t_1, \boldsymbol{s}(t_1)), \ldots \rangle,$$

where $\langle t_0, t_1, \ldots \rangle$ is the ordered sequence of all timestamps in $T(\boldsymbol{s})$ that are lower than (or equal to) the minimal exclusive (or inclusive) progress t of $\boldsymbol{s}$ and

$$\boldsymbol{s}(t) = (s_1(t), s_2(t), \ldots, s_k(t)).$$

Then the abstraction function $\alpha \colon \mathcal{P}_{\mathbb{D}}^k \to \mathcal{Q}_k$ is given by

$$\alpha(\boldsymbol{s}) = \begin{cases} q \,\&\langle (t, \bot) \rangle & \text{if } |q| < \infty \text{ and } t \notin T(\boldsymbol{s}) \text{ and } t \text{ inclusive,} \\ q \,\&\langle (t, _) \rangle & \text{if } |q| < \infty \text{ and } t \text{ exclusive,} \\ q & \text{otherwise.} \end{cases}$$

⌟

A tuple of monitoring streams $\boldsymbol{s} \in \mathcal{P}_{\mathbb{D}}^k$ may contain multiple points where the individual streams show Zeno behaviour, i. e. they have infinitely many events whose timestamps converge towards a finite limit. On the other hand, the sequence $\langle t_0, t_1, \ldots \rangle$ can only contain the first of these limits because it is a sequence. However, this is not a limitation because the sequence $\langle t_0, t_1, \ldots \rangle$ contains only those timestamps up to the minimal progress of $\boldsymbol{s}$. If a monitoring stream is Zeno, its progress cannot be more than the limit its timestamps are progressing towards, and that progress is always exclusive because the timestamps never reach the limit.

The concretisation function γ decodes a synchronised stream back into a tuple of monitoring streams:

Definition 4.6 (Concretisation Function for Synchronised Streams)**.** Let $q \in \mathcal{Q}_k$ be a synchronised stream of size k. Then let $t \in \mathbb{T}_\infty$ be the progress of q and $\boldsymbol{s} \in \mathcal{P}_{\mathbb{D}}^k$ any tuple of monitoring streams such that $\alpha(\boldsymbol{s}) = q$. The concretisation function $\gamma\colon \mathcal{Q}_k \to \mathcal{P}_{\mathbb{D}}^k$ is given by

$$\gamma(q) = \begin{cases} \boldsymbol{s}|_{\leq t} & \text{if } t \text{ is inclusive,} \\ \boldsymbol{s}|_{<t} & \text{otherwise.} \end{cases}$$

In the above definition, we assume $\boldsymbol{s}|_{\leq\infty} := \boldsymbol{s}$ to simplify the notation. There are many different possibilities to choose an $\boldsymbol{s} \in \mathcal{P}_{\mathbb{D}}^k$ such that $\alpha(\boldsymbol{s}) = q$, but since we only consider $\boldsymbol{s}|_{\leq t}$ the function γ is well-defined.

As a direct consequence of the above definitions, we get:

Lemma 4.7 (Abstraction Preserves Progress)**.** *For any tuple of monitoring streams $\boldsymbol{s} \in \mathcal{P}_{\mathbb{D}}^k$, the minimal progress of $\boldsymbol{s}$ is the same as the progress of $\alpha(\boldsymbol{s})$.*

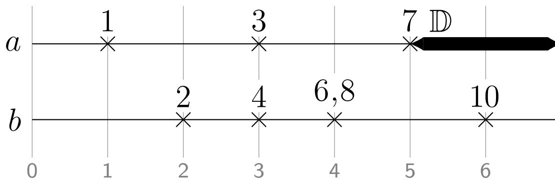
For any synchronised stream $q \in \mathcal{Q}_k$, the progress of q is the same as the minimal progress of $\gamma(q)$.

In other words, for any tuple of streams $\boldsymbol{s} \in \mathcal{P}_{\mathbb{D}}^k$, the minimal progress of $\boldsymbol{s}$ and of $\gamma(\alpha(\boldsymbol{s}))$ are the same. However, the progress of all individual streams in the tuple $\gamma(\alpha(\boldsymbol{s}))$ is the same after the concretisation. So for streams in $\boldsymbol{s}$ whose individual progress is larger than the minimal progress of the tuple $\boldsymbol{s}$ the progress of their corresponding stream in the tuple $\gamma(\alpha(\boldsymbol{s}))$ is smaller.

Example 4.8 (Abstraction and Concretisation for Synchronised Streams)**.** As an example consider the following monitoring streams $a, b \in \mathcal{P}_{\mathbb{D}}$

$$\begin{aligned} a &= \{\langle(1,1),(3,3),(5,7)\rangle\}|_{\leq 5} \\ b &= \{\langle(2,2),(3,4),(4,6),(6,10)\rangle, \\ &\qquad \langle(2,2),(3,4),(4,8),(6,10)\rangle\} \end{aligned}$$

The monitoring stream a consists of infinitely many streams all being identical up to and including timestamp 5 and every possible continuation afterwards. The monitoring stream b consists of two nearly identical streams with a difference in the value of the event at timestamp 4. Both monitoring streams are depicted in the following stream diagram using the syntax introduced in the last chapter:



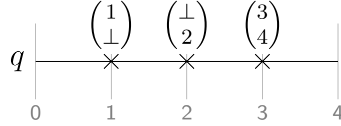
The following timestamps appear in the tuple $\boldsymbol{s} = (a, b)$ of monitoring streams:

$$T(\boldsymbol{s}) = \{1, 2, 3, 4\} \cup \{t \geq 5 \mid t \in \mathbb{T}\}.$$

The monitoring stream a has an inclusive progress of 5 and b has an exclusive progress of 4. Hence, the tuple $\boldsymbol{s}$ has a minimal exclusive progress of 4. For the abstraction $\alpha(\boldsymbol{s})$ we consider the sequence 1 2 3 of timestamps. We get the synchronised stream $q \in \mathcal{Q}_2$ with

$$q := \alpha(\boldsymbol{s}) = \langle (1, (1, \bot)), (2, (\bot, 2)), (3, (3, 4)), (4, _) \rangle.$$

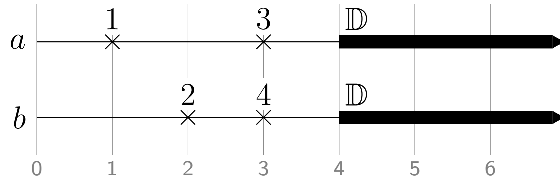
Following a similar style than the monitoring stream diagrams we can visualise such a stream as follows:



We further get for the concretisation

$$\gamma(q) = \boldsymbol{s}|_{<4} = (a|_{<4}, b|_{<4}) = (\{(1, 1), (3, 3)\}|_{<4}, \{(2, 2), (3, 4)\}|_{<4}),$$

which is depicted in the following diagram:



⌟

In the similar way how streams $s \in \mathcal{S}_{\mathbb{D}}$ can be seen as functions $f_s \colon \mathbb{T} \to \mathbb{D}_\bot$ (see Definition 3.21 in Section 3.2.1) we can see synchronised streams as functions, too:

Definition 4.9 (Functional View of Synchronised Streams)**.** Let $q \in \mathcal{Q}_k$ be a synchronised stream with progress t. Then $f_q \colon \mathbb{T} \to \mathbb{D}_\bot^k \cup \{?\}$ is the *functional view* of q. For any timestamp $t' \in \mathbb{T}$ we have

$$f_q(t') = \begin{cases} d & \text{if } q \text{ contains } (t', d) \text{ with } d \in \mathbb{D}_\bot^k, \\ \bot & \text{if } t' \notin T(\gamma(q)) \text{ and } t' < t \text{ if } t \text{ exclusive or } t' \leq t \text{ if } t \text{ inclusive}, \\ ? & \text{otherwise.} \end{cases}$$

⌟

The function maps a timestamp t to a k-tuple of values $d \in \mathbb{D}_\perp^k$ if any of the streams encoded in q has an event at time t. All timestamps t without any event at t are mapped to $\perp$ if t is lower than the progress of q. All timestamps after the progress of the stream are mapped to the new symbol ?.

If the usage is clear from the context, we use q to refer to f_q.

Using this functional view, we can now define a prefix relation on synchronised streams:

Definition 4.10 (Prefix Relation on Synchronised Streams)**.** Let $q, r \in \mathcal{Q}_k$ be two abstract monitoring streams. We define the *prefix relation* $\sqsubseteq \subseteq \mathcal{Q}_k \times \mathcal{Q}_k$ as follows:

$$q \sqsubseteq r :\Longleftrightarrow \forall\, t \in \mathbb{T}\colon q(t) \in \{r(t), ?\}.$$

⌟

The prefix relation forms a partial order $(\mathcal{Q}_k, \sqsubseteq)$. If we use the inverse prefix relation on synchronised streams and the inverse refinement relation on monitoring streams, it follows directly from the definitions above:

Lemma 4.11 (Galois Connection for Synchronised Streams)**.** *The abstraction function $\alpha\colon \mathcal{P}_\mathbb{D}^k \to \mathcal{Q}_k$ and the concretisation function $\gamma\colon \mathcal{Q}_k \to \mathcal{P}_\mathbb{D}^k$ are a Galois connection between $(\mathcal{P}_\mathbb{D}^k, \sqsupseteq)$ and $(\mathcal{Q}_k, \sqsupseteq)$.*

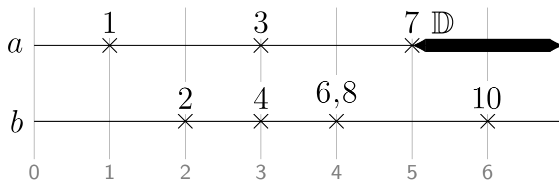
The following example illustrates the Galois connection:

Example 4.12 (Galois Connection Between Monitoring Streams and Synchronised Streams)**.** The functions α and γ are a Galois connection between the partial orders $(\mathcal{P}_\mathbb{D}^k, \sqsupseteq)$ and $(\mathcal{Q}_k, \sqsupseteq)$, i. e. we have

$$\forall\, \boldsymbol{a} \in \mathcal{P}_\mathbb{D}^k, b \in \mathcal{Q}_k\colon \alpha(\boldsymbol{a}) \sqsupseteq b \Longleftrightarrow \boldsymbol{a} \sqsupseteq \gamma(b).$$

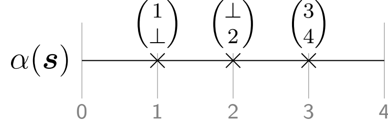
We illustrate this relation for $k = 2$. Recall the tuple $\boldsymbol{s} \in \mathcal{P}_\mathbb{D}^2$ of monitoring streams from Example 4.8:

$$\begin{aligned}\boldsymbol{s} = (a, b) = (&\{\langle(1,1),(3,3),(5,7)\rangle\}|_{\leq 5},\\ &\{\langle(2,2),(3,4),(4,6),(6,10)\rangle,\\ &\;\langle(2,2),(3,4),(4,8),(6,10)\rangle\})\end{aligned}$$



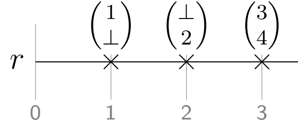
The minimal progress of $\boldsymbol{s}$ is an exclusive progress of 4 and thus we get the following synchronised stream $\alpha(\boldsymbol{s}) \in \mathcal{Q}_2$:

$$\alpha(\boldsymbol{s}) = \langle(1,(1,\bot)),(2,(\bot,2)),(3,(3,4)),(4,_)\rangle.$$



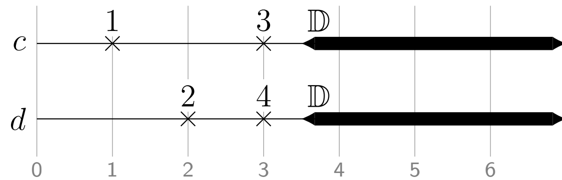
We now choose another synchronised stream $r \in \mathcal{Q}_2$ which is a prefix of $\alpha(\boldsymbol{s})$, i. e. $\alpha(\boldsymbol{s}) \sqsupseteq r$:

$$r = \langle(1,(1,\bot)),(2,(\bot,2)),(3,(3,4)),(3.5,\bot)\rangle.$$



The synchronised stream r has an inclusive progress of 3.5 and thus we get the following tuple of monitoring streams $\gamma(r) \in \mathcal{P}_{\mathbb{D}}^2$:

$$\gamma(r) = (c,d) = (\{(1,1),(3,3)\}|_{\leq 3.5}, \{(2,2),(3,4)\}|_{\leq 3.5}))$$



We can see that $\gamma(r)$ is, in fact, a prefix of $\boldsymbol{s}$, i. e. $\boldsymbol{s} \sqsupseteq \gamma(r)$. ⌟

4.1.3. Operator Functions

In this section, formal semantics for the synchronous evaluation of the basic TeSSLa operators on $\mathcal{Q}_n$ is given in the form of operator functions. The operator functions work under the assumption that they are evaluated in the correct order: Every operator is provided with the current values of its dependencies. Further, every operator is equipped with a memory cell to store data locally. After defining the operator functions below, we will describe how an entire TeSSLa specification is translated into a synchronous monitoring function that evaluates the flow graph synchronously for a global current timestamp. (See Definition 4.20 below.)

An operator function $o\colon \mathbb{T} \times \mathbb{D}_\perp \times \mathbb{D}_\perp^n \to \mathbb{D}_\perp \times \mathbb{D}_\perp \times \mathbb{T}_\perp$ with $o(t, m, \boldsymbol{v}) = (m', o, c)$ takes

- a current timestamp $t \in \mathbb{T}$,
- a current value $m \in \mathbb{D}_\perp$ for the memory cell and
- an n-tuple $\boldsymbol{v} \in \mathbb{D}_\perp^n$ of current values of its input streams at time t.

It produces

- a new value $m' \in \mathbb{D}_\perp$ for the memory cell,
- a value $o \in \mathbb{D}_\perp$ for its output stream at timestamp t and
- a timestamp $c \in \mathbb{T}_\perp$ for the next evaluation.

Definition 4.13 (Semantics of the Synchronous Operator $\mathbf{unit}^\mathsf{s}$)**.** The synchronous operator $\mathbf{unit}^\mathsf{s}\colon \mathbb{T} \times \mathbb{D}_\perp \times \mathbb{D}_\perp^0 \to \mathbb{D}_\perp \times \mathbb{D}_\perp \times \mathbb{T}_\perp$ is given by

$$\mathbf{unit}^\mathsf{s}(t, m) = (\perp, o, \perp)$$

with

$$o = \begin{cases} \square & \text{if } t = 0, \\ \perp & \text{otherwise.} \end{cases}$$

⌟

The $\mathbf{unit}^\mathsf{s}$ operator produces an event at timestamp 0 and ignores all inputs. For this operator to work, it is necessary to evaluate all flow graphs at timestamp 0 even if the input does not contain any event at that timestamp. Note that the definition of timestamp-conservative functions in Definition 3.95 in Section 3.6 explicitly allows the generation of additional events at the timestamp 0.

Definition 4.14 (Semantics of the Synchronous Operator $\mathbf{time}^\mathsf{s}$)**.** The synchronous operator $\mathbf{time}^\mathsf{s}\colon \mathbb{T} \times \mathbb{D}_\perp \times \mathbb{D}_\perp^1 \to \mathbb{D}_\perp \times \mathbb{D}_\perp \times \mathbb{T}_\perp$ is given by

$$\mathbf{time}^\mathsf{s}(t, m, v) = (\perp, o, \perp)$$

with

$$o = \begin{cases} t & \text{if } v \neq \perp, \\ \perp & \text{otherwise.} \end{cases}$$

⌟

For every incoming event, the $\mathbf{time}^\mathsf{s}$ operator sends out an event with the current timestamp used as the value.

Definition 4.15 (Semantics of the Synchronous Operator $\mathbf{lift}^{\mathsf{s}}$)**.** The synchronous operator $\mathbf{lift}^{\mathsf{s}} \colon (\mathbb{D}^n \rightarrowtail \mathbb{D}) \to (\mathbb{T} \times \mathbb{D}_\perp \times \mathbb{D}_\perp^n \to \mathbb{D}_\perp \times \mathbb{D}_\perp \times \mathbb{T}_\perp)$ is given by

$$\mathbf{lift}^{\mathsf{s}}(f)(t, m, \boldsymbol{v}) = (\perp, o, \perp)$$

with

$$o = f(\boldsymbol{v}).$$

The $\mathbf{lift}^{\mathsf{s}}$ operator lets the lifted function f handle the computation of the output value. There is no memory involved, so the function f is only called with the values of the input streams at the current timestamp. The input might be (a tuple of) $\perp$ if there is no event at that timestamp. The definition of the operator $\mathbf{lift}^{\mathsf{s}}$ defined in Definition 3.26 in Section 3.2.3 requires the lifted $\perp$-function $f \colon \mathbb{D}^n \rightarrowtail \mathbb{D}$ to be well-formed, i. e. it must return $\perp$ if all its arguments are $\perp$.

Definition 4.16 (Semantics of the Synchronous Operator $\mathbf{last}^{\mathsf{s}}$)**.** The synchronous operator $\mathbf{last}^{\mathsf{s}} \colon \mathbb{T} \times \mathbb{D}_\perp \times \mathbb{D}_\perp^2 \to \mathbb{D}_\perp \times \mathbb{D}_\perp \times \mathbb{T}_\perp$ is given by

$$\mathbf{last}^{\mathsf{s}}(t, m, v, r) = (m', o, \perp)$$

with

$$m' = \begin{cases} v & \text{if } v \neq \perp, \\ m & \text{otherwise.} \end{cases}$$

$$o = \begin{cases} m & \text{if } r \neq \perp, \\ \perp & \text{otherwise.} \end{cases}$$

The $\mathbf{last}^{\mathsf{s}}$ operator is the only operator in timestamp-conservative specifications that uses the memory cell. It stores the current value of the value stream in the memory cell and produces an event with the current value of the memory cell for every event on its second input. Note that the operator is only evaluated once per timestamp and that the output is based on the old memory value m and not the new memory value m'. This evaluation reflects how the $\mathbf{last}^{\mathsf{s}}$ operator defined in Definition 3.27 in Section 3.2.3 always refers to the last known value but not the current value.

Definition 4.17 (Semantics of the Synchronous Operator $\mathbf{delay}^{\mathsf{s}}$)**.** The synchronous operator $\mathbf{delay}^{\mathsf{s}} \colon \mathbb{T} \times \mathbb{D}_\perp \times \mathbb{D}_\perp^1 \to \mathbb{D}_\perp \times \mathbb{D}_\perp \times \mathbb{T}_\perp$ is given by

$$\mathbf{delay}^{\mathsf{s}}(t, m, v) = (m', o, m')$$

with

$$m' = \begin{cases} t + v & \text{if } v \neq \bot \wedge v > 0, \\ m & \text{if } v = \bot \wedge m \neq \bot \wedge m > t, \\ \bot & \text{otherwise.} \end{cases}$$

$$o = \begin{cases} \square & \text{if } m = t, \\ \bot & \text{otherwise.} \end{cases}$$

⌟

The **delay**$^{\mathsf{s}}$ operator can create events at arbitrary timestamps after the given delay is over. It uses the memory cell to track when to produce an event on the output stream. A given delay of 0 resets the currently active delay. The **delay**$^{\mathsf{s}}$ is the only operator which returns an actual timestamp for the next evaluation, i. e. not $\bot$. The operator always returns (m', o, m'), which requests that the operators' next evaluation always happens at the timestamp stored in its memory cell. Note how the definition of m' precisely reflects the semantics of the operator given in Definition 3.28 in Section 3.2.3.

Note that returning m' as the timestamp for the next evaluation does not guarantee that the operator will not be evaluated earlier. The flow graph is evaluated synchronously for all relevant timestamps, and if other timestamps before m' are relevant for other operators, this operator will be evaluated earlier. Hence the operator needs to return m' because the synchronous monitoring function does not store these obligations separately. It computes the minimal next timestamp from all internal obligations returned by the operators and the next timestamp in the input stream after every step. (See next section.)

4.1.4. Synchronised Monitoring Function

Now we can combine the individual operator functions to a monitoring function. So far, we only described the individual operator functions assuming that we can provide them with all their dependencies at a particular timestamp. Next, we consider when and in which order the operator functions should be evaluated to fulfil these dependencies.

We start by taking all the operators in the flow graph of a specification and combining them in a joined operator function. This intermediate step combines all the operators in a big tuple of functions. The obtained function computes a single next timestamp as the minimum of all the next timestamps returned by the individual operators.

Definition 4.18 (Joined Operator Function)**.** Let φ be a TeSSLa specification with k free streams $\boldsymbol{y} = (y_1, y_2, \ldots, y_k)$ and n bound streams $\boldsymbol{z} = (z_1, z_2, \ldots, z_n)$ that is well-formed. The specification φ consists of n equations $z_i = o_i(\boldsymbol{y})(\boldsymbol{z})$ for $1 \leq i \leq n$ with o_i being a basic operator, i. e. the specification is flat.

For every operator o_i we take the corresponding operator function $p_i \colon \mathbb{T} \times \mathbb{D}_\perp \times (\mathbb{D}_\perp^k \times \mathbb{D}_\perp^n) \to \mathbb{D}_\perp \times \mathbb{D}_\perp \times \mathbb{T}_\perp$ defined above. The input signature of the operator function is extended to all free and bound streams of φ. The operator function with the extended signature ignores all the additional streams in its input.

The *joined operator function* $g \colon \mathbb{T} \times \mathbb{D}_\perp^n \times (\mathbb{D}_\perp^k \times \mathbb{D}_\perp^n) \to \mathbb{D}_\perp^n \times \mathbb{D}_\perp^n \times \mathbb{T}_\perp$ is derived by applying all p_i for $1 \leq i \leq n$ in parallel: Let $p_i(t, m_i, (\boldsymbol{y}, \boldsymbol{z})) = (m_i', o_i, c_i)$ for $1 \leq i \leq n$ be the individual operator functions, then $g(t, \boldsymbol{m}, (\boldsymbol{y}, \boldsymbol{z})) = (\boldsymbol{m}', \boldsymbol{z}', c)$ is given by the following rules:

- The same timestamp $t \in \mathbb{T}$ is passed to all p_i.
- The i-th entry in the memory input tuple $\boldsymbol{m} \in \mathbb{D}_\perp^n$ is passed to p_i.
- The input values of the free and bound streams $(\boldsymbol{y}, \boldsymbol{z}) \in (\mathbb{D}_\perp^k \times \mathbb{D}_\perp^n)$ are passed to all p_i because every p_i selects its needed inputs from the entire tuples.
- The updated memory tuple $\boldsymbol{m}' \in \mathbb{D}_\perp^n$ is derived by concatenating the updated memory values $m_i' \in \mathbb{D}_\perp$ of every p_i.
- The tuple of output values for the bound streams $\boldsymbol{z}' \in \mathbb{D}_\perp^n$ is derived by concatenating the outputs o_i of every p_i.
- The timestamp $c \in \mathbb{T}_\perp$ for the next evaluation is derived as minimum of all c_i with $c_i \neq \perp$ or $\perp$ if all c_i are $\perp$. ⌟

The joined operator function takes the current valuation of all the free and bound streams as input and returns the valuation of the bound streams at the same timestamp as output. We cannot use the function like that because we do not know the valuation of the bound streams. Hence, the next step is to close this function over the valuations of the bound streams. We leave the co-domain and the image of g as it is, but we remove the bound streams from its domain. From all the possible images for the thereby identified elements of the function's domain, we choose those where the bound streams are mapped to themselves. This step is essentially the equivalent of taking the fixed point in the definition of the monitoring semantics in Definition 3.67 in Section 3.5.2.

Definition 4.19 (Closed Operator Function)**.** Let $g \colon \mathbb{T} \times \mathbb{D}_\perp^n \times (\mathbb{D}_\perp^k \times \mathbb{D}_\perp^n) \to \mathbb{D}_\perp^n \times \mathbb{D}_\perp^n \times \mathbb{T}_\perp$ be a joined operator function with $g(t, \boldsymbol{m}, (\boldsymbol{y}, \boldsymbol{z})) = (\boldsymbol{m}', \boldsymbol{z}', c)$ for a TeSSLa specification φ as defined above. Then the *closed operator function* $e \colon \mathbb{T} \times \mathbb{D}_\perp^n \times \mathbb{D}_\perp^k \cup \{_\} \to \mathbb{D}_\perp^n \times \mathbb{D}_\perp^n \cup \{_\} \times \mathbb{T}_\perp$ is derived from g by closing over $\boldsymbol{z} = \boldsymbol{z}'$, i. e. taking

those elements from g where $\boldsymbol{z} = \boldsymbol{z}'$. The additional input _ is passed through for any memory $\boldsymbol{m} \in \mathbb{D}_\perp^n$ and any timestamp $t \in \mathbb{T}$ as follows:

$$e(t, \boldsymbol{m}, _) = (\boldsymbol{m}, _, \perp).$$ ⌟

The well-formedness of the TeSSLa specification φ guarantees that its closed operator function e is a well-defined function. The well-formedness of a TeSSLa specification requires that every cycle in its flow graph contains at least one **last** or **delay**. In the operator functions $\mathbf{last}^{\mathsf{s}}$ and $\mathbf{delay}^{\mathsf{s}}$ the output does not depend on the input, but only on the memory. Hence, there is only one closed operator function for a given joined operator function because every dependency cycle of the bound streams is broken up when considering only one timestamp.

The additional input _ represents inclusive progress up to but not including the given timestamp. It is only included in the closed operator function to make the definition of the synchronised monitoring function more convenient. With this addition, it does not need to handle this exclusive progress explicitly: For every given timestamp, all internally generated timestamps being lower to the given timestamp are processed before processing the given timestamp.

The closed operator function evaluates the flow graph for a particular timestamp. The synchronised monitoring function defines how to apply the closed operator function to a synchronised stream.

In the following, we assume that every synchronous monitoring stream $q \in \mathcal{Q}_k$ starts with the timestamp 0. This assumption can be easily met by prepending the stream with $(0, \perp)$ if needed.

Definition 4.20 (Synchronised Monitoring Function)**.** Let $e\colon \mathbb{T} \times \mathbb{D}_\perp^n \times \mathbb{D}_\perp^k \to \mathbb{D}_\perp^n \times \mathbb{D}_\perp^n \times \mathbb{T}_\perp$ be a closed operator function with $e(t, \boldsymbol{m}, \boldsymbol{y}) = (\boldsymbol{m}', \boldsymbol{z}', c)$ for a TeSSLa specification φ as defined above. Then the *synchronised monitoring function* $f\colon \mathcal{Q}_k \to \mathcal{Q}_n$ is given by $f(q) = p$ as follows:

Let $q := (t_0, \boldsymbol{q}_0)(t_1, \boldsymbol{q}_1)\ldots$ be the synchronised input stream, $p := (\tau_0, \boldsymbol{p}_0)(\tau_1, \boldsymbol{p}_1)\ldots$ be the output stream, $y := \boldsymbol{y}_0\boldsymbol{y}_1\ldots$ be the sequence of input events, $m := \boldsymbol{m}_0\boldsymbol{m}_1\ldots$ be the sequence of memories and $c := c_0c_1\ldots$ the sequence of next timestamps with the following conditions:

- We start with the initial timestamp $\tau_0 = 0$, the initial memory $\boldsymbol{m}_0 = \perp$ and the initial input $\boldsymbol{y}_0 = \boldsymbol{q}_0$.
- We continue for all $j \geq 0$ with

$$(\boldsymbol{m}_{j+1}, \boldsymbol{p}_j, c_j) = e(\tau_j, \boldsymbol{m}_j, \boldsymbol{y}_j).$$

- With $i > 0$ chosen such that $t_{i-1} \leq \tau_{j-1} < t_i$ we inductively define the next timestamp τ_j and its input $\boldsymbol{y}_j$ for all $j > 0$ as follows: If $c_{j-1} \neq \bot$ and $c_{j-1} \leq t_i$ we define

$$\tau_j = c_{j-1} \quad \text{and} \quad \boldsymbol{y}_j = \bot$$

and otherwise

$$\tau_j = t_i \quad \text{and} \quad \boldsymbol{y}_j = \boldsymbol{q}_i.$$

⌟

The synchronised monitoring function applies the closed operator function to every tuple of input events. The closed operator function provides an updated memory tuple $\boldsymbol{m}_{j+1}$, an output tuple $\boldsymbol{p}_j$ and a potentially generated next timestamp c_j. If that generated timestamp is smaller than the next input timestamp, then the closed operator function is evaluated on that timestamp before processing the next input event.

4.1.5. Examples

We now discuss examples of synchronised monitoring functions for several specifications. We start with a simple timestamp-conservative TeSSLa specification without any recursion:

Example 4.21 (Simple Example of a Synchronised Monitoring Function)**.** Consider the TeSSLa specification $y = \mathsf{merge}(a, b)$ with the free (input) streams $a, b \in \mathcal{P}_{\mathbb{D}}$ and the bound (output) stream $y \in \mathcal{P}_{\mathbb{D}}$. The derived TeSSLa operator merge is defined in Definition 3.37 in Section 3.3.1 for the binary case as $\mathbf{lift}(h)$ with the function $h\colon \mathbb{D}_\bot^2 \to \mathbb{D}_\bot$ given by

$$h(a, b) = \begin{cases} a & \text{if } a \neq \bot, \\ b & \text{otherwise.} \end{cases}$$

We get the operator function $p = \mathbf{lift}^{\mathsf{s}}(h)$ with the signature $p\colon \mathbb{T} \times \mathbb{D}_\bot \times \mathbb{D}_\bot^2 \to \mathbb{D}_\bot \times \mathbb{D}_\bot \times \mathbb{T}_\bot$ and the definition

$$p(t, m, (a, b)) = (\bot, h(a, b), \bot).$$

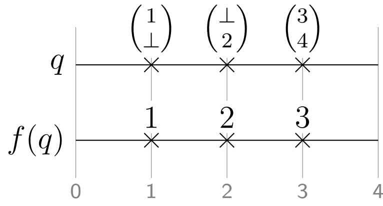
Since there is only one operator function involved, the joined operator function $g\colon \mathbb{T} \times \mathbb{D}_\bot \times (\mathbb{D}_\bot \times \mathbb{D}_\bot^2) \to \mathbb{D}_\bot \times \mathbb{D}_\bot \times \mathbb{T}_\bot$ looks similar to the operator function p:

$$g(t, m, y, (a, b)) = (\bot, h(a, b), \bot)$$

Note that the output of g is independent of the current timestamp t, the memory value m and the bound (output) stream y. Hence we can derive the closed operator function $e\colon \mathbb{T} \times \mathbb{D}_\perp \times \mathbb{D}_\perp^2 \to \mathbb{D}_\perp \times \mathbb{D}_\perp \times \mathbb{T}_\perp$ simply by removing the additional input parameter y again:

$$e(t, m, (a, b)) = (\perp, h(a, b), \perp)$$

The monitor function $f\colon \mathcal{Q}_2 \to \mathcal{Q}_1$ is derived by applying e to all timestamped 2-tuples of the input stream. In this case, this boils down to applying h to all tuples of the input stream. The application of f to the input stream $q \in \mathcal{Q}_2$ from Example 4.8 (Abstraction and Concretisation for Synchronised Streams) in Section 4.1.2 can be visualised as follows:



⌟

In the following example, we consider a recursive TeSSLa specification, i. e. one that contains a cycle in the flow graph. For such a specification, the closed operator function now actually needs to close over the valuation of the bound streams because those are used in the joined operator function.

Example 4.22 (Closed Operator Function for a Recursive TeSSLa Specification)**.** Consider the following TeSSLa specification with the free (input) stream $x \in \mathcal{P}_\mathbb{D}$ and the bound (output) streams $y, \ell, a \in \mathcal{P}_\mathbb{D}$:

$$\begin{aligned} y &= \mathsf{default}(\ell, 0) \\ \ell &= \mathbf{lift}(+)(a, x) \\ a &= \mathbf{last}(y, x) \end{aligned}$$

We assume an operator function $\mathsf{default}^\mathsf{s}$ being derived by combining $\mathsf{merge}^\mathsf{s}$ and $\mathsf{const}^\mathsf{s}$ appropriately. For the joined operator function $g\colon \mathbb{T} \times \mathbb{D}_\perp \times (\mathbb{D}_\perp \times \mathbb{D}_\perp^3) \to \mathbb{D}_\perp \times \mathbb{D}_\perp^3$ we ignore the memory update of the operator functions if it is constantly $\perp$. Further, the timestamp for the next evaluation is ignored because it is always $\perp$. After all, no **delay** is used in the specification:

$$g(t, m, x, (y, \ell, a)) = (m', (y', \ell', a'))$$

with

$$y' = \begin{cases} 0 & \text{if } t = 0 \wedge \ell = \perp, \\ \ell & \text{otherwise.} \end{cases}$$

$$\ell' = \begin{cases} a + x & \text{if } a \neq \bot, \\ \bot & \text{otherwise.} \end{cases}$$

$$a' = \begin{cases} m & \text{if } x \neq \bot, \\ \bot & \text{otherwise.} \end{cases}$$

$$m' = \begin{cases} y & \text{if } y \neq \bot, \\ m & \text{otherwise.} \end{cases}$$

To derive the closed operator operator function $e \colon \mathbb{T} \times \mathbb{D}_\bot \times \mathbb{D}_\bot \to \mathbb{D}_\bot \times \mathbb{D}_\bot^3$ with

$$g(t, m, x) = (m', (y', \ell', a'))$$

we take the definition of g and replace all occurrences of y, ℓ and a with y', ℓ' and a', respectively. Because of the well-formedness of φ this definition does not contain any cyclic dependencies and we end up with a well-defined function. To double-check this we can reformulate all definitions directly in terms of m and x:

$$a' = \begin{cases} m & \text{if } x \neq \bot, \\ \bot & \text{otherwise.} \end{cases}$$

$$\ell' = \begin{cases} m + x & \text{if } x \neq \bot \wedge m \neq \bot, \\ \bot & \text{otherwise.} \end{cases}$$

$$y' = \begin{cases} 0 & \text{if } t = 0 \wedge (x = \bot \vee m = \bot), \\ m + x & \text{if } x \neq \bot \wedge m \neq \bot \\ \bot & \text{otherwise.} \end{cases}$$

$$m' = \begin{cases} 0 & \text{if } t = 0 \wedge (x = \bot \vee m = \bot), \\ m + x & \text{if } x \neq \bot \wedge m \neq \bot \\ m & \text{otherwise.} \end{cases}$$

We can simplify further using the fact that $t = 0$ implies $m = \bot$:

$$y' = \begin{cases} 0 & \text{if } t = 0, \\ m + x & \text{if } x \neq \bot \wedge t > 0, \\ \bot & \text{otherwise.} \end{cases}$$

$$m' = \begin{cases} 0 & \text{if } t = 0, \\ m + x & \text{if } x \neq \bot \wedge t > 0, \\ m & \text{otherwise.} \end{cases}$$

Now we can see how this function sums up all the values of x into the memory. It outputs the stored value with every event on x, starting with a default value of 0 at

timestamp 0. The cases $y' = \bot$ and $m' = m$ are relevant if the function is evaluated at timestamps without an event on the stream x. Then no output is generated, and the memory is left untouched. ⌟

Finally, we consider the specification from Example 3.80 (Variable Frequency Period) in Section 3.5.3 as an example of a specification which is recursive and not timestamp-conservative:

Example 4.23 (Not Timestamp-Conservative Synchronised Monitoring Function). Consider the following TeSSLa specification φ with the free (input) stream $x \in \mathcal{S}_{\mathbb{R}\geq 0}$ and the bound (derived) streams $d \in \mathcal{S}_{\mathbb{U}}$ and $z, \ell \in \mathcal{S}_{\mathbb{R}\geq 0}$ whose dependency graph was shown in Example 3.33 (Dependency Graph of a Well-Formed TeSSLa Specification) in Section 3.2.4:

$$
\begin{aligned}
z &= \mathsf{merge}(x, \ell) \\
\ell &= \mathbf{last}(x, d) \\
d &= \mathbf{delay}(z)
\end{aligned}
$$

We get the joined operator function $g \colon \mathbb{T} \times \mathbb{D}_\bot^3 \times (\mathbb{D}_\bot \times \mathbb{D}_\bot^3) \to \mathbb{D}_\bot^3 \times \mathbb{D}_\bot^3 \times \mathbb{T}_\bot$ with

$$g(t, (m_z, m_\ell, m_d), x, (z, \ell, d)) = ((m'_z, m'_\ell, m'_d), (z', \ell', d'), c)$$

given by

$$
\begin{aligned}
z' &= \begin{cases} x & \text{if } x \neq \bot, \\ \ell & \text{otherwise.} \end{cases} \\
m'_z &= \bot \\
\ell' &= \begin{cases} m_\ell & \text{if } d \neq \bot, \\ \bot & \text{otherwise.} \end{cases} \\
m'_\ell &= \begin{cases} m_\ell & \text{if } x = \bot, \\ x & \text{otherwise.} \end{cases} \\
d' &= \begin{cases} \Box & \text{if } t = m_d, \\ \bot & \text{otherwise.} \end{cases} \\
c = m'_d &= \begin{cases} t + z & \text{if } z \neq \bot \wedge z > 0, \\ m_d & \text{if } z = \bot \wedge m_d \neq \bot \wedge m_d > t, \\ \bot & \text{otherwise.} \end{cases}
\end{aligned}
$$

The closed operator function $e \colon \mathbb{T} \times \mathbb{D}_\bot^3 \times \mathbb{D}_\bot \to \mathbb{D}_\bot^3 \times \mathbb{D}_\bot^3 \times \mathbb{T}_\bot$ with

$$e(t, (m_z, m_\ell, m_d), x) = ((m'_z, m'_\ell, m'_d), (z', \ell', d'), c)$$

can be derived by replacing z, ℓ and d with z', ℓ' and d', respectively, in the above equations. Because of the well-formedness of φ we do not end up with cyclic dependencies and we get a well-defined function e. To get a clearer picture of what e does, we can rewrite some of the equations in terms of the arguments of e:

$$z' = \begin{cases} x & \text{if } x \neq \bot, \\ m_\ell & \text{if } x = \bot \wedge t = m_d, \\ \bot & \text{otherwise.} \end{cases}$$

$$m'_\ell = \begin{cases} m_\ell & \text{if } x = \bot, \\ x & \text{otherwise.} \end{cases}$$

$$m'_d = \begin{cases} t + x & \text{if } x \neq \bot \wedge x > 0, \\ t + m_\ell & \text{if } x = \bot \wedge t = m_d \wedge m_\ell > 0, \\ m_d & \text{if } x = \bot \wedge m_d \neq \bot \wedge m_d > t, \\ \bot & \text{otherwise.} \end{cases}$$

Now we can see how this function realises the specification

$$z = \mathsf{merge}(x, \mathbf{last}(x, \mathbf{delay}(z))).$$

Every event on the stream x is directly passed through to z. If x does not contain an event and the current timestamp matches the target timestamp of the delay stored in m_d, then the last value seen on x is outputted as an event on z. The last value seen on x is stored in m_ℓ, updated with every new value seen on x. The target timestamp m_d is computed either

- as $t + x$ if there is an event on x or
- as $t + m_\ell$ if the target timestamp is reached and hence a new target timestamp needs to be computed.

The target timestamp is set to $\bot$ if x is negative or 0. In any other case, the old target timestamp is returned as the next one and thus left unchanged. It is necessary to leave the target timestamp unchanged if the closed operator function is evaluated with timestamps irrelevant for this specification, e. g. if the period is combined with other specifications whose events are not synchronous to the period.

Figure 4.2 shows an evaluation of the synchronous monitoring function derived from this closed monitoring function.

The synchronised input stream $q \in \mathcal{Q}_1$ is the synchronised version of the monitoring stream x, which sets the period's frequency.

The sequence $y \in (\mathbb{D}_\bot)^\infty$ contains the input events: It contains all events from q without timestamps and $\bot$ at positions without input events. The closed operator

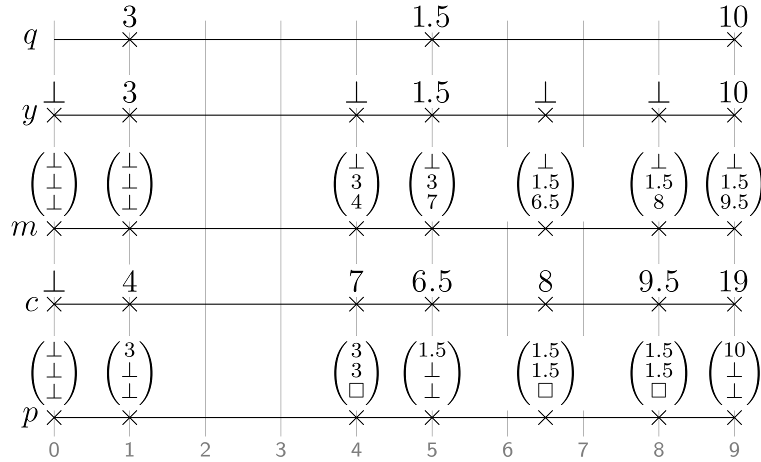


Figure 4.2.: Evaluation of the synchronous monitoring function for a specification which generates a period with a frequency depending on the input stream x.

function is evaluated at these positions because of an expired delay. Note that the sequence y is not a stream and does not contain timestamps. In the figure, it is aligned to the events of p'.

The sequence $m \in (\mathbb{D}_\perp^3)^\infty$ contains the memory values. The tuples are drawn as stacked matrix and contain the elements in order m_z, m_ℓ, m_d, i. e. the memory of the operator functions $\mathsf{merge}^\mathsf{s}$, $\mathbf{last}^\mathsf{s}$ and $\mathbf{delay}^\mathsf{s}$. Note that $\mathsf{merge}^\mathsf{s}$ does not actually need any memory, and hence the first entry in this tuple is always $\perp$. The sequence is aligned with the streams in the diagram such that the closed operator function's input memory value is aligned with the timestamp. Consequently, the updated memory value can be found as input to the next timestamp.

The sequence $c \in (\mathbb{T}_\perp)^\infty$ consists of the next timestamps. It is aligned with the streams in the diagram such that the closed operator function's output is aligned to the timestamp.

Finally, the synchronised stream $p \in \mathcal{Q}_3$ is the output stream. The entries of its tuples are in the same order as the tuples of the memory values, i. e. their values represent the outputs of the operators $\mathsf{merge}^\mathsf{s}$, $\mathbf{last}^\mathsf{s}$ and $\mathbf{delay}^\mathsf{s}$ in that order. ⌟

4.1.6. Correctness and Properties

The following theorem states that the synchronised monitoring function adheres to the monitoring semantics. As discussed in the introduction, it does not provide

the same output, but it always provides a prefix of the monitoring semantics' output. More precisely, up to the input streams' minimal progress, the synchronous monitoring function provides the same output as the monitoring semantics.

Note that the largest timestamp of $\alpha(\boldsymbol{s})$ for a tuple of monitoring streams $\boldsymbol{s} \in \mathcal{P}_{\mathbb{D}}^k$ is the largest timestamp in $T(\boldsymbol{s})$ being lower (or equal to) than the minimal exclusive (or inclusive) progress of $\boldsymbol{s}$.

Theorem 4.24 (Correctness of Synchronised Monitoring)**.** *Let φ be a well-formed TeSSLa specification and $f\colon \mathcal{Q}_k \to \mathcal{Q}_n$ the corresponding synchronised monitoring function and $\hat{\boldsymbol{f}}_\varphi\colon \mathcal{P}_{\mathbb{D}}^k \to \mathcal{P}_{\mathbb{D}}^n$ the monitoring semantics of φ. Let $\boldsymbol{s} \in \mathcal{P}_{\mathbb{D}}^k$ be any tuple of input streams, and t its minimal progress. We than have*

$$\gamma(f(\alpha(\boldsymbol{s}))) = \begin{cases} \hat{\boldsymbol{f}}_\varphi(\boldsymbol{s})|_{\leq t} & \textit{if } t \textit{ is inclusive,} \\ \hat{\boldsymbol{f}}_\varphi(\boldsymbol{s})|_{<t} & \textit{otherwise.} \end{cases}$$

Proof. The operator functions are defined consistent with the operator's semantics. The closed operator function realises precisely the fixed-point operation used in the definition of $\hat{\boldsymbol{f}}_\varphi$. The closed operator function only works on individual values and not streams, so it only performs the calculation for a single global timestamp. The well-formedness of φ guarantees that the closed operator function is well defined, i. e. it can be evaluated to a unique solution for every timestamp. The synchronised monitoring function executes the closed operator function repeatedly for all relevant timestamps of the input streams. The closed operator function must be called often enough, which is natural for timestamp-conservative specifications and explicitly handled for **delay**, which is the only operator which can create events with additional timestamps. The synchronisation ensures that every operator function has all the required inputs ready when called and the future independence of the specification guarantees that information about previous events is sufficient. The synchronised monitoring function executes the closed operator function up to the last input event and hence generates output precisely up to the minimal progress t of $\boldsymbol{s}$. □

By rewriting the above correctness statement, we get the weaker statement for any synchronised stream $q \in \mathcal{Q}_k$

$$\gamma(f(q)) \sqsubseteq \hat{\boldsymbol{f}}_\varphi(\gamma(q)).$$

With this relation and Lemma 4.11 (Galois Connection for Synchronised Streams) from Section 4.1.2 we can conclude:

Corollary 4.25 (Synchronised Monitoring is an Abstraction of the Monitoring Semantics)**.** *Let φ be a TeSSLa specification and $f\colon \mathcal{Q}_k \to \mathcal{Q}_n$ the corresponding synchronised monitoring function and $\hat{\boldsymbol{f}}_\varphi\colon \mathcal{P}_{\mathbb{D}}^k \to \mathcal{P}_{\mathbb{D}}^n$ the monitoring semantics of φ. Then f is an abstraction of $\hat{\boldsymbol{f}}_\varphi$.*

If we relax the statement of the above theorem differently, we get

$$\gamma(f(\alpha(\boldsymbol{s}))) \sqsubseteq \hat{\boldsymbol{f}}_\varphi(\boldsymbol{s}).$$

So we can follow from the above theorem this statement on the relation of the synchronised monitoring function and the monitoring semantics, too:

Corollary 4.26 (Synchronised Monitoring is Behavioural Equivalent to the Monitoring Semantics)**.** *Let φ be a TeSSLa specification and $f\colon \mathcal{Q}_k \to \mathcal{Q}_n$ the corresponding synchronised monitoring function and $\hat{\boldsymbol{f}}_\varphi\colon \mathcal{P}_\mathbb{D}^k \to \mathcal{P}_\mathbb{D}^n$ the monitoring semantics of φ. Then $\gamma \circ f \circ \alpha$ is behavioural equivalent to $\hat{\boldsymbol{f}}_\varphi$.*

We can derive further insights on the quality of the abstraction from the theorem: The output of the synchronised monitoring function (in combination with the abstraction function α and the concretisation function γ) is not only a prefix of the monitoring semantics, but this prefix is equal to the output of the monitoring semantics up to the minimal progress of the input streams. Due to the synchronous nature of the synchronised monitoring function, its output's progress is synchronised across all the monitoring streams contained in the output tuple. Further, its output's (minimal) progress is always its inputs minimal progress. So the synchronised monitoring function provides the least progress and the least refinement on the output, which is possible while still being future independent.

From Lemma 4.2 (Future Independence Preserves Progress) in Section 4.1.1 we learned that a future-independent function must at least preserve the minimal progress. So, in conclusion, the synchronised monitoring function can be seen as the simplest non-trivial abstraction which generates minimal progress while still being future independent.

4.2. Implementation Concepts

In order to implement the synchronised monitoring function defined in the last section, we need to realise the individual operator functions, the joined operator function, the closed operator function and finally, the synchronised monitoring function itself. The main challenge is realising the closed operator function because this requires executing the individual operator functions in the proper order. In the examples at the end of the previous section, we already used a linearisation of the operator functions to specify the synchronised monitoring function for a concrete specification.

In order to compute all dependencies before their usage, we sort the nodes in the given specification's flow graph topologically. The sorting can be done either by explicitly performing a linearisation or dynamically via message passing. The message

passing approach requires fewer steps for the static compilation. It is used for the interpreter, making its implementation as simple as possible. On the other hand, static linearisation requires a static compilation step. However, it benefits from the static compilation in terms of performance.

The main difference between the synchronised monitor function and its implementation is memory management: In the formal definition of the synchronised monitor function, the closed operator function gets a tuple of memory values as input and produces an updated tuple of memory values. The synchronised monitor function then keeps track of these memories. In the implementation, every individual operator can keep its memory in a local field. Since every operator only manipulates its own memory, this is a simple and efficient implementation.

Before getting into the actual implementations of the interpreter and the software compiler in the next two sections, we discuss two considerations common for both approaches:

- The imperative algorithm for the synchronised monitoring function, i. e. the outer loop of the monitor implementation, and
- the linearisation of the closed operator function, i. e. how to execute a single step inside the monitoring loop.

4.2.1. Imperative Algorithm for the Synchronised Monitoring Function

Definition 4.20 (Synchronised Monitoring Function) in Section 4.1.4 can be seen as a denotational description. The following definition describes the same synchronised monitoring function more operationally:

Definition 4.27 (Imperative Algorithm for the Synchronised Monitoring Function)**.** Let $e\colon \mathbb{T} \times \mathbb{D}_\perp^n \times \mathbb{D}_\perp^k \to \mathbb{D}_\perp^n \times \mathbb{D}_\perp^n \times \mathbb{T}_\perp$ be a closed operator function for a well-formed TeSSLa specification φ as defined above. Then the *imperative algorithm for the synchronised monitoring function* is given as follows:

We assume the following two imperative procedures with side effects:

- The procedure $\mathit{read}() = (t, d)$ takes no arguments and returns a timestamp $t \in \mathbb{T}$ and a data value $d \in \mathbb{D}_\perp^k$. The tuple (t, d) are the timestamp and value of the next event from the synchronised input stream. If there is no next event in the input stream, t becomes $\perp$.

- The procedure $\mathit{write}(t, d)$ takes a timestamp $t \in \mathbb{T}$ and a data value $d \in \mathbb{D}_\bot^n$ and returns nothing. It appends an event with the given timestamp and data value to the synchronised output stream.

The setup initialises the variables for the timestamps and the inputs and reads the initial input:

$\boldsymbol{m} := \bot$
$(t, \boldsymbol{y}) := \mathit{read}()$
$(t', \boldsymbol{y}') := \mathit{read}()$

The main loop runs as long as input is available:

while $t \neq \bot$ **do**
  $(\boldsymbol{m}, \boldsymbol{z}, c) := e(t, \boldsymbol{m}, \boldsymbol{y})$
  $\mathit{write}(t, \boldsymbol{z})$
  if $c \neq \bot \wedge t' \neq \bot \wedge c < t'$ **then**
    $(t, \boldsymbol{y}) := (c, \bot)$
  else
    $(t, \boldsymbol{y}) := (t', \boldsymbol{y}')$
    $(t', \boldsymbol{y}') := \mathit{read}()$
  end
end

Note that the current timestamp t and the next timestamp t', as well as the current input $\boldsymbol{y}$ and the next input $\boldsymbol{y}'$ and the memory $\boldsymbol{m}$, are mutable variables in the global scope of this routine. The output $\boldsymbol{z}$ and the next requested timestamp c are local variables only used inside the loop body.

The following result follows directly from the definition of the synchronised monitoring function and the above definition of its imperative description:

Lemma 4.28 (Correctness of the Imperative Algorithm for the Synchronised Monitoring Function)**.** *Let φ be a well-formed TeSSLa specification with k free and n bound streams. Further, let $f \colon \mathcal{Q}_k \to \mathcal{Q}_n$ be its synchronised monitoring function and let $p \in \mathcal{Q}_k$ and $q \in \mathcal{Q}_n$ be two synchronised monitoring streams. The imperative algorithm for the synchronised monitoring function applied on p such that we get q realises $f(p) = q$.*

Note that the above lemma only states that the imperative description is equivalent to the synchronised monitoring function defined in Definition 4.20 in Section 4.1.4. However, with the correctness of the synchronised monitoring from Theorem 4.24, the correctness of the imperative description follows.

The imperative algorithm explicitly calls the closed operator function e. We will discuss its implementation in the following subsection.

As already discussed in this chapter's introduction, there are two main differences between the synchronised monitoring function for TeSSLa shown above and the established common synchronous execution scheme shown in Figure 4.1:

- TeSSLa has explicit timestamps. The call to $\mathit{read}()$ provides the next event's data $\boldsymbol{y}'$ from the synchronized input stream together with its timestamp t'. The current timestamp t is passed to the closed operator function e with every call.
- TeSSLa can generate additional events with arbitrary timestamps. The closed operator function e returns an updated memory tuple $\boldsymbol{m}$, an output $\boldsymbol{z}$, and a timestamp c. In the case of timestamp-conservative specifications, c is always $\bot$, and the loop calls the closed operator function for every input event. However, in the general case, the variable c can contain a timestamp that indicates when the functions must be executed next. The conditional statements after the call to $\mathit{write}(t, \boldsymbol{z})$ determine the next current timestamp t and the corresponding input $\boldsymbol{y}$ for the next iteration of the loop: The current timestamp t is either updated to the next timestamp t' of the synchronised input stream or to the timestamp c, depending on which is smaller. In the latter case, the current input $\boldsymbol{y}$ is set to $\bot$ because the synchronised input stream contains no event at c.

The concept of a next timestamp c is loosely related to RTLola's ability to determine a step rate for a specification as presented in [BFS+20]. However, RTLola's step rate is fixed for an entire specification, and by calculating the c as shown above, we are dynamically adjusting the step rate based on the input.

As discussed in Section 3.4.4 TeSSLa is able to generate Zeno streams. The synchronised monitoring function can do so as well: There is no limit on how many intermediate timestamps can be added before the next timestamp from the input is considered. As long as the minimal next timestamp is always smaller than the next timestamp from the input, we will never consider the input again. This situation only occurs if the generated timestamps converge towards a limit but never reach the limit – which is precisely the pattern for Zeno behaviour in streams.

4.2.2. Implementing the Closed Operator Function

The imperative algorithm for the synchronised monitoring function discussed above explicitly calls the closed operator function $e\colon \mathbb{T} \times \mathbb{D}_\bot^n \times \mathbb{D}_\bot^k \to \mathbb{D}_\bot^n \times \mathbb{D}_\bot^n \times \mathbb{T}_\bot$ of the TeSSLa specification φ. In the following, we discuss the two slightly different

approaches of the interpreter and the software compiler on implementing the closed operator function. The implementation details of these two approaches are given in the following sections.

The TeSSLa flow graph might contain cycles. Thus, there is no apparent order for the execution of the operator functions contained in the closed operator function. However, we can distinguish three steps of the computation:

1. Computation of the outputs. Every operator computes the value for its output stream at the current timestamp.
2. Computation of the updated memory values. The operators which use their memory cell compute a new value for their memory cell.
3. Computation of the timestamp for the next evaluation. The operator **delay**$^{\mathsf{s}}$ additionally computes a timestamp for the subsequent evaluation.

The operator functions are defined such that they do not need dependencies related to the delayed-labelled edges in the flow graph to compute their outputs for the current timestamps. These inputs are only required to compute the updated values for their memory cells and the timestamps for the subsequent evaluation, which in turn are both not used until the next evaluation. Thus, we can implement the closed operator function by splitting the computation into the three steps listed above.

For every operator, we distinguish its inputs as follows:

Definition 4.29 (Delayed and Immediate Inputs)**.** Let φ be a TeSSLa specification and o an operator in its flow graph. Every incoming edge of o in the flow graph is called *input* of o. The incoming delayed-labelled edges of o are called *delayed inputs*, the others are called *immediate inputs*. ⌟

For the computations of the outputs, the operators must still be evaluated in proper order. In the case of the interpreter, this scheduling is done dynamically via message passing at runtime:

Definition 4.30 (Message-Passing Implementation of the Closed Operator Function)**.** Let φ be a well-formed TeSSLa specification with the closed operator function e. The *message-passing implementation of* e works as follows: Every operator has a variable for every input. The inputs of the closed operator function are distributed into the input variables of all dependent operators according to the flow graph. For every operator whose immediate inputs are available, the operator's output is computed and distributed into the input variables of all dependent operators according to the flow graph. For every operator whose immediate and delayed inputs are available, the memory cells are updated, and the timestamps for the next evaluation are computed. ⌟

Note that the condition to compute the output of an operator is immediately satisfied in case of the operators $\mathbf{delay}^{\mathsf{s}}$ and $\mathbf{unit}^{\mathsf{s}}$: They do not have any immediate inputs, and thus, their output values for the current timestamp can be computed immediately based on their memory values. The operator $\mathbf{delay}^{\mathsf{s}}$ then needs its delay-labelled input in order to update its memory value, but the updated memory value is not needed until the evaluation for the next timestamp.

The following lemma states the correctness of the above construction:

Lemma 4.31 (Correctness of the Message-Passing Implementation of the Closed Operator Function)**.** *Let φ be a well-formed TeSSLa specification with the closed operator function e. Then the message-passing implementation of e is correct.*

Proof. All operator functions are defined such that the computations of the output values only relies on the old memory values and the immediate inputs. The flow graph of φ without the delayed-labelled edges is acyclic. The inputs of e are distributed to the operators, and consequently, the output of every operator is computed. The computation of the updated memory values only depends on the input values. Once every output is computed, all delayed inputs are available, too. □

In the case of the software compiler, the closed operator function is implemented with a static evaluation order of the operators. Again we split the assignments of the closed operator function into the computation of the stream values and the memory updates:

Definition 4.32 (Linearising Implementation of the Closed Operator Function)**.** Let φ be a well-formed TeSSLa specification with the closed operator function e. The *linearising implementation of e* is obtained as follows: Use the flow graph of φ without the delayed-labelled edges to perform a topological sort of the derived streams. Realise the computation of the values of the derived streams in the order given by the topological order. Next, translate the computation of the new memory values and the next timestamps for all operators in arbitrary order. ⌟

The following lemma states the correctness of the above construction:

Lemma 4.33 (Correctness of the Linearising Implementation of the Closed Operator Function)**.** *Let φ be a well-formed TeSSLa specification with the closed operator function e. Then the linearising implementation of e is correct.*

Proof. We extend the proof of Lemma 4.31 as follows: For every directed acyclic graph, a topological order exists. If executed in any topological order, operators are executed if all immediate inputs are available. The updated memory values and the

timestamps are not used before the next iteration and can thus be computed in any order. □

The following two sections discuss how the interpreter's message passing and the software compiler's linearisation are implemented.

4.3. Interpreter

The interpreter uses the approach from Definition 4.27 (Imperative Algorithm for the Synchronised Monitoring Function) in Section 4.2.1 and Definition 4.30 (Message-Passing Implementation of the Closed Operator Function) in Section 4.2.2. The input events are assumed to be provided in a synchronised fashion, i. e. we get the information included in one tuple of the synchronised monitoring stream.

Object-oriented programming languages provide natural implementations of message passing: Dynamic object structures communicating by exchanging messages is one of the main object-oriented principles established by Smalltalk. [GR83] Objects represent every node of the dependency graph and perform the corresponding individual operator function. These objects are connected using the common observer pattern [GHJV94], i. e. every node listens to notifications of the nodes it depends on directly.

4.3.1. Implementing the Closed Operator Function

Figure 4.3 shows the interpreter in a UML class diagram with simplified type signatures. Parameter names are omitted, and return types are only specified if the method returns something. The `Value` type must be an option type, e. g. `Option[Any]`.

A `Stream` has `Listeners` listening to propagations of the streams it depends on directly. During the initialisation, a `Stream` registers its listeners by calling `addListener` on each direct predecessor in the flow graph. A stream can call `propagate` with a value to send this value to all its listeners.

Bound streams of a TeSSLa are defined in terms of operators. Bound streams are realised as subclasses of `Stream`, which overrides `init` with the registration of listeners to its dependencies. The late initialisation using the method `init` is discussed further in Section 4.3.4.

A stream keeps track of its dependencies and propagates a value for the current timestamp to its listeners when it has received enough information from its dependencies.

A derived stream defined by a **lift** operator can propagate a new value when it has received a value (or $\perp$) from all its direct predecessors in the flow graph. Due to the synchronous nature of the interpreter, all streams are evaluated for every relevant timestamp.

The dynamic implicit computation of the topological order does work because the operands are implemented to be independent of the order in which they receive information. They wait until they get all the needed information and perform their action.

A stream defined by a **last** can propagate a new value as soon as it receives a trigger on its input r. It is not necessary to wait for the update of its value input because, as one can see in the operator's definition in Definition 4.16 in Section 4.1.3, the input v is only used to update the local memory.

The **delay** operator only has a delayed input: A stream defined by a **delay** can propagate a new value immediately for every new timestamp without waiting for any input because, as one can see in the operator's definition Definition 4.17 in Section 4.1.3, the output is only defined in terms of the memory and the timestamp.

Hence a stream defined by a **delay** operator is a `TriggeredStream` which has an additional method `step` which will be called explicitly by the `Specification` for every timestamp. The same holds for the **unit** operator, which does not depend on any other streams. It must be triggered explicitly, too.

An `InputStream` propagates its explicitly set value to its listeners with every new timestamp. Hence it is a `TriggeredStream`, too. Additionally, it has a public method to set its current value externally.

The closed operator function is realised by the message passing along the dependency graph. The `Specification` realises the computation of the following timestamp defined in the joined operator function and the synchronised monitoring function: Streams can request next timestamps via `setTrigger` and the next timestamp of the input is set via `setTime`.

4.3.2. Implementing the Synchronised Monitoring Function

For every timestamp of the input streams, the user

a) sets the timestamp of the specification via `setTime`, then
b) sets current values of the input streams for that timestamp and finally

c) executes the specification for that timestamp calling `step`.

For every call to `setTime`, the specification sets the current time to the next requested time and steps the specification until the externally set timestamp is smaller than the next requested timestamp. Because the input streams are not provided with new values, they propagate $\bot$ for those timestamps.

A call to `step` steps the specification for the timestamp set via `setTime`: The specification calls `step` on all its `TriggerStreams`, and the current timestamp is made available to all streams via `getTime`.

How a specification is built and executed is discussed further in Section 4.3.4.

Every operator function is implemented as a class extending the class `Stream` (or the class `TriggeredStream`) and overrides `init` (and `step`). When they are explicitly stepped or receive propagated values, they perform the following steps:

1. *Reading the global timestamp.* Every stream can read the current global timestamp by calling `getTime` on the specification.
2. *Receiving values from the dependent streams.* Streams receive values from other streams they depend on because they listen to their propagations.
3. *Receiving a call to step.* Triggered streams are explicitly triggered for every new timestamp.
4. *Propagating the computed value.* Streams manually keep track of the information they receive from their predecessors in the flow graph. They propagate their current value to their listeners as soon as they receive enough information.
5. *Updating the memory.* If streams receive enough information from their predecessors in the flow graph, they update their local memory. In the case of $\mathbf{last}^{\mathsf{s}}$ and $\mathbf{delay}^{\mathsf{s}}$ updating the memory requires more information than propagating the newly computed value.
6. *Register next timestamps.* The implementation of $\mathbf{delay}^{\mathsf{s}}$ finally registers its next timestamp from the specification. This registration is done again with every specification evaluation because the specification does not keep track of registered timestamps. Instead, it computes the next timestamp of the minimum of all requested timestamps and the next external timestamp.

Note that the individual steps are not necessarily always executed in that order. Steps 2 and 3 are the only two entry points where a method of the individual object is called. The timestamp is read as part of step 2 or 3, which propagates the computed value, updates the memory and requests the trigger.

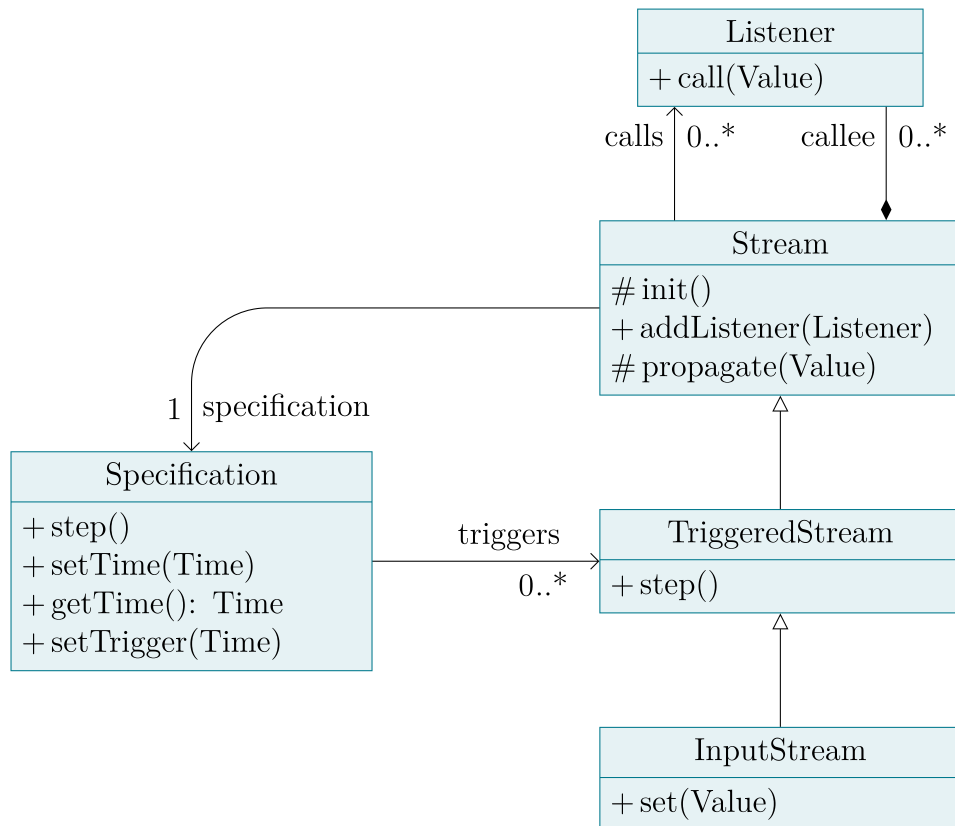


Figure 4.3.: UML class diagram of the interpreter.

4.3.3. Example

Figure 4.4 shows an object diagram for the specification discussed in Example 4.23 in Section 4.1.5. One can see the relation to the dependency diagram for this specification shown in Example 3.33 from Section 3.2.4. Note that the direction of the edges is reversed. In the dependency diagram, an edge depicts a dependency while the direction of the edges in the object diagram corresponds to the data flow: Streams propagate their value to their listeners.

The associations related to the delayed-labelled edges in the dependency graph are drawn in thick and blue. For the current timestamp, the data passed along these associations is not needed to send out an event but only to update the local memory. Hence, every cycle in the graph must contain at least one such edge because otherwise, there would be a deadlock of streams waiting for enough inputs.

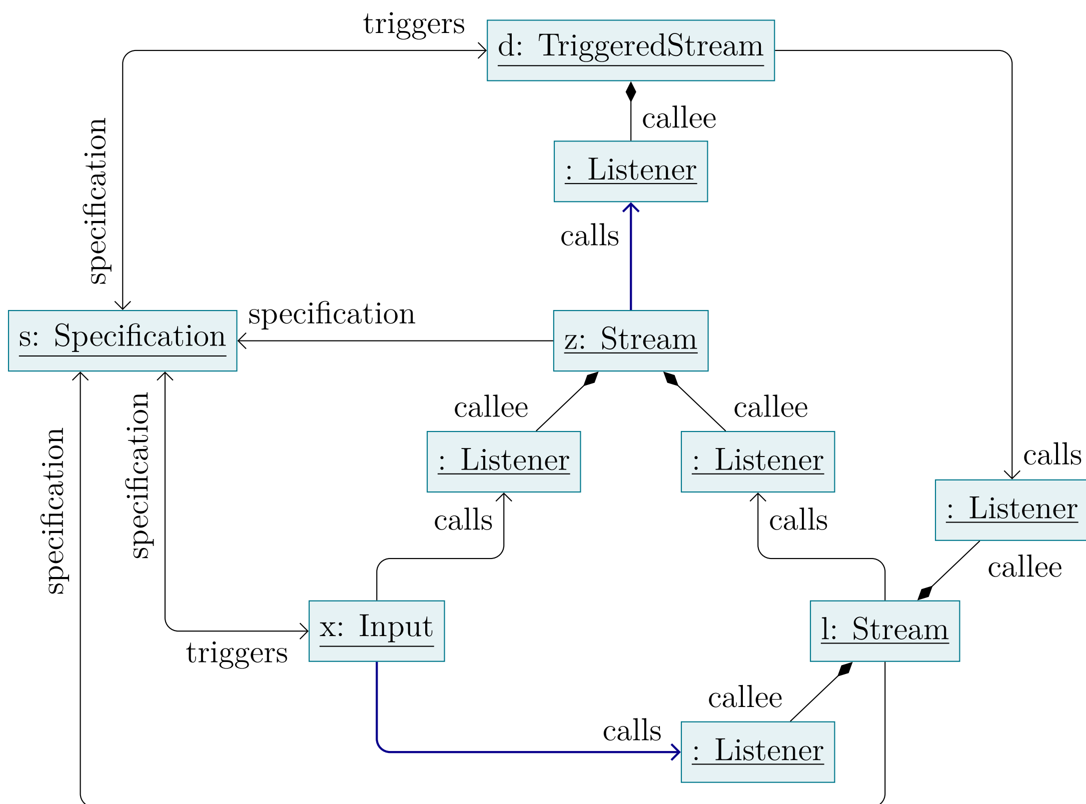


Figure 4.4.: UML object diagram for the specification discussed in Example 4.23.

4.3.4. Scala DSL

This section describes how a TeSSLa specification is translated into a configuration for the interpreter:

Operators are converted into a function that takes streams and creates a new stream which it returns. This way, we already get a very simple internal Scala DSL that can specify TeSSLa specifications. The following code snippet shows how to express the specification from Example 4.23 in Section 4.1.5 in this DSL:

```
var s = new Specification {
  val x = new InputStream
  val z: Stream = merge(x, last(x, delay(z)))
  z.addListener(_.map(v => println(s"$time: z = $v")))
}
```

The recursive stream `z` is defined in terms of `z`, i. e. the stream object `z` depends on itself. For this cyclic definition to work, the listeners are registered lazily. The object `z` cannot accept listeners when the expression `delay(z)` is evaluated first. Thus, every stream has an `init` method, in which the stream registers its listeners. This method is called when the first listener was registered to that stream. In the above example, the listeners are not created and registered until the printing listener is registered to `z`. This late registration ensures that all the stream objects are already created and can accept listeners when they create and register their listeners. This late initialisation works because, for every recursive definition, at least one of the involved streams is used outside of the recursive equation. At least, as in the above example, to print the event's values.

The input is then passed by incrementing the timestamp, setting the input values for that timestamp and calling the specification's `step` method:

```
s.time = 100
s.x.value = Some(BigInt(3))
s.step()
```

This straightforward implementation already supports macro expansion. For example it is possible to define default based on merge and const as follows:

```
def default(s: Stream, value: Any): Stream =
  merge(s, const(value, unit))
```

However, this implementation does not have any proper type system or type checking. There is also no static optimisation or constant folding. Everything is computed at runtime. These features could be added either by extending this basic internal DSL into a proper one or by implementing an external DSL that compiles into this

backend. The latter approach was chosen for the TeSSLa compiler and interpreter available on the TeSSLa website[1]. A thorough discussion of the engineering details of proper type checking and constant folding would go far beyond the scope of this thesis.

The primary purpose of the interpreter lies in its simplicity. The implementation of the operator functions is straightforward due to their synchronous nature: They are provided with all the data they depend on and executed for a global timestamp. The correct execution order is determined dynamically at runtime using message passing. Every operator sends its output to all operators depending on it. Operators without input are scheduled explicitly. This implementation can be easily checked for correctness and is used as a reference implementation for the more advanced implementations discussed in the following chapters.

4.4. Software Compiler

Due to its simplicity, the interpreter presented in the last section is a reference implementation for the TeSSLa semantics. However, it is not very fast. The approach of the interpreter has one main disadvantage: The execution is based on passing messages between objects. The program execution flow is entirely determined by the objects created dynamically at runtime. This dynamic prevents compile-time optimisation, and it can be assumed that the runtime optimisation in the just-in-time (JIT) compilation of the Java virtual machine (VM) [CFM+97, Ven98] cannot gain much in this scenario. All conditions and jumps are entirely based on the dynamic object graph, which the JVM cannot analyse in detail and must assume that it might change in the future. We will see in the evaluation in Chapter 8 that the interpreter is much slower than any other implementation.

In this section, we will discuss a different approach: As the interpreter, the compiler uses the approach from Definition 4.27 (Imperative Algorithm for the Synchronised Monitoring Function) in Section 4.2.1, and the input events are assumed to be provided in a synchronised fashion. However the compiler relies on Definition 4.32 (Linearising Implementation of the Closed Operator Function) in Section 4.2.2. So instead of message passing, plain variables are updated in the loop.

The TeSSLa specification is compiled into imperative code with local variables representing the current value of the streams and the memory cells of the operators. This code is close to a manual implementation and can profit from compile-time optimisation performed by different backends, e. g. the LLVM compiler [Lat02, LA04, Lat12] or the Java VM JIT.

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

4.4.1. Implementing the Synchronised Monitoring Function

The code presented in Definition 4.27 (Imperative Algorithm for the Synchronised Monitoring Function) in Section 4.2.1 is given in an imperative form that can be realised in various programming languages. The body of the main loop consists of three steps:

1. Apply the closed operator function using the current timestamp, the old memory values and the current input stream values.
2. Write the output, i. e. the computed derived stream values.
3. Compute the next timestamp and read the next input stream values if needed.

The imperative algorithm is given in pseudocode, which can be realised with different imperative programming languages. The implementation was done for Java and Rust. See Chapter 8 for an empirical comparison of the two implementations.

Although the translation scheme would generally work with complex data types, the implementation assumes that all data types are primitive. We do not discuss the memory management of complex data structures and assume all data structures to be immutable.

The pseudocode uses the additional value $\perp$ indicating the absence of an event or a value in the memory cell. In order to avoid the overhead of additional objects, especially in the Java implementation, all variables whose data type allows such an additional value are implemented as a variable accompanied by an additional Boolean flag indicating if the value is present or not.

4.4.2. Implementing the Closed Operator Function

The code of Definition 4.27 (Imperative Algorithm for the Synchronised Monitoring Function) in Section 4.2.1 explicitly calls the closed operator function. The implementation inlines the entire code to avoid the overhead of any function calls. The memory variables are local to their basic operator: A memory variable is only used to compute a stream value or to update that particular memory value. Hence, inlining the closed operator function can be done without reproducing any local variables. The input stream variables are immutable inside the closed operator function. They are only changed during the input computation. The derived stream variables are fully immutable. The memory variables are only changed in the final block where all memory values are updated, and the memory values are only updated depending on their old value and the immutable stream values computed above.

The software compiler translates a TeSSLa specification into Rust or Java code, then compiled further with the corresponding compilation toolchain. Such a translation of one high-level language into another is commonly called a source-to-source compiler or transpiler. The overall compilation process benefits from existing optimisation phases in the different compilation backends, e. g. the LLVM compiler or the Java VM JIT.

The code generation follows these principles:

- *Simple Code.* The generated code is kept as simple as possible. We use local variables to store the values of derived streams and the memory cells. The closed operation function is inlined into the loop of the synchronised monitoring function.
- *Structured Programming.* The generated code only uses common features of structured imperative programs, i. e. arithmetic computations, variable assignments, conditional statements and a while loop.

So, in the end, the code consists of conditional variable assignments performed in an endless loop. The code generation was implemented with Java and Rust, and further target languages can easily be added because no language-specific features are used. This language-agnostic implementation has the advantage that the generated code is simple and reusable.

However, it has the disadvantage that compiler-specific features such as function pointers or jumps to memory addresses are not used. For example, [HRR91] introduces the idea for the Lustre compiler not to execute all the synthesised code for every time instant but to only execute those parts of the code whose dependencies have changed. They use jumps to specific addresses to switch between different execution states in which only the currently relevant parts of the code are executed. The Signal compiler uses a similar approach of checking the dependencies statically and only executing those parts of the code whose dependencies are changed. [ABL95] The pacing type refinement and filter refinement for RTLola try to achieve the same goal of not executing code related to streams unaffected by the current timestamp. [BFKS20] For simplicity, such approaches are not considered in this thesis.

4.4.3. Example

As an example for the code generation we continue with the specification from Example 4.23 in Section 4.1.5:

Example 4.34 (Imperative Code)**.** Consider the following TeSSLa specification φ with the free (input) stream $x \in \mathcal{S}_{\mathbb{R}\geq 0}$ and the bound (derived) streams $d \in \mathcal{S}_{\mathbb{U}}$ and $z, \ell \in \mathcal{S}_{\mathbb{R}\geq 0}$ whose dependency graph was shown in Example 3.33 in Section 3.2.4:

$$\begin{aligned} z &= \mathsf{merge}(x, \ell) \\ \ell &= \mathbf{last}(x, d) \\ d &= \mathbf{delay}(z) \end{aligned}$$

For this specification the following pseudo code for the synchronised monitoring function was obtained by applying Definition 4.27 (Imperative Algorithm for the Synchronised Monitoring Function) from Section 4.2.1 and Definition 4.32 (Linearising Implementation of the Closed Operator Function) from Section 4.2.2:

```
// initialisation
(m_ℓ, m_d) := (⊥, ⊥)
(t, x) := read()
(t', x') := read()

// main loop
while t ≠ ⊥ do
  // 1. execute closed operator function
  // 1.1 compute derived stream values
  d := (m_d = t) ? 0 : ⊥
  ℓ := (d ≠ ⊥) ? m_ℓ : ⊥
  z := (x ≠ ⊥) ? x : ℓ
  // 1.2 update memory values
  m_ℓ := (x = ⊥) ? m_ℓ : x
           ⎧ t + z   if z ≠ ⊥ ∧ z > 0,
  m_d :=   ⎨ m_d     if z = ⊥ ∧ m_d ≠ ⊥ ∧ m_d > t,
           ⎩ ⊥       otherwise.
  // 1.3 compute next timestamp request
  c := m_d
  // 2. write events
  write(t, d, ℓ, z)
  // 3. compute next timestamp and next input
  if c ≠ ⊥ ∧ t' ≠ ⊥ ∧ c < t' then
    (t, x) := (c, ⊥)
  else
    (t, x) := (t', x')
    (t', x') := read()
  end
end
```

The code starts with an initialisation of the memory cells. The tuple (t, x) always contains the current timestamp and the corresponding values of the input streams. The tuple (t', x') is the same for the next timestamp. The procedures *read* and *write* are assumed as defined in Definition 4.27. The variable t' becomes $\perp$ if there is no next timestamp in the input.

The main loop first computes the current values of the derived streams d, ℓ and z. Then the memory cells for the $\mathbf{delay}^{\mathsf{s}}$ and the $\mathbf{last}^{\mathsf{s}}$ operator are updated based on the new value of z. Finally, the next timestamp request is set. Since this specification contains only a single operator $\mathbf{delay}^{\mathsf{s}}$ which can make such requests, there is no need to compute the minimum.

Finally, the computed values are written, and the next timestamp is computed. As already discussed in Section 4.2.1, splitting the closed operator function into the computation stream values and memory updates fits precisely the well-established execution scheme discussed in [BCE+03] and shown in Figure 4.1 in this chapter's introduction. The primary adjustment for TeSSLa is this computation of the next timestamp which can be either the next timestamp if the input stream or – if it exists and is smaller – the computed next additional timestamp c. ⌟

See Figure 4.2 and its explanation in Section 4.1.5 for an illustration of the execution of this synchronised monitoring function.

4.4.4. Compiler Frontend

The software compiler uses a compiler frontend, parsing and processing a TeSSLa specification in a textual format. This textual format can be seen as an external DSL in contrast to the internal DSL used in the Scala interpreter.

Example 4.35 (Textual Format of TeSSLa Specifications)**.** Consider the following specification from Example 3.80 (Variable Frequency Period) in Section 3.5.3 which was used as running example in this chapter in Examples 4.23 and 4.34:

Let $x \in \mathcal{P}_{\mathbb{R}^+}$ be a free monitoring stream and let $\ell, z \in \mathcal{P}_{\mathbb{R}^+}$ and $d \in \mathcal{P}_{\mathbb{U}}$ be derived monitoring streams given by the following specification:

$$d = \mathbf{delay}(z)$$
$$\ell = \mathbf{last}(x, d)$$
$$z = \mathsf{merge}(x, \ell)$$

The specification's output consists of the derived stream z.

This specification is written in the textual format as follows:

```
# free input streams
in x: Events[Float]

# derived bound streams
def d: Events[Unit] = delay(z)
def l = last(x, d)
def z = merge(x, l)

# output streams
out z
```

Note that we need an explicit type annotation for at least one stream in a recursive definition. ⌟

This section gives a rough overview of the compiler frontend. Its details are not covered in this thesis. The TeSSLa compiler frontend consists of the following phases:

- *Parser.* The parser is written in ANTLR [PQ95, PF11] and parses an abstract syntax tree from the textual representation of a TeSSLa specification.
- *Type Checker.* The type checker infers the types of the declared streams and performs implicit type conversions defined in Section 3.3.7.
- *Constant Evaluator.* The constant evaluator evaluates constant expressions. It expands derived operators and macros (see Section 3.3) into their definition.

This compiler frontend is used for the further implementations for EPUs, and FPGAs discussed in Chapters 5 and 7, too. The set of basic operators that are not expanded can be configured such that for the different implementations, different sets of operators can be translated directly. As discussed in Section 3.4.2 different sets of basic operators are sufficient. In addition, some derived operators can be translated directly for performance reasons, too.

4.5. Integration and Test Setup

We conclude this chapter with some final remarks regarding the integration and test setup for the interpreter and the software compiler. The interpreter is built with the primary goal of simplicity and correctness and serves as a reference implementation. It was extensively tested with a test suite of simple specifications whose expected outputs were manually defined. The other implementations were tested by comparing their output with the interpreter. With this approach, the correctness of an

implementation can be tested for different complex specifications and input traces without the error-prone task of manually specifying an expected output.

For every implementation considered in this thesis, we will discuss the integration and test setup for two scenarios: Testing the correctness of the implementation by comparison with the interpreter and evaluating its performance as discussed in Chapter 8.

4.5.1. Trace Encoding

The above sections are agnostic to the encoding of the input and output trace. The functions read and write in Definition 4.27 (Imperative Algorithm for the Synchronised Monitoring Function) from Section 4.2.1 are assumed to have the proper side effects.

For debugging and testing purposes, a simple text format for traces is manageable for parsing and writing and, most importantly, easy to read, write and manipulate for humans: Every line represents one event and contains a timestamp, the name of the stream and the event's value. As an example, consider the event on stream with the name *foo* at the timestamp 12 with value 42. Such an event would be represented as follows:

```
12: foo = 42
```

Such a human-readable plain text format becomes inefficient for larger traces because parsing the trace may consume more time than processing the events in the synthesised specification. Binary formats can be read much easier without any parsing.

The Common Trace Format (CTF)[2] is a standardised interchange format for binary traces, which is used by several tracing tools like Linux Trace Toolkit Next Generation (LTTng) [SLD12]. CTF is optimised for complex hierarchical data to be written very efficiently. This format allows minimal intrusive tracing because observed data needs only minimal processing before it is written into the trace. CTF does not specify an exact encoding, but it is a standardised way to describe data traces that follow some conventions.

This thesis uses simple binary formats roughly inspired by CTF and optimised for fast processing, i. e. everything is encoding as 64-bit integers that can be directly mapped to variables, and the order of data values and timestamps is specific for every specification without any metadata stored in the traces.

[2] https://diamon.org/ctf/

The more information about the event pattern in the trace is already available at compile-time, and the more regular this pattern is, the more can it be utilised by the input reader. For example, if a trace contains alternating events on two streams and every event occurs at a different timestamp, then a binary encoding could leave out any form of address or event type encoding. The input reader can read the data based on its internal state. If the input trace is less regular, some event type or address encoding is added. See Chapter 8 for more information on the actually performed experiments.

4.5.2. Test Setup

The software compiler was tested by comparing its output with the interpreter. Figure 4.5 gives an overview of the simple architecture of such an integration test. Both backends were equipped with a parser and printer for the text-based trace format mentioned above. For this setup, the interpreter was attached to the compiler frontend discussed in Section 4.4.4, and the object graph is built dynamically from the parsed specification.

The same specification is used as input for the interpreter and the compiler. The compiler produces source code for the target programming language, i. e. Scala, Java or Rust. In the same way, as the functions read and write in the above pseudocode are assumed to have the proper side effects, the input and output functions are provided as templates to the compiler. The compiler of the target programming language then compiles the source code to a binary which is executed in the runtime environment, e. g. the JVM, or directly on the target platform.

The same input trace is fed into the interpreter and the compiled engine. The output trace of both backends is fed into the trace comparator. The trace comparator sorts the events with the same timestamp alphabetically because they might be produced in arbitrary order. Afterwards, it can simply compare the text-based traces.

4.6. Conclusion

This chapter introduced the synchronised monitoring function on synchronised monitoring streams as an abstraction of the monitoring semantics from Section 3.5. Due to its synchronous nature, the semantics introduced in this chapter generates the least progress. The synchronised monitoring function executes the closed operator function for every timestamp of the input stream. If the specification creates additional timestamps, these are considered and inserted appropriately. The closed

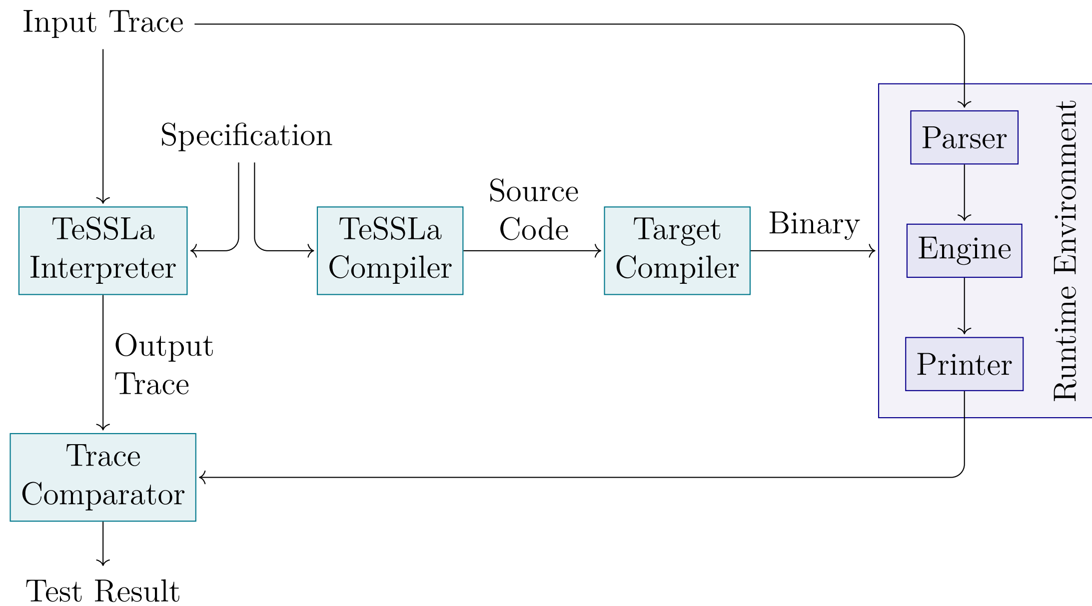


Figure 4.5.: Integration test setup for the software compiler in comparison with the interpreter used as reference implementation.

operator function for a specification is built from the operator functions corresponding to the operators in the specification's flow graph.

The synchronised monitoring function is implemented as a while loop iterating over all timestamps in the interpreter and the software compiler. The interpreter uses message passing to dynamically evaluate the operators in the flow graph in a causal order. The software compiler explicitly performs a topological sorting of the operators in the flow graph and compiles it into a structured program. The interpreter is implemented in Scala; the compiler generates Java or Rust code.

The correctness of the interpreter was tested with manual unit tests; the software compiler was tested by comparing it with the interpreter. The synchronous software implementation will serve as a baseline regarding the evaluation of the EPU and the FPGA implementations discussed throughout the rest of this thesis.

5 TeSSLa on Embedded Procssing Units (EPUs)

In the previous chapter we discussed software-based TeSSLa implementations. For the rest of this thesis we will introduce hardware-based implementations and compare their performance with the software implementations in the evaluation in Chapter 8. The main idea of hardware-based implementations is to utilise the inherent parallelism of hardware by passing events along the flow graph in order to evaluate a TeSSLa specification. We discuss two different approaches to map the flow graph of a TeSSLa specification onto processing hardware: Chapter 7 covers the synthesis of a TeSSLa specification on an FPGA. This synthesis places every operator of the flow graph individually on the FPGA. This chapter presents a different approach. We map the flow graph onto a linear sequential pipeline of processing units that run in parallel.

This thesis uses Event Processing Units (EPU) [Weia, Weib] which are a specially made hardware for stream processing, especially for processing TeSSLa. The EPUs are designed and built by Accemic[1] and are made available synthesised on an FPGA. They are programmed by writing a special configuration into their memory. This chapter discusses how to compile a TeSSLa specification into such an EPU configuration.

The EPUs are organised in a pipeline, i. e. the output events of one EPU are the input events of the next EPU. A single EPU is a processing unit that processes incoming messages and sends outgoing ones further down the pipeline. Such a message encodes a TeSSLa event and consists of a timestamp, a data value and a target address. Inspired by data flow processors the target address indicates how a message is processed by the EPU. Every EPU is equipped with its own data and command memory. The data memory is used to store computed data, and the command memory is used to program the EPU: It contains instructions how to process a message for a given target address. Events from the input streams are encoded as messages to the first EPU of the pipeline. The outgoing messages of the last EPU of the pipeline are interpreted as events of the specification's output streams.

[1] `https://accemic.com`

Accemic has provided the EPU setup, and the author has no access to the source code of the EPUs. All information about the inner workings of the EPUs are taken from publications [DDG+18, CHS+18, DGH+17] and patents [Weia, Weib] as well as private communication with Albert Schulz and Alexander Weiss from Accemic. The EPUs are covered in this thesis as a compilation target. Their design and implementation is not part of this thesis.

TeSSLa is evaluated on the EPUs following the synchronous semantics defined in Section 4.1, i. e. there is no explicit encoding of progress or the absence of events. Instead the timestamp increase is used as a trigger for additional computations. However, the EPUs add pipelining to the synchronous execution model of the interpreter: Instead of a single global current timestamp, every EPU has its own current timestamp, such that the increment of the current timestamp is not required to happen synchronously on all EPUs but can sequentially travel along the pipeline of EPUs. EPUs located further back in the pipeline either have the same or an earlier current timestamp.

For the compilation towards the EPUs, we only consider timestamp-conservative TeSSLa specifications. The introduction of additional timestamps which are not already present in the input is not covered for EPUs in this thesis because it would require the ability to enqueue and sort messages by their timestamps to achieve an effect similar to the execution of the entire closed operator function for the computed next timestamp in Definition 4.27 from Section 4.2.1.

This chapter is organised as follows: Before we discuss the EPU architecture in detail, we first give a rough overview of the concept of data flow processing in order to compare the classic data flow architecture with EPUs. After the introduction of the EOU hardware in Section 5.2 a formal model of this hardware is introduced in Section 5.3 which is used to formally define the implementation of TeSSLa operators on the EPUs and the mapping of TeSSLa flow graphs to an EPU pipeline. We come back to the actual hardware in Section 5.7 and discuss several implementation details and optimisations.

5.1. Data Flow Processors

The von Neumann architecture is the classic control flow architecture. Figure 5.1 depicts a simplified version of this architecture on the left. The program memory contains a sequence of instructions that are executed in sequential order. The control unit has a program counter pointing to the current instruction in the program memory. An instruction can

– load data from the data memory into a register,

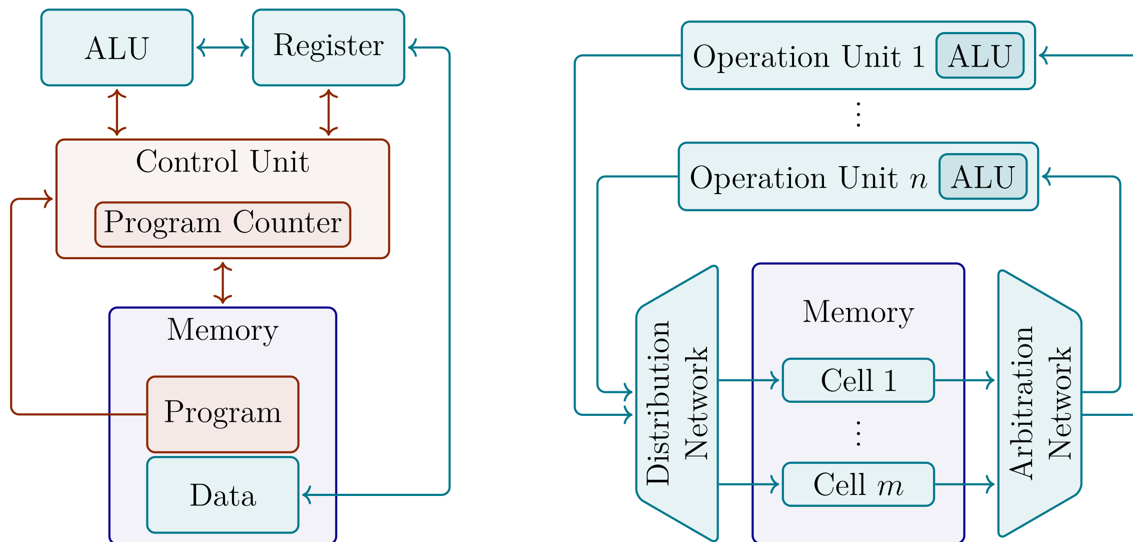


Figure 5.1.: Von Neumann control flow architecture on the left. Diagram based on [SRU99, Fig. 1.1]. Control flow depicted in red and data flow in blue. Basic data flow architecture on the right. Diagram based on [DM74, Figure 2].

– operate on the registers with the arithmetic logic unit (ALU),
– write data from a register to the data memory, or
– (conditionally) manipulate the program counter.

On the other hand a basic dataflow architecture was introduced in [DM74] is shown on the right in Figure 5.1: The memory is split into many cells. In [DM74], each cell contains:

– two input data values,
– two ready bits,
– an instruction code, and
– a target address.

The arbitration network selects cells whose ready bits are set and loads them into an operation unit. Every operation unit is equipped with an ALU. Multiple operation units can work independently in parallel and execute their cell's instruction with their ALU. The distribution network writes the operation units' results into the cells' data values indicated by the target address. The ready bits of the data values are set when the data value is written and erased when the cell is executed.

This section can only give a rough overview of the basic principles of data flow processors as far as these are needed to classify the concept of EPUs. For a more detailed overview and introduction into data flow processors see for example [HK08] and [SRU99, Chapter 2].

While a control flow architecture processes the program code in sequential order, the execution order of a data flow program is only determined by the availability of the input data values of the cells. This scheduling of the cells is done dynamically at runtime. As soon as the ready bits are set, a cell can be chosen for execution by the arbitration network. The literature refers to such a cell as one that can fire. There is no fixed execution order if multiple cells can fire. Every cell can be executed on every operation unit; there is no static mapping from cells to operation units. The mapping is done entirely dynamic at runtime by the arbitration network.

While imperative languages are compiled into a sequence of instructions for a control flow processor, data flow processors can execute data flow graphs. A *data flow graph* is a directed graph consisting of nodes representing operations and edges representing data flow, i. e. input dependencies and output targets. The nodes of a data flow graph are represented by the memory cells of the data flow architecture. The edges are encoded as the cell's target addresses. The execution semantics of data flow graphs represent the data flow architecture as follows: Every edge can hold at most one data value, and a node can be executed if all its input dependencies are fulfilled, i. e. if every incoming edge holds a data value. Executing a node performs is arithmetic logic operation and consumes the data values on its input edges, and produces data values on its output edges. Data flow graphs are functional and composable, i. e. the operations associated with the nodes cannot have any side effects, and the outputs of one graph can be directly used as inputs for another graph.

Languages which can be compiled into data flow graphs are *single-assignment languages*. (see for example [SRU99, Chapter 2]): Every variable can only be written once, i. e. it may only appear once on the left-hand side of an assignment.

The first data flow architecture whose simple approach is described above and visualised in Figure 5.1 on the right was introduced in [DM74] and many extensions and improvements have been made since then: Every edge of the data flow graph can hold at most one value because the input value field of the cell in the memory is statically mapped to the edge of the graph. A dynamic mapping from edges of the graph to memory locations was introduced in [Vee86] to overcome this limitation: Every value is equipped with a tag (sometimes also called colour) indicating its computational context. A node can be executed if values carrying the same tag for all inputs are available. The generated output carries the same tag. New tags are generated, for example, with every loop iteration or procedure call. This approach is called dynamic or tagged-token data flow and improves the parallelism of the architecture. An *explicit token store (ETS)* [AC03] takes this approach one step further and replaces the inefficient tag matching with a direct mapping between tags and memory offsets. Another extension is the addition of a pipelining of the steps of the operation unit [DG88]: The *Pipelined Instruction Processing Unit (PIPU)* performs the steps instruction fetch, operand fetch, scalar operation and result store in

a pipeline. This pipeline never stalls because the *Dataflow Instruction Scheduling Unit (DISU)* only schedules nodes that can be executed. Monsoon [PC90] is an example of a commercially available ETS computer.

5.2. EPU Hardware

The central motivation for data flow architectures is their inherent parallelism. A fixed input is encoded in the processor's memory together with a data flow graph which is then executed in a highly parallel fashion. TeSSLa's flow graphs as defined in Definition 3.31 in Section 3.2.4 are quite similar to data flow graphs. However, there are several main differences between the data flow architectures presented in the last section and the requirements for TeSSLa implementations as discussed in Chapter 1:

- Data flow processing is not explicitly optimised for stream processing. We are in the specific setting of online monitoring: We want to process events on an input stream in their order of appearance without the need to store the entire stream in the memory.
- Data flow graph's semantics can be seen as an event processor: If there is a value for all inputs of a node, that node can fire and produces an output event. TeSSLa is a synchronous language that requires additional synchronisation mechanisms to preserve the relation of input and output events regarding the logical timing of events (see synchronous hypothesis in Chapter 1). Because of the synchronisation based on timestamps used in the TeSSLa semantics, such a TeSSLa flow graph cannot be directly executed by the simple data flow architecture introduced in the last section.
- TeSSLa uses explicit timestamps attached to every event in order to synchronise events across different streams. Synchronising events and maintaining the event's order based on their timestamps is a crucial aspect of a TeSSLa implementation.
- The idea behind tagged-token data flow is to allow the parallel execution of operations that depend on the same data. However, we target especially the scenario of many event streams, which are independent of each other most of the time. So the main problem which motivated the tagged-token architecture is not present: While waiting for data, independent operations on other data can be executed.

On these grounds, we consider two conceptional differences between the EPU architecture and the data flow architecture discussed in the last section:

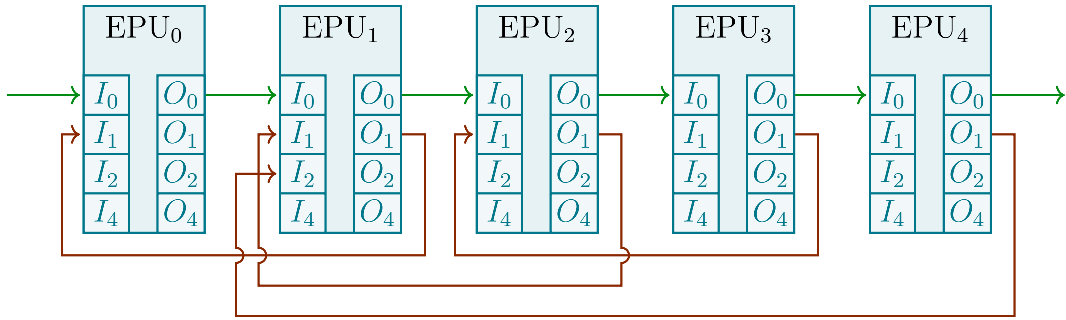


Figure 5.2.: Exemplary message routing between EPUs. Diagram based on [Weia, Weib, Fig. 22].

1. Every EPU is a processor equipped with a controller, an arithmetic logic unit (ALU) and a separated data and command memory. Commands are statically assigned to a single EPU and not dynamically scheduled at runtime. However, the individual EPUs resemble the idea of data flow processing because the incoming events entirely determine the executed operation: The command memory contains instructions at dedicated addresses, called command IDs. Every incoming message contains a command ID next to its data that specifies how the EPU should process the data.

2. EPUs follow the idea of stream processing and process the incoming events in the order of their arrival. There is no central scheduling, like the arbitration network of data flow processors, that decides at runtime which nodes can be fired.

Figure 5.2 shows the sequential pipeline of EPUs. For the simple case of acyclic TeSSLa specifications, we only consider the green arrows and ignore the existence of multiple inputs and outputs. Messages are coming from a message source on the left which is not shown in the diagram. Every EPU has an input on the left, where it reads incoming messages. After processing the message according to its command ID, an output message might be generated with a new command ID. The output message is then sent to the next EPU.

The general idea of the mapping of a TeSSLa specification's flow graph onto such a pipeline of EPUs is as follows: The dependency graph is layered such that nodes connected by an edge are located on different layers. Every EPU of the pipeline roughly resembles a layer of the specification's dependency graph. The details are given in Section 5.5 and a concrete example is discussed in Section 5.5.1 after introducing a formal model of the EPUs, and a mapping of the TeSSLa operators to EPU commands in the following sections.

If the the specification's flow graph contains cycles, then messages must be passed

address		data	
EPU ID	command ID	timestamp	value

Figure 5.3.: Message frame encoding of the EPUs. Diagram based on [Weia, Weib, Figs. 5–8].

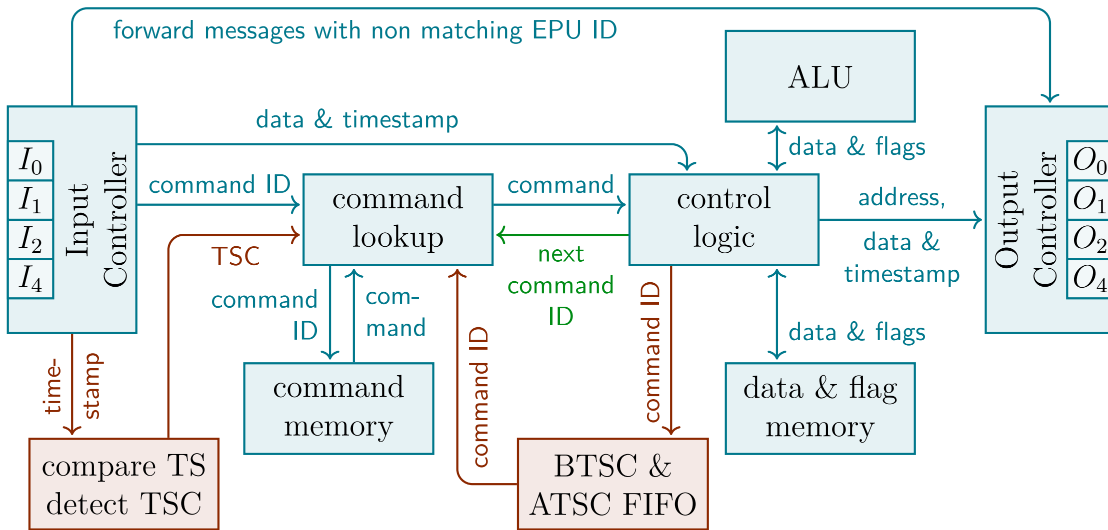


Figure 5.4.: Architectural overview of the components of an Event Processing Unit (EPU). The diagram based on [Weia, Weib, Figs. 4, 9, 10–12] shows the conceptional data flow through the components of an EPU.

from later EPUs back to earlier EPUs in the pipeline. How to achieve this in accordance with the TeSSLa semantics is discussed in Section 5.6. Technically this is supported by additional outputs which are not connected to the next EPU but earlier EPUs in the pipeline. These connections are shown in red in Figure 5.2. One output can only be connected to one input, but these connections can be configured freely through switch boxes in a configurable switching network. The simple pipeline connection shown in green which connects every EPU with the next one, is always there. The feedback connection shown in red which connects some EPUs with other EPUs located earlier in the pipeline, is configured based on the needs of the specification.

Figure 5.3 shows the encoding of a message in detail: The address consists of the target EPU of the message and the command ID. The EPU ID specifies which EPU is supposed to handle the message. If an EPU reads a message not addressed to it, it directly forwards the message to the next EPU in the pipeline. The command ID determines how the EPU processes the message. The data consists of the event's timestamp and value.

Figure 5.4 shows an architectural overview of a single EPU. It shows the conceptional data flow through the components of an EPU. Depicted in blue is the default data flow through the EPU: The first step in processing an incoming message is comparing the target EPU encoded in the message address with the ID of the EPU. If the target ID does not match, the EPU forwards the message to the output controller without changing it. With this forwarding mechanism, messages addressed to EPUs located later in the pipeline are correctly dispatched.

Every EPU has a command memory which is programmed statically at compile time. It maps command IDs to commands. A command contains instructions how to process incoming messages. If the message is addressed to this EPU, the command ID is looked up, and the corresponding command is loaded from the command memory.

The control logic then processes the command. A command can consist of any of the following steps:

- Loading flags and data from memory,
- processing the message's timestamp or value or load data in the ALU,
- writing the message's timestamp or value or the ALU result to the memory,
- writing flags to the memory,
- sending a message to the output controller.

If a command sends out a message, this message's target address, consisting of the EPU ID and the command ID, is determined only by the command. The command must statically provide the target EPU ID and command ID independently of the message's timestamp, value, loaded memory values, flags, or computed values. The message's value can be chosen from the memory, the incoming message's value or the ALU result.

Depicted in green is the handling of next commands: A command can specifying a next command ID. This command is looked up and processed immediately after the current command without loading a new message.

Depicted in red are the components related to the *timestamp change (TSC)*: An EPU has a current timestamp determined by the currently processed message. An increment of this timestamp can schedule additional commands: A command can schedule command IDs into the *before timestamp change (BTSC)* or *after timestamp change (ATSC)* FIFO. The corresponding commands are processed before or after the timestamp change, respectively. The timestamp of every incoming message is compared with the current timestamp. If the incoming timestamp is larger than the current timestamp, several steps are executed:

a) the *before timestamp change (BTSC)* FIFO is processed,
b) the current timestamp is updated,

c) the *after timestamp change (ATSC)* FIFO is processed, and finally
d) the incoming message is processed.

The outgoing message's timestamp is always the current timestamp of the EPU; it can never be changed. As a result, EPUs can only realise timestamp-conservative specifications.

A final important concept of EPUs used to realise cyclic TeSSLa specifications is the *blocking counter*: The blocking counter can be incremented or decremented by a command. If the blocking counter is above zero, the EPU does not process incoming messages, which would change the current timestamp of the EPU. The blocking counter is used to realise synchronisation mechanisms between recursive messages coming through the feedback connections (shown in red in Figure 5.2) with the regular messages travelling down the pipeline through the forward pipeline connections (shown in green in Figure 5.2). Compiling recursive TeSSLa specifications to EPUs is discussed in detail in Section 5.6.

This thesis only considers a single sequential pipeline of EPUs. Technically it would be possible to have multiple pipelines of EPUs running in parallel and connecting these pipelines with switching networks similarly to the ones used for the feedback connections for recursive messages. This approach, however, raises multiple problems which are not addressed in the context of this thesis:

1. It requires a synchronisation based on the timestamps when the events of multiple pipelines are supposed to be joined again. Such a synchronisation could be realised similarly to the recursive feedback connections, but we will see later that this mechanism is not very efficient in terms of possible throughput.

2. This approach would further require an automatic allocation during the translation from a TeSSLa specification to a configuration of the EPU pipeline. Mapping every branch of the dependency graph onto a separate pipeline is impossible because the number of available EPUs is limited, so a choice based on the event frequencies and the recursions' depth of different branches would be required. A complex analysis of the expected throughputs occurring in different parts of the specification is needed to solve the optimisation problem of levelling the potential throughput gain from the parallel pipelines and the slowdown caused by the additional synchronisations.

3. The main issue, however, is the limited number of EPUs. For the practical evaluations in the context of this thesis, a pipeline with 12 EPUs was available. This thesis assumes that it is more efficient in this scenario to use the EPUs in a single pipeline than splitting them up into multiple pipelines.

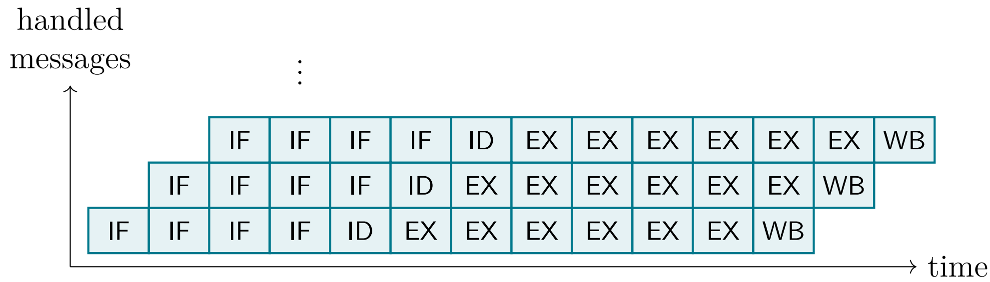


Figure 5.5.: Schematic visualisation of the stages of the inner pipeline over time.

5.2.1. Inner Pipeline

The pipeline of connected EPUs discussed so far will be called the outer pipeline in the following. The event processing in a single EPU is pipelined, too. We call this the inner pipeline. The technical details of the EPU's realisation are neither known to the author nor relevant for this thesis. This section gives a rough overview of the inner pipeline as far as it is required to understand the evaluation performed on the EPUs.[2]

The processing of an incoming message on an EPU is implemented in several phases:

- *Instruction Fetch (IF)* performs everything from comparing the message's timestamp with the current timestamp to loading the command from the command memory.
- *Instruction Decode (ID)* decodes the instructions loaded from the command memory and assigns the correct values to the ALU based on the command's instructions.
- *Execute (EX)* executes the command, i. e. performs the computation in the ALU and the logical computations based on the flags and the ALU result.
- *Write Back (WB)* writes values and flags to the memory and sends an output message.

The internal pipeline has 12 stages. Every stage takes one clock cycle on the FPGA. IF takes 4 stages, ID a single stage, EX 6 stages and WB again a single stage. Figure 5.5 shows a visualisation of the pipeline stages over time. In the optimal case, a new incoming message can be handled with every new clock cycle. If a command chains another command via the next command chain or commands from the BTSC or ATSC FIFO are executed, those commands prevent further processing

[2]The information about the inner pipeline presented in this section are based on private communication with Albert Schulz and Alexander Weiss from Accemic.

of input messages. The pipeline is stalled for one cock cycle in those cases. In the following cases, the pipeline is not only stalled, but rolled back in order to fulfil causal dependencies between sequential commands:

- During the ID phase, the EPU checks whether the current command has a next command set in its command instructions loaded from the command memory. If a next command is configured for the current command, then this next command must be executed immediately after the current one before any other commands whose processing might already have been started in the pipeline. The pipeline thus rolls back the already fetched commands.
- During the ID phase, the EPU checks whether the current command might schedule a command for BTSC or ATSC execution or might change the blocking counter. Such changes must be considered before executing the following incoming message. If that message changes the current timestamp of the EPU, the blocking counter must be considered, and BTSC or ATSC executions are handled first. Hence, the EPU flushes the IF stages and stalls the entire internal pipeline until this command has passed the WB stage.
- The EPU checks for data dependencies when reading or writing data or flags from or to the memory. Examples for data dependencies are the command executed next reading the same flags or data written immediately before. The IF and ID stages are stalled in those cases until the current command has passed the WB stage.

5.3. Formal EPU model

This section abstracts from the concrete EPU hardware presented in the last section and introduces a formal model. This model is used to discuss the translation of a TeSSLa specification into an EPU configuration. It includes an operational semantics of an EPU pipeline.

The behaviour of an EPU is determined by the commands in its command memory. Consequently these commands are the central element of the representation introduced in this chapter. Intuitively, an *EPU command* is represented as a function mapping their input message and their current memory to an (optional) output message and an updated memory. The commands interact with each other by sending messages, causing a next command to be executed and scheduling commands for BTSC and ATSC execution. This is represented by a graph, called *EPU network*. This graph is organized in layers representing the EPUs. Every command is located on an EPU and can only issue next commands and enqueue commands located on the same EPU. It can only send messages to commands located on other EPUs.

We start with the formal definition of an EPU command and organize them in an EPU network next:

Definition 5.1 (EPU Command)**.** Let $\mathcal{C}$ be the set of all EPU commands:

$$\mathcal{C} = \mathbb{T} \times \mathbb{T} \times \mathbb{D} \times \mathbb{D}^n \to \mathbb{D}_\perp \times \mathbb{D}^n \times \{-1, 0, 1\}.$$

An *EPU command* $c \in \mathcal{C}$ with $c(t, u, i, \boldsymbol{m}) = (o, \boldsymbol{m}', b)$ takes

- a current timestamp t,
- a received timestamp u,
- an input value i and
- a memory tuple $\boldsymbol{m}$.

It produces

- an output value o which can be $\perp$ indicating no output and
- an updated memory tuple $\boldsymbol{m}'$ and
- a delta b for the blocking counter. ⌟

In this representation, we merge the data and flag memory into a single memory tuple $\boldsymbol{m}$. In the following we assume the naming convention $e(t, u, i, \boldsymbol{m}) = (o, \boldsymbol{m}', b')$ and denote EPU commands as relation between these variables given in imperative pseudocode assigning the output variables. If not stated otherwise we assume no output, i. e. $o = \perp$, the identity mapping of unmentioned entries of the memory tuple, i. e. $\boldsymbol{m}' = \boldsymbol{m}$, and no changes to the blocking counter, i. e. $b' = 0$. For example with the memory tuple $\boldsymbol{m} = (m_1, m_2) \in \mathbb{B}^2_\perp$ the notation

$$e\colon \text{if } m_1 \text{ then } o = i,\ m'_2 = \mathit{false}$$

denotes the function

$$e(t, u, i, m_1, m_2) = \begin{cases} (i, m_1, \mathit{false}, 0) & \text{if } m_1, \\ (\perp, m_1, m_2, 0) & \text{otherwise.} \end{cases}$$

Definition 5.2 (EPU Network)**.** An *EPU network* consists of a finite set V of command nodes, a finite set O of output nodes, and a finite set E of EPU nodes with the following functions:

- $\mathit{cmd}\colon V \to \mathcal{C}$ maps a command node to its EPU command.
- $\mathit{epu}\colon V \to E$ locates every command node on an EPU node.
- $\mathit{first} \in E$ indicates the first EPU node.
- $\mathit{succ}\colon E \to E_\perp$ maps an EPU node to its successor.

- $target\colon V \to (V \cup O)_\perp$ maps a command node either to another command node or to an output node. The EPU command of this node sends outgoing messages to the configured target.
- $btsc\colon V \to V_\perp$ and $atsc\colon V \to V_\perp$ maps a command node to a command node whose command is added to the BTSC or ATSC set, respectively, when the first node's command is executed. Further, the overloaded function $btsc\colon E \to 2^V$ and $atsc\colon E \to 2^V$ maps an EPU node to a set of initially scheduled commands for BTSC or ATSC execution, respectively.
- $next\colon V \to V_\perp$ maps a command node to a next command node whose command is executed immediately after the current node's command.
- $init\colon E \to M$ maps an EPU to its initial memory tuple with M being the set of all possible memory values of the EPU, i. e. the Cartesian product of all possible memory values of the EPU commands of the EPU. ⌟

Note that a node can have no *target*, *btsc*, *atsc* or *next* node because the corresponding functions can map to $\perp$ instead of a value. In the same way the last EPU does not have a successor. Where convenient we extend *epu* to output nodes and use the convention $epu(o) = \perp$ for any output node $o \in O$.

Definition 5.3 (Well-Formed EPU Network)**.** An EPU network is called *well-formed* iff

- The EPU nodes in E are totally ordered with *first* being minimal, i. e. when starting with the first node one eventually reaches all nodes when iterating their successors.
- A command node's target must be located on a different EPU, i. e. for any two command nodes $v_1, v_2 \in V$ we have

$$target(v_1) = v_2 \implies epu(v_1) \neq epu(v_2).$$

- A command node's next command, BTSC command and ATSC command node lay all on the same EPU if they exist, i. e. for any two nodes $v_1, v_2 \in V$ we have

$$\begin{aligned} next(v_1) = v_2 &\implies epu(v_1) = epu(v_2),\\ btsc(v_1) = v_2 &\implies epu(v_1) = epu(v_2) \text{ and}\\ atsc(v_1) = v_2 &\implies epu(v_1) = epu(v_2). \end{aligned}$$

⌟

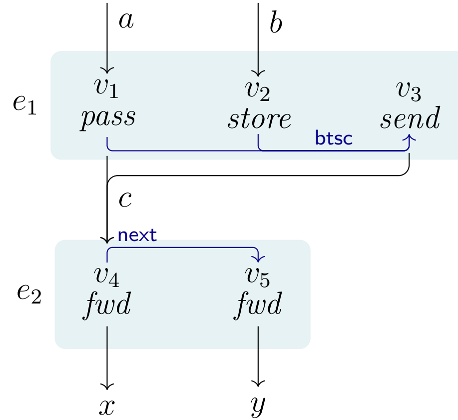


Figure 5.6.: EPU network diagram for the EPU network from Example 5.4.

In the following we only consider well-formed EPU networks.

EPU networks are specified using EPU network diagrams. See Figure 5.6 for an example of such a network diagram: The black arrows indicate the target addresses configured for outgoing messages. Incoming and outgoing arrows are labelled with the names of streams indicating how the translated commands are connected to an entire EPU network and how inputs and outputs are mapped to the EPU network. The blue arrows labelled with next, btsc and atsc represent the scheduling of other commands or the execution of the next command. Blue arrows without a source indicate commands being initially scheduled for BTSC or ATSC execution.

Example 5.4 (EPU Network)**.** As an example for an EPU network consider its diagram depicted in Figure 5.6. This network merges incoming messages on a and b into c and duplicates the output into x and y. The merging prioritizes messages on a in case of messages with the same timestamp. The commands on the first EPU are a simplified version of the commands given later in Definition 5.10 (Commands for merge).

The network consists of the set $E = \{e_1, e_2\}$ of EPU nodes, the set $V = \{v_1, v_2, \dots, v_5\}$ of command nodes and the set $O = \{x, y\}$ of output nodes. The command nodes v_1, v_2 and v_3 are located on the first EPU, i. e.

$$epu(v) = e_1 \text{ for } v \in \{v_1, v_2, v_3\}.$$

Further, the command nodes v_4 and v_5 are located on the second, i. e.

$$epu(v) = e_2 \text{ for } v \in \{v_4, v_5\}.$$

The EPUs are organized sequentially starting with $\mathit{first} = e_1$:

$$succ(e_1) = e_2 \text{ and } succ(e_2) = \bot.$$

The command node's outputs are routed as follows: v_1 and v_3 send messages to v_4:

$$target(v_1) = target(v_3) = v_4,$$

and the command nodes v_4 and v_5 send messages to x and y, respectively, i. e.

$$target(v_4) = x \text{ and } target(v_5) = y.$$

The diagram further includes the edge labels a, b and c, which are technically not part of the EPU network.

The command node v_1 passes on incoming messages and sets the flag m_{sent} to *true*:

$$cmd(v_1) = c_{pass} \text{ with } c_{pass}\colon m'_{sent} = true,\ \ o = i$$

The command node v_2 stores the value of incoming messages in the memory m_{value}:

$$cmd(v_2) = c_{store} \text{ with } c_{store}\colon m'_{value} = i$$

The command nodes v_1 and v_2 schedule the command node v_3 for BTSC execution:

$$btsc(v_1) = btsc(v_2) = v_3 \text{ and } btsc(v) = \bot \text{ for } v \in \{v_3, v_4, v_5\}.$$

The command node v_3 is executed before a timestamp increase if a message on a or b was received. If only a message on b was received then this message is send out. In any case the flag m_{sent} is reset:

$$\begin{aligned} cmd(v_3) = c_{send} \text{ with } c_{btsc}\colon & m'_{sent} = false, \\ & \text{if } \neg m_{sent} \text{ then } o = m_{value} \end{aligned}$$

The memory of e_1 is initialized by $init(e_1)$ as follows:

$$m_{sent} = false \text{ and } m_{value} = 0.$$

The command nodes v_4 and v_5 pass every incoming message without any further action:

$$cmd(v_4) = cmd(v_5) = c_{fwd} \text{ with } c_{fwd}\colon o = i$$

The command node v_4 issues v_5 as next command, i. e. the command of v_5 will be executed immediately after v_4 on the same input message:

$$next(v_4) = v_5 \text{ and } next(v) = \bot \text{ for } v \in V \backslash \{v_4\}.$$

Thus every incoming message on c is send out twice to x and y. ⌟

5.3.1. Execution of a single EPU

Let V be the command nodes and E the EPU nodes of an EPU network. We use the following procedures, which receive and send messages on an EPU. A message consists of a target, which is either a command node or an output node, a data value and a timestamp. We use $\mathcal{M} := (V \cup O) \times \mathbb{D} \times \mathbb{T}$ for the set of all messages for a given EPU network. The procedures are assumed to have imperative semantics with side effects:

- $\mathit{receive}() = m$ takes no arguments and returns the next message $m \in \mathcal{M}$ addressed to the given EPU. A message consists of a target, a timestamp and a data value.
- $\mathit{nextTime}() = t$ takes no arguments and returns a timestamp $t \in \mathbb{T}$. It allows for peeking at the timestamp of the next message without receiving it.
- $\mathit{send}(m)$ takes a message $m \in \mathcal{M}$ and returns nothing. It sends an outgoing message, again consisting of a target command node, a timestamp and a data value.

These procedures are blocking until there are messages available or until the outgoing message was send. See the next section on their implementations, i. e. their side effects.

For every EPU $e \in E$, we assume the following variables:

- $t \in \mathbb{T}$ contains the current timestamp of the EPU initialised with 0,
- $b_c \in \mathbb{N}$ is the current value of the EPU's blocking counter initialised with 0,
- $b, a \subseteq V$ store the commands scheduled for BTSC and ATSC execution and
- $\boldsymbol{m} \in \mathbb{D}^k$ is the memory tuple of the EPU.

The initial assignment of the memory tuple $\boldsymbol{m}$ and the sets b and a is given by the functions *init*, *btsc* and *atsc* of the EPU network, respectively. The FIFOs of the actual hardware are represented as sets. This abstraction simplifies the encoding of the commands, because a command can only be scheduled once each for BTSC or ATSC execution. See Section 5.7.4 for a discussion on how to implement the BTSC and ATSC scheduling on the actual hardware.

The following pseudocode describes the execution of a single EPU $e \in E$:

loop
  wait until $b_c = 0 \vee \mathit{nextTime}() = t$
  $(c, d, u) = \mathit{receive}()$
  if $u > t$ **then**
    for all $c' \in b$ **do** $\mathit{handle}(c');\ b = b \setminus \{c'\}$
    $t = u$

for all $c' \in a$ **do**
$handle(c');\ a = a \backslash \{c'\}$
end
if $epu(c) = e$ **then** $handle(c)$ **else** $send(c, d, u)$
end

The procedure $handle(v)$ with imperative side effects executes a command node. It takes a command node $v \in V$ as argument and returns nothing. This execution includes applying the actual EPU command of the command node, sending the output message, and handling the blocking counter, the BTSC and ATSC scheduling and the next command of the node. The details are given by the following pseudocode:

$(o, \boldsymbol{m}, b) = cmd(c)(t, u, d, \boldsymbol{m})$
if $o \neq \bot \wedge target(c) \neq \bot$ **then** $send(target(c), o, t)$
$b_c = b_c + b$
if $btsc(c) \neq \bot$ **then** $b = b \cup \{btsc(c)\}$
if $atsc(c) \neq \bot$ **then** $a = a \cup \{atsc(c)\}$
if $next(c) \neq \bot$ **then** $handle(next(c))$

For simplicity, we assume that every EPU command implicitly knows which part of the memory tuple is relevant for it and only updates that part of the memory tuple while returning the rest of the EPU's memory unchanged.

5.3.2. Execution of an EPU Network

Based on the execution of a single EPU, we can now describe the execution of the entire EPU network. First, the side effects of the procedures *receive*, *nextTime* and *send* are given. These procedures are used in the above pseudocode, and they define how the message passing between the EPUs works. On the actual hardware, an outgoing message on an EPU is available as an incoming message on the following EPU in the next clock cycle. There is no buffer between the EPUs. If the next EPU is not ready to receive a message, the current EPU cannot send a message. If this blocking continues through multiple EPUs on the outer pipeline, it causes backpressure.

In order to model that EPUs have multiple inputs, one regular input and depending on the routing up to three recursive inputs, every EPU $e \in E$ has two queues:

- An input queue, which is a priority queue with length 1 per priority, i. e. two variables. We write $e.\,input, e.\,importantInput \in \mathcal{M}_\bot$.
- An output queue, which is a queue with length 1, i. e. one variable. We write $e.\,output \in \mathcal{M}_\bot$.

EPUs are equipped with a priority queue to avoid deadlocks: Sometimes EPUs are waiting for certain recursive messages, i. e. those coming from an EPU located later in the pipeline, before they are allowed to accept messages with a larger timestamp. In these cases the EPU must be able to receive recursive messages even if the regular input queue already contains a message.

The procedures *send*, *receive* and *nextTime* perform the following operations on these queues:

– The procedure *send* puts its argument into the output queue of the current EPU.
– The procedure *receive* takes the value from the input queue, respecting the priority, i. e. an element from the variable *importantInput* is used in preference to an element from the variable *input*.
– The procedure *nextTime* reads the time from the input queue following the same rules as *receive* but leaving the queue unchanged.

All procedures are considered blocking if the corresponding queue is full (writing) or empty (reading).

The execution of a single EPU was described in the previous section as an endless loop. The loop is executed for every EPU in parallel. On the actual hardware, the EPUs are operating independently on the FPGA, too. Every EPU processes messages from its input queue and writes messages to its output queue. The following code describes how messages between these queues are dispatched. Altogether, the EPU network implements a translation from a synchronised input stream to a synchronised output stream. The next section addresses the mapping between events in a synchronised stream and nodes in the EPU network. In the following pseudocode, we assume two procedures with imperative side effects:

- The procedure $\mathit{readMessage}() = m$ takes no arguments and returns a message $m \in \mathcal{M}$. It returns the next input event from the synchronised input stream in the form of a message, i. e. a target node, a data value and a timestamp.
- The procedure $\mathit{writeMessage}(m)$ takes a message $m \in \mathcal{M}$ and returns nothing. It appends the content of the message as an event to the synchronised output stream.

The execution of the EPU network is given by the following pseudocode which is executed in parallel with the individual EPUs:

loop
  for all $e \in E$ **do**
    if $e.\mathit{output} \neq \bot$ **then**
      $(c, d, u) = e.\mathit{output}$
      if $\mathit{epu}(c) > e$ **then**

```
        if succ(e) ≠ ⊥ ∧ succ(e). input = ⊥ then
          e. output = ⊥
          succ(e). input = (c, d, u)
        end
        else if succ(e) = ⊥ then writeMessage(c, d, u)
      else if epu(c) < e ∧ epu(c). importantInput = ⊥ then
        e. output = ⊥
        epu(c). importantInput = (c, d, u)
      end
    end
  end
  if first . input = ⊥ then first . input = readMessage()
end
```

If an EPU e wants to send an output, we distinguish two cases:

1. If the target EPU is located later in the pipeline ($epu(c) > e$), then the message is transmitted to its successor $succ(e)$.
2. If the target EPU is located earlier in the pipeline, then the message is transmitted directly to the target EPU $epu(c)$. On the actual hardware, there must be a feedback connection (shown in red in Figure 5.2) available connecting e and $epu(c)$. For such recursive messages, we use the *importantInput* of the target EPU.

Transmitting a message is only possible if the target input is currently empty. Otherwise, the message stays in e's output until it can be transmitted in a future cycle.

The two edge cases of the first and the last EPU in the pipeline are handled by feeding events taken from *readMessage* into the first EPU's input and calling *writeMessage* instead of transmitting a message if an EPU has no successor.

The following examples demonstrates the execution of an EPU network including the execution of its EPUs:

Example 5.5 (Execution of an EPU Network)**.** Recall the EPU network from Example 5.4 (EPU Network): Its first EPU merges merges messages on a and b into c, prioritizing messages on a in case of messages with the same timestamp. The second EPU duplicates the output into x and y. We send a message with value 2 at timestamp 7 into a, followed by a message with value 4 at timestamp 9 into b. The first EPU passes on both messages. The first because it is the only message with that timestamp and the second because it is prioritized. Both messages are then duplicated to the outputs x and y by the second EPU.

In the following walk through we ignore the counter b_c and the ATSC set a, because they stay 0 or $\emptyset$, respectively. The timestamps of both EPUs are initialized with 0 and the memory of EPU is initialized according to $\mathit{init}(e_1)$:

$$e_1 \colon t = 0,\ m_{sent} = \mathit{false},\ m_{value} = 0,\ b = \emptyset$$
$$e_2 \colon t = 0,\ b = \emptyset$$

The input message $(v_2, 7, 2)$ is dispatched to e_1.

The EPU e_1 receives and processes $(v_2, 7, 2)$: It updates its timestamp to 7 and executes $\mathit{handle}(v_2)$. The execution of $\mathit{cmd}(v_2) = c_{store}$ updates m_{value} and adds v_3 to the BTSC set:

$$e_1 \colon t = 7,\ m_{sent} = \mathit{false},\ m_{value} = 7,\ b = \{v_3\}$$

The input message $(v_2, 9, 4)$ is dispatched to e_1.

The EPU e_1 receives and processes $(v_1, 9, 4)$: Before it changes its timestamp, it executes $\mathit{handle}(v_3)$ and removes v_3 from b. The execution of $\mathit{cmd}(v_3) = c_{send}$ runs $\mathit{send}((v_4, 7, 2))$.

$$e_1 \colon t = 7,\ m_{sent} = \mathit{false},\ m_{value} = 7,\ b = \emptyset$$

The sent message $(v_4, 7, 2)$ is dispatched from e_1 to e_2.

The EPU e_1 continues to process $(v_1, 9, 4)$: It updates its timestamp to 9 and executes $\mathit{handle}(v_1)$. The execution of $\mathit{cmd}(v_1) = c_{pass}$ runs $\mathit{send}((v_4, 9, 4))$, updates m_{sent} and adds v_3 to the BTSC set:

$$e_1 \colon t = 9,\ m_{sent} = \mathit{true},\ m_{value} = 7,\ b = \{v_3\}$$

The EPU e_2 receives and processes $(v_4, 7, 2)$: It updates its timestamp to 7 and executes $\mathit{handle}(v_4)$. The execution of $\mathit{cmd}(v_4) = c_{fwd}$ runs $\mathit{send}((x, 7, 2))$ and issues v_5 as next command:

$$e_2 \colon t = 7,\ b = \emptyset$$

The sent message $(v_4, 9, 4)$ is dispatched from e_1 to e_2 and the message $(x, 7, 2)$ is dispatched from e_2 to the output.

The EPU e_2 receives $(v_4, 9, 4)$, but continues processing $(v_4, 7, 2)$: It executes the next command $\mathit{handle}(v_5)$. The execution of $\mathit{cmd}(v_5) = c_{fwd}$ runs $\mathit{send}((y, 7, 2))$:

$$e_2 \colon t = 7,\ b = \emptyset$$

The message $(y, 7, 2)$ is dispatched from e_2 to the output.

The EPU e_2 processes $(v_4, 9, 4)$: It updates its timestamp to 9 and executes $\mathit{handle}(v_4)$. The execution of $\mathit{cmd}(v_4) = c_{\mathit{fwd}}$ runs $\mathit{send}((x, 9, 4))$ and issues v_5 as next command:

$$e_2\colon t = 9,\ b = \emptyset$$

The message $(x, 9, 4)$ is dispatched from e_2 to the output.

The EPU e_2 continues processing $(v_4, 9, 4)$: It executes $\mathit{handle}(v_5)$. The execution of $\mathit{cmd}(v_5) = c_{\mathit{fwd}}$ runs $\mathit{send}((y, 9, 4))$:

$$e_2\colon t = 9,\ b = \emptyset$$

The message $(y, 9, 4)$ is dispatched from e_2 to the output. ⌟

More complex executions of EPU networks are discussed after the introduction of the EPU commands for TeSSLa operators in the next sections: Section 5.5.1 demonstrates the execution of an EPU network generated from a simple TeSSLa specification and Section 5.6.1 one generated from a recursive specification.

5.3.3. Mapping Events to EPUs

The procedures *readMessage* and *writeMessage* connect the EPU network to an input and an output synchronised stream. A synchronised stream as defined in Definition 4.3 in Section 4.1.2 is a sequence of events. Every event consists of a timestamp and either a value or the symbol $\bot$. The symbol $\bot$ represents the absence of an event for every stream encoded in the synchronised stream. On the other hand, a message consists of a target command node or output node, a data value, and a timestamp.

In order to map between events to messages, we assume an implicit *EPU mapping* for a given EPU network. The *input mapping* maps streams, i. e. an index in the value tuple of a synchronised stream's event, to a command node (or an output node) of the EPU network. The *output mapping* maps an output node of the EPU network back to a stream, i. e. an index in the event's value tuple.

Every call to *readMessage* returns the events whose data values are not $\bot$ from the current event of the synchronised input stream. As timestamp, the timestamp of the current event is used. The data value is taken from the event, and the target is looked up in the input mapping. If no events are left, then the input stream is advanced to the next event of the input stream.

The procedure *writeMessage* collects messages with the same timestamp and appends an event to the synchronised output stream with all these data values filling all the missing values with $\perp$. The output mapping is used to determine the index of the data values in the event's tuple.

5.3.4. EPU Simulation

The semantics of the formal EPU model given in this section has been implemented in TypeScript. This simulation was used to test and debug the EPU commands and networks introduced in the next section. For that purpose the simulation can generate stream diagrams showing the messages passed through the EPUs. Such generated diagrams are shown in Sections 5.5.1 and 5.6.1.

In the simulation, the memory and the commands are identified using descriptive string keys. EPUs are described as configuration objects specifying for every command the name of the relevant memory section, the function implementing the EPU command and optional a name of the next, btsc or atsc command and a target identified by an EPU id and a command name. Additionally, every target contains human-readable a name, which is used for logging the messages. Functions implementing EPU commands have the following signature, which is as close as possible to Definition 5.1 (EPU Command) from Section 5.3:

```
(t: Time, u: Time, i: Value, m: Memory)
  => [Value, Memory, BlockingCounterDelta]
```

A `Value` can be any primitive type. The `Memory` is a lookup table that maps string keys to arbitrary values. The `BlockingCounterDelta` is either $-1, 0$ or 1. So for this simulation, an EPU's memory is organised as a two-staged lookup table where the first key identifies the memory region and the second key identifies the concrete memory cell in that region.

While the pseudocode given above assumes the EPUs and the dispatching of messages between the EPUs happening in parallel, the implemented simulation performs the dispatching of messages between EPUs, the feeding of input messages to the first EPU and the sequential execution of all EPUs in an alternating sequence. This alternating sequence simulates the concurrent execution defined by the pseudocode because an EPU can only produce at most one output message in a single execution step, which is then dispatched afterwards. Output messages are collected and converted into a synchronised stream to compare the generated output with the expected one in unit tests.

During the execution of the EPUs, every execution step of an EPU is sent to a central logger. Such a log message contains the current EPU id, the current cycle, the

received input, the send output, the executed command together with the information if the command was executed because of an incoming message, from a BTSC or ATSC schedule or as a next command, the current value of the blocking counter and the current value of the memory. These log messages are used to generate stream diagrams that visualise the execution of an EPU network, as shown in Figure 5.13 in Section 5.5.1 and Figure 5.17 in Section 5.6.1.

This simulation is by no means a simulation of the actual EPU hardware. It is only a simulation of the formal model of the hardware presented in this section. The relation between this formal model and the actual EPU hardware is discussed further in Section 5.7.

5.4. EPU Commands for TeSSLa Operations

In this section, we discuss translations for the TeSSLa operators to EPU commands. For a given TeSSLa specification, the corresponding EPU commands are given together with an EPU network describing their interconnection: It contains the initial memory assignments of the relevant parts of the memory tuple of that EPU and the network connections of the command node, i. e. its target, next command, btsc and atsc command. We use the EPU network diagrams introduced inw Section 5.3 with the following additional conventions: An EPU command is represented by the translated TeSSLa operator and the command's name written in cursive below the operator name. The thick grey bar represents a shared memory region, i. e. commands connected with this bar are accessing the same data and flag memory.

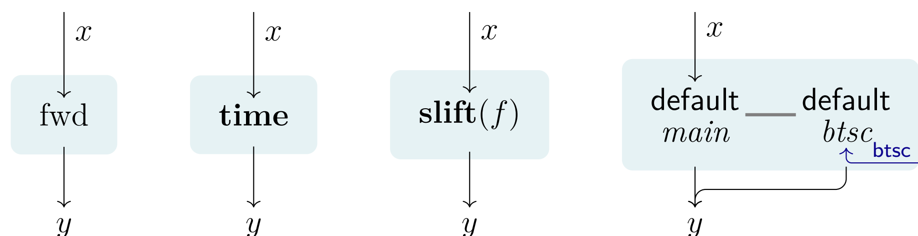


Figure 5.7.: EPU network diagrams for the forward command and the commands for **time**, the unary slift(f) and default.

We start with the identity function, which is usually not used on its own in a TeSSLa specification but is utilised to forward messages to the next EPU. See the next section on how EPU commands are mapped to the EPUs for an exemplary usage of this command. It simply forwards every incoming message unprocessed to the output.

Definition 5.6 (Forward Command)**.** The TeSSLa specification

$$y = \mathsf{lift}(id)(x)$$

is translated into the EPU command c given by:

$$c\colon o = i$$

The corresponding EPU network diagram is given in Figure 5.7. ⌟

The command for **time** is similar to the forward command but uses the messages timestamp instead of its value.

Definition 5.7 (Command for **time**)**.** The TeSSLa specification

$$y = \textbf{time}(x)$$

is translated into the EPU command c given by:

$$c\colon o = t$$

The corresponding EPU network diagram is given in Figure 5.7. ⌟

The command for unary arithmetic operations is the last of these simple commands with a single input. It applies the function f to every input value.

Definition 5.8 (Commands for $\mathsf{slift}(f)$ With a Unary Arithmetic Operation f)**.** The TeSSLa specification

$$y = \mathsf{slift}(f)(x)$$

is translated into the EPU command c given by:

$$c\colon o = f(i)$$

The corresponding EPU network diagram is given in Figure 5.7. ⌟

The binary $\mathsf{const}(d, r)$ can be translated as a special case of then unary $\mathsf{slift}(f)(r)$ with f being a constant function mapping all inputs to d.

Translating $\mathsf{default}$ requires two commands: c_{main} forwards incoming messages and sets the flag m_{sent}. c_{btsc} is initially scheduled for BTSC execution and sends the default value if m_{sent} is not set. The flag m_{sent} ensures that the default value is not send if the stream x has an event at timestamp 0.

Definition 5.9 (Commands for default)**.** The TeSSLa specification

$$y = \mathsf{default}(x, v)$$

is translated into the EPU commands c_{main} and c_{btsc} given by:

$$c_{main}\colon o = i,\ m'_{sent} = \mathit{true}$$
$$c_{btsc}\colon \text{if } \neg m_{sent} \text{ then } o = m_{value}$$

The EPU's memory tuple is initialised as follows:

$$\boldsymbol{m} = (m_{sent}, m_{value}) = (\mathit{false}, v).$$

The corresponding EPU network diagram is given in Figure 5.7. ⌟

The unary $\mathsf{const}(d)$ is translated as a special case of $\mathsf{default}(\mathsf{nil}, d)$. Remember that $\mathsf{const}(d)$ is the implicit conversion for constant values into streams. The empty stream nil is translated simply by never sending a message to that input.

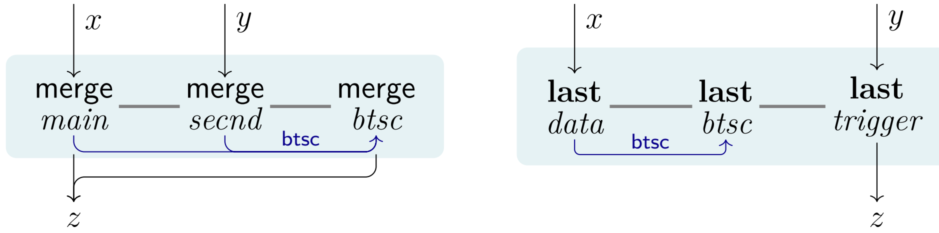


Figure 5.8.: EPU network diagrams for the commands for merge and **last**.

merge is the first actual binary operator. It requires three commands: c_{main} simply forwards the message because the left operand of merge is prioritised if both streams contain an event with the same timestamp. c_{secnd} stores the data in the memory and schedules c_{btsc} for BTSC execution, which sends the value if main was not executed for that timestamp.

Definition 5.10 (Commands for merge)**.** The TeSSLa specification

$$z = \mathsf{merge}(x, y)$$

is translated into the EPU command c_{main}, c_{secnd} and c_{btsc} given by:

$$c_{main}\colon m'_{sent} = \mathit{true},\ o = i$$
$$c_{secnd}\colon m'_{value} = i$$
$$c_{btsc}\colon m'_{sent} = \mathit{false},\ \text{if } \neg m_{sent} \text{ then } o = m_{value}$$

The EPU's memory tuple is initialised as follows:

$$\boldsymbol{m} = (m_{sent}, m_{value}) = (false, 0).$$

The corresponding EPU network diagram is given in Figure 5.8. ⌟

The commands for **last** store the last known value of the data stream (c_{data}) and send it out for every event on the trigger stream ($c_{trigger}$). They ensure that always the last known value is generated and never the current value even if both streams contain an event at the same timestamp. In that case the events might occur in an arbitrary order. Hence, c_{data} stores the new value in $m_{current}$ and schedules c_{btsc} for BTSC execution, which then writes $m_{current}$ to m_{last}.

Definition 5.11 (Commands for **last**)**.** The TeSSLa specification

$$z = \textbf{last}(x, y)$$

is translated into the EPU commands $c_{data}, c_{trigger}$ and c_{btsc} given by:

$$\begin{aligned} c_{data} &: m'_{current} = i \\ c_{trigger} &: o = m_{last} \\ c_{btsc} &: m'_{last} = m_{current} \end{aligned}$$

The EPU's memory tuple is initialised as follows:

$$\boldsymbol{m} = (m_{current}, m_{last}) = (0, \bot).$$

The corresponding EPU network diagram is given in Figure 5.8. ⌟

Note that the definition above uses the additional value $\bot$ for the memory entry m_{last}. Actually, this is syntactic sugar for two separate entries in the memory tuple: A value and a flag indicating if the value is present or $\bot$.

The commands for the binary $\mathsf{slift}(f)$ store the data values of the incoming messages (c_{store1} and c_{store2}) and schedules c_{btsc} for BTSC execution to apply f on the stored values and send out the result. Note how the two memory entries m_1 and m_2 fulfil the same purpose as in the case of the **last**. The main difference is that events at the current timestamp are already considered for the event with that timestamp. The flag m_{send} ensures that only one message is sent out per timestamp even if both streams have an event at the same timestamp, which enqueues c_{btsc} twice.

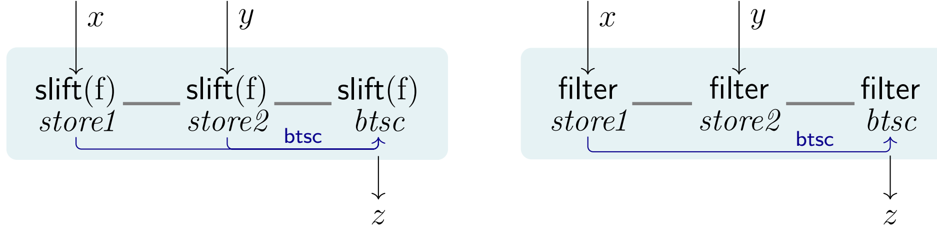


Figure 5.9.: EPU network diagrams for the commands for the binary slift(f) and filter.

Definition 5.12 (Commands for slift(f) With a Binary Arithmetic Operation f)**.** The TeSSLa specification

$$z = \mathsf{slift}(f)(x, y)$$

is translated into the EPU command c_{store1}, c_{store2} and c_{btsc} given by:

$$\begin{aligned} c_{store1} &: m_1' = i,\ m_{send} = true \\ c_{store2} &: m_2' = i,\ m_{send} = true \\ c_{btsc} &: m_{send}' = false,\ \text{if } m_1 \neq \bot \wedge m_2 \neq \bot \wedge m_{send} \text{ then } o = f(m_1, m_2) \end{aligned}$$

The EPU's memory tuple is initialised as follows:

$$\boldsymbol{m} = (m_1, m_2, m_{send}) = (\bot, \bot, false).$$

The corresponding EPU network diagram is given in Figure 5.9. ⌟

If the slift is used with a ternary function or one with even higher arity, we utilise Lemma 3.50 (Associativity of slift) from Section 3.4.1 to split it into multiple binary operations that can be translated as defined above. The ternary operator *ite* is treated as a special case in Definition 5.14 below.

filter is translated similar to slift. The main difference between the EPU commands for slift and filter is c_{store2} not enqueuing c_{btsc}. Instead of applying an arithmetic function to the stored values c_{btsc} checks the last known condition stored in m_2 and sends out the event only if that is a true value. Note that c_{store1} cannot send out messages directly because if both streams have an event at the same timestamp, they might arrive in arbitrary order.

Definition 5.13 (Commands for filter)**.** The TeSSLa specification

$$z = \mathsf{filter}(x, y)$$

is translated into the EPU command c_{store1}, c_{store2} and c_{btsc} given by:

$$c_{store1}: m_1' = i$$
$$c_{store2}: m_2' = i$$
$$c_{btsc}: \text{if } m_2 \neq 0 \text{ then } o = m_1$$

The EPU's memory tuple is initialised as follows:

$$\boldsymbol{m} = (m_1, m_2) = (0, 0).$$

The corresponding EPU network diagram is given in Figure 5.9. ⌟

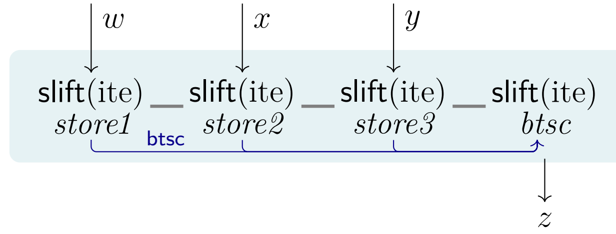


Figure 5.10.: EPU network diagram for the command for slift(ite).

The commands for slift(ite) are very similar to the translation for the binary slift. Instead of two there are now three storing commands c_{store1}, c_{store2} and c_{store3}, each scheduling c_{btsc} for BTSC execution. c_{btsc} sends out the value of m_2 or m_3 depending on m_1 being true. The commands realise the signal view of slift: No message is send until all streams are initialised, i. e. all memory entries m_1, m_2 and m_3 are no longer $\bot$.

Definition 5.14 (Commands for slift(ite))**.** The TeSSLa specification

$$z = \mathsf{slift}(ite)(w, x, y)$$

is translated into the EPU command $c_{store1}, c_{store2}, c_{store3}$ and c_{btsc} given by:

$$c_{store1}: m_1' = i,\ m_{send} = true$$
$$c_{store2}: m_2' = i,\ m_{send} = true$$
$$c_{store3}: m_3' = i,\ m_{send} = true$$
$$c_{btsc}: m_{send}' = false$$
$$\text{if } m_1 \neq \bot \wedge m_2 \neq \bot \wedge m_3 \neq \bot \wedge m_{send} \text{ then}$$
$$o = \begin{cases} m_2 & \text{if } m_1 \neq 0, \\ m_3 & \text{otherwise.} \end{cases}$$

The EPU's memory tuple is initialised as follows:

$$\boldsymbol{m} = (m_1, m_2, m_3, m_{send}) = (\bot, \bot, \bot, \mathit{false}).$$

The corresponding EPU network diagram is given in Figure 5.10. ⌟

5.5. Mapping the Dependency Graph on an EPU Network

We will now discuss how to translate an entire TeSSLa specification into an EPU network. In this section, we will only consider specifications with an acyclic dependency graph. Recursive specifications will be considered separately in Section 5.6.

Let φ be an acyclic TeSSLa specification. Organise the dependency graph in layers such that every node is a different layer than all the nodes on which it depends. These layers will be the EPUs of the EPU network. Translate every node on the dependency graph into the EPU commands given in Section 5.4 and put the command nodes on the EPU corresponding to their layer.

If multiple nodes in the dependency graph depend on the same node, then that node's corresponding commands must send their outgoing messages to multiple targets. Because the EPU network does not directly support this, we introduce additional commands depending on where the target node is located:

- If a stream is needed as input of multiple commands located on the same EPU, then send the message to the first of these commands and chain the other commands using the next command mechanism.
- If a stream is needed as input of multiple commands located on different EPUs, then insert forward command nodes on the first EPU, which forwards their input to the commands located on later EPUs. These forwards commands can then be chained together using the next command mechanism as in the first case.

5.5.1. Example

As an example of this translation we consider a TeSSLa specification with the input streams $x, y \in \mathcal{S}_{\mathbb{Z}}$ and the derived output stream $z \in \mathcal{S}_{\mathbb{Z}}$:

$$z = x > y \,?\, x - y : y - x.$$

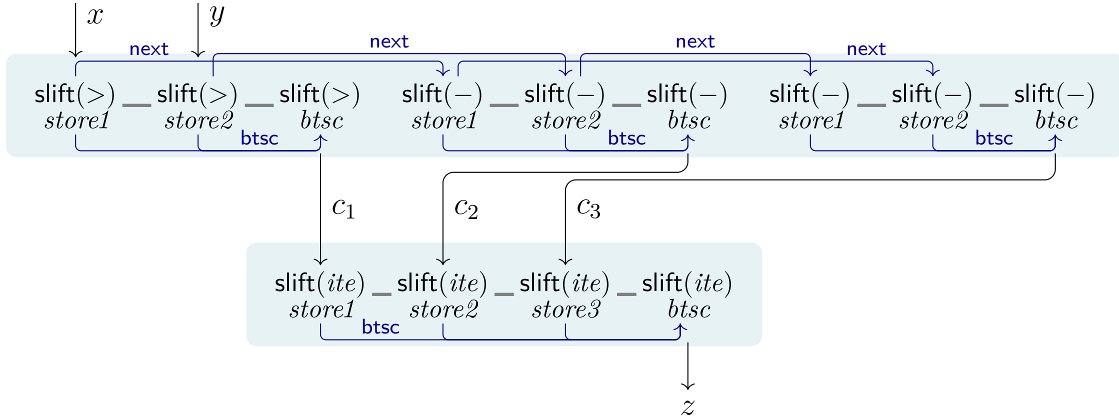


Figure 5.11.: EPU network diagram for the absolute value example.

With the implicit conversion from Section 3.3.7 the given function on the data domain is lifted to streams using slift. For the translation into an EPU network, we utilise Lemma 3.50 (Associativity of slift) from Section 3.4.1 which allows us to apply the slift to all operators individually instead of lifting the entire function at once. We can only translate an slift with an operation that the ALU supports. Nested operations on values must be converted into nested expressions on streams. We introduce some additional derived variables used to identify sub-expressions for the rest of this example and end up with the following specification:

$$\begin{aligned}
c_1 &= \mathsf{slift}(>)(x, y) \\
c_2 &= \mathsf{slift}(-)(x, y) \\
c_3 &= \mathsf{slift}(-)(y, x) \\
z &= \mathsf{slift}(ite)(c_1, c_2, c_3)
\end{aligned}$$

Figure 5.11 shows the EPU network diagram for this specification.

This specification is a rather artificial example to explain how the commands are nested. The ALU of the EPUs has a builtin absolute value function which could be used to simplify this expression to $abs(x - y)$.

Figure 5.12 shows an exemplary evaluation of this absolute value specification using the synchronous semantics from Section 4.1. There is no output for the first event on x because the stream y has not been initialised yet. With the first event on y the current value on y is larger than on x. Hence the condition c_1 becomes false, and the output becomes the value on c_3, a positive 1. With another value on x this turns the other way round: The output becomes the value on c_2, which is now the positive value. Finally, we have two simultaneous events on x and y, resulting in the third and final event on z.

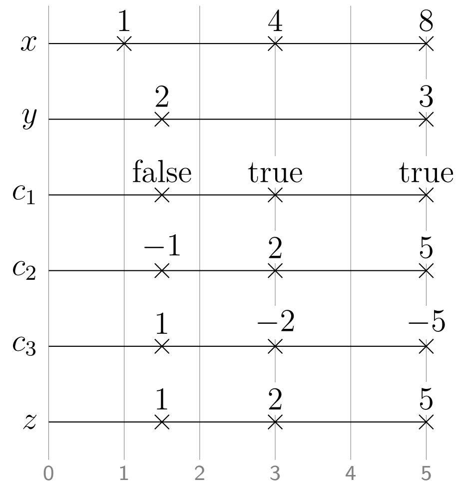


Figure 5.12.: Exemplary evaluation of the absolute value example using the synchronous semantics.

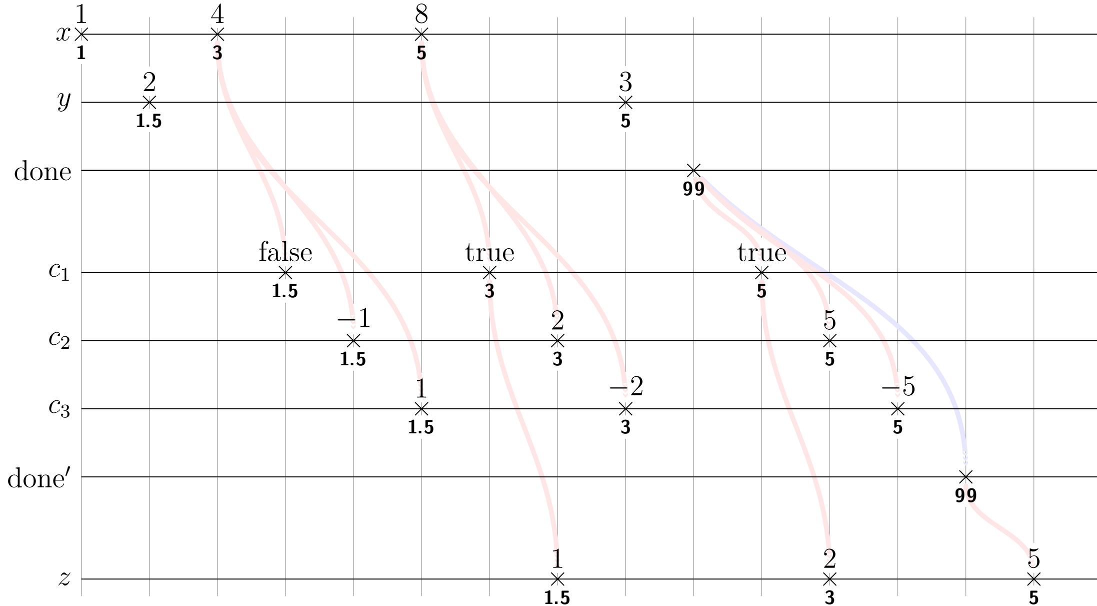


Figure 5.13.: Exemplary evaluation of the EPU network derived from the absolute value example using the same input as in Figure 5.12.

Figure 5.13 shows the evaluation of the EPU network translated from the absolute value TeSSLa specification. The same input as in Figure 5.12 is used. This stream diagram is based on the EPU network execution semantics given above and was generated using the EPU simulation discussed in Section 5.3.4. In particular these diagrams adhere the following rules:

- An event's data value is depicted above the cross. The event's logical timestamp is depicted below the cross.
- Events are always depicted when the EPU first reads them.
- Direct causal relations between events are drawn in blue. Events sent because of the timestamp increase caused by an event are drawn in red.
- The X-axis represents physical time passing by during the execution of the EPU network. However, the axis has no linear relation to any measured time. Instead, it represents causal relations: Events can only be processed after they have been sent.
- An EPU can only read and send one event simultaneously. The diagram does not depict the execution of commands but receiving and sending events. In particular, commands that neither read nor write an event do not consume time in this representation.
- As discussed in the execution semantics of the EPU network, the queues between the EPUs are of size one. If an EPU cannot process its input fast enough, this causes back pressure along the pipeline.

Note the additional streams done and done′ which are not depicted in the EPU network diagram in Figure 5.13. After the input stream was read entirely, an additional message with a maximal timestamp was sent to an empty command on the last EPU. The stream done represents this message being forwarded by the first EPU because its target address does not match. The stream done′ represents this message being received by the second EPU. This empty command itself does nothing on its own, but the maximal timestamp of this message triggers a timestamp increase on all the EPUs the message passes through. As one can see in Figure 5.13 this final flush of the EPU pipeline is needed to execute all commands scheduled for BTSC execution.

5.6. Recursion

So far, we have only considered acyclic TeSSLa specifications, but to gain the full expressiveness of TeSSLa, we now discuss the translation of specifications with cycles

in the dependency graph. We only consider well-formed timestamp-conservative specifications, i. e. every cycle is guarded by a **last** operator.

The main question addressed in this section is how to integrate cycles into the outer EPU pipeline. The current timestamp of an EPU must never decrease. This invariant is essential for the mechanics of the pipeline and the EPUs to work. While this is naturally the case if events are fed synchronously into the pipeline and only traverse down the pipeline, this invariant might break as soon as messages can travel the opposite direction up the pipeline.

We assume that every cycle is started with the **last** operator guarding it. This assumption can be easily fulfilled by reordering the nodes of the dependency graph, i. e. moving all other nodes involved in the cycle to a later EPU. The statement

$$y := \mathbf{last}(v, r)$$

defines the derived stream y: For every event on the trigger stream r, an event on y carries the value of the last event that happened on the value stream v before. If there was no event on the value stream yet, then the event on r is ignored.

Recall that the EPUs form a pipeline processing events. The different EPUs have different current timestamps, but the timestamps are increasing, so further along the pipeline we get, the lower the current timestamps are. In case of the non-recursive **last** in Definition 5.11 from Section 5.4 this invariant makes the implementation of the **last** semantics rather straight forward. Trigger events only need to consider events on the value stream which arrived earlier. We have to make sure that we only use values from events that happened before the trigger. This is done by the c_{btsc} command which copies $m_{current}$ into m_{last}.

In the case of cycles, we have to solve two additional problems:

(P1) We must ensure that the EPU has received enough information through the recursive loop to send out the data in case of a trigger event. We can no longer assume that all relevant data events arrived before the trigger event at the EPU because the data event arrives through the recursive channel, which is not implicitly synchronised with the other events. So now we have to keep track of every event going through the EPU, which can eventually end up in the recursive data input of the **last**. We need to keep track of whether we are waiting for such an event. We are again up to date if all such events have made it through the recursive cycle. Since the EPUs can filter events, we introduce progress messages to detect the absence of events: With every event going into the recursive cycle, we send a progress message following it. The progress message is sent with the next occurring timestamp, which guarantees that this message is received after the original message because the EPUs always keep the order of timestamps.

(P2) If an EPU waits for data arriving through the recursive channel, it cannot immediately send the message responding to the trigger. In those cases, we must ensure that this message still has the correct timestamp and that the invariant is preserved, that the timestamps of all messages sent out by an EPU never decrease. So if the **last** operator gets triggered and the required information about the value stream is not yet available, we must ensure that the EPU's current timestamp stays the same until the trigger is handled. Blocking the EPU ensures this. It prevents the entire EPU from accepting incoming messages, which increase the current timestamp. Note that such a block is not necessary for every message being sent in the recursive cycle but only if the **last** gets triggered without all required information about the cyclic dependent value stream being available yet. So it highly depends on the event frequency on the trigger stream if the EPU gets blocked or not.

With these considerations, we can now look at the following definition giving the EPU commands used to translate a **last** operator guarding a cycle. Because of (P2), this is called a blocking **last**.

Definition 5.15 (Commands for Blocking last)**.** The TeSSLa specification

$$z = \mathsf{last}(x, y)$$

in which x is (transitively) depending on z is translated into the EPU commands c_{obs}, c_{btsc}, c_{trig}, c_{data}, c_{prog} and c_{atsc} given by:

$$
\begin{aligned}
c_{obs}\colon\ & o = i\\
c_{btsc}\colon\ & \text{if } m_{time} < t \text{ then } m'_{time} = t,\ m'_{state} = \mathit{invalid}\\
c_{trig}\colon\ & \text{if } m_{state} = \mathit{valid} \text{ then}\\
& \qquad \text{if } m_{time} = t \wedge m_{prev} \neq \bot \text{ then } o = m_{prev},\ m'_{state} = \mathit{invalid}\\
& \qquad \text{if } m_{time} < t \wedge m_{value} \neq \bot \text{ then } o = m_{data},\ m'_{time} = t,\ m'_{state} = \mathit{invalid}\\
& \text{if } m_{state} = \mathit{invalid} \text{ then } m'_{state} = \mathit{triggered},\ b' = 1\\
c_{data}\colon\ & m'_{data} = i,\ m'_{prev} = m_{data},\\
& \text{if } m_{state} = \mathit{valid} \text{ then } m'_{time} = u\\
& \text{if } m_{state} = \mathit{invalid} \wedge m_{time} = u \text{ then } m'_{state} = \mathit{valid}\\
& \text{if } m_{state} = \mathit{triggered} \wedge m_{time} = u \text{ then}\\
& \qquad o = i,\ m'_{time} = t,\ m'_{state} = \mathit{invalid},\ b' = -1\\
c_{prog}\colon\ & \text{if } u > m_{time} \text{ then}\\
& \qquad \text{if } m_{state} = \mathit{invalid} \text{ then } m'_{state} = \mathit{valid}\\
& \qquad \text{if } m_{state} = \mathit{triggered} \text{ then}\\
& \qquad\qquad \text{if } m_{data} \neq \bot \text{ then}
\end{aligned}
$$

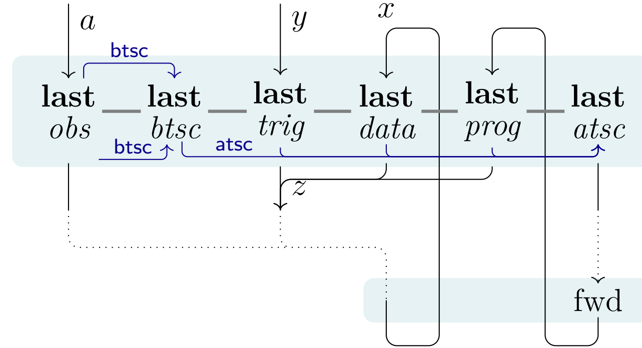


Figure 5.14.: EPU network diagrams for the commands for blocking **last**.

$$o = m_{data},\ m'_{time} = t,\ m'_{state} = invalid,\ b' = -1$$
$$\text{if } m_{data} = \bot \text{ then } m'_{state} = valid,\ b' = -1$$
$$c_{atsc}\colon \text{if } m_{state} = invalid \text{ then } o = 0$$

The EPU's memory tuple is initialised as follows:

$$\boldsymbol{m} = (m_{data}, m_{prev}, m_{time}, m_{state}) = (\bot, \bot, -\infty, valid).$$

The corresponding EPU network diagram is given in Figure 5.14. ⌟

There are, in general, three different states the system can be in: The memory field m_{state} is either *valid*, *invalid* or *triggered*. In the following we describe the effect of the three variables m_{data}, m_{time} and m_{prev} in these three states and which state transitions are triggered by receiving a *trigger*, *observing* an event going into the recursive cycle, and receiving a *data* value or a *progress* through the recursive cycle.

valid m_{data} contains the most recent data, which was last updated at c. m_{time} contains the timestamp at which we received this data. m_{prev} contains the previous data value.

Trigger. We can simply send out an event with the value m_{data}. If m_{data} is $\bot$, then the **last** has not been initialised, and we do not send out anything.

There is one rare special case: If We have received data with the same timestamp with which we are triggered now, then we cannot send out the current data. The **last** operator always produces the previous value, so if m_{time} equals the current timestamp, then we send out m_{prev} instead of m_{data}.

We have sent an event into the recursion in both cases, so we are now invalid.

Observe. We have to become invalid because we have to wait for this event to come back through the recursive cycle. Since the **last** operator always produces the last known value, an observed value does not immediately invalidate the data, but only for the next occurring timestamp. Hence, we perform the state change right before the timestamp increase.

Data. An observed event made it through the recursive cycle before the timestamp increased. We update m_{data} and m_{prev} and m_{time} accordingly and stay valid.

Progress. We can safely ignore this because we have already processed the data preceding this progress. Otherwise, the current state would not be valid.

invalid m_{data} contains the previous data value. m_{time} contains the latest timestamp at which an event was send into the recursion, which we are waiting for now. m_{prev} contains nothing meaningful.

Trigger. We cannot respond to this trigger. So we go to the triggered state and increment the blocking counter to prevent any timestamp increase until the trigger at the current timestamp could be reacted to.

Observe. We are already invalid, so at the timestamp increase, we update the timestamp we are waiting for in m_{time}.

Data / Progress. If this was the data we are waiting for, i. e. the received timestamp equals m_{time} or is greater, respectively, then we are now valid again. In case of *data* we update m_{data} and m_{prev} accordingly.

triggered The variables have the same meaning as in the invalid state. Additionally, we know in this state that we have postponed a trigger which must be executed as soon as we have valid data.

Trigger. We cannot be triggered in this state because a timestamp increase is blocked, and we have already been triggered. Two triggers cannot happen with the same timestamp because two events on the same stream always have different timestamps.

Observe. Timestamp increases are blocked in this state, so we cannot observe a timestamp increase in this state.

Data / Progress. If this was the data we are waiting for, i. e. the received timestamp equals m_{time} or is greater, respectively, then we can now react to the postponed trigger and send out the just received data. In case of *data* we update m_{data} and m_{prev} accordingly. Since we have sent a new event into the recursion, we become invalid again and update m_{time}. We decrement the blocking counter because there is no longer a postponed trigger, so a timestamp increase is allowed again.

5.6.1. Example

To illustrate the translation of a blocking **last** to an EPU network we consider the following TeSSLa specification with the free input variables $x, y, a \in \mathcal{S}_{\mathbb{D}}$:

$$
\begin{aligned}
z &= \mathsf{default}(\mathbf{last}(z, x) + y, 0) \\
b &= a + 1
\end{aligned}
$$

We split up the definition of z into multiple intermediate derived streams as follows:

$$
\begin{aligned}
\ell &= \mathbf{last}(z, x) \\
s &= \ell + y \\
z &= \mathsf{default}(s, 0) \\
b &= a + 1
\end{aligned}
$$

The EPU network translated from this specification is shown in Figure 5.15. The stream y is observed so that we know about all messages going into the recursive cycle. The messages on y are only forwarded to y' on the first EPU. The stream x triggers the **last** which then sends out a message on ℓ. Every event observed on y and every message sent out as a response to a trigger on x schedules a command for ATSC execution, sending a progress message after the timestamp is increased. The events on y' and ℓ are added together on the second EPU following the signal view. The third EPU applies a default value in order to get the recursion started, and the fourth EPU forwards the resulting events on z back to the first EPU as input to the **last** guarding the recursion and a copy of that message out as the result of the recursive computation. Progress messages are forwarded back to the first EPU through the recursive channel. Mostly not interfering with all this – at least while the EPU is not blocked – for every event on a the message's value is incremented on the first EPU and forwarded on all subsequent EPUs. This stream is included in the example to demonstrate the interference – or its absence – of regular messages with the recursive messages.

Figure 5.16 shows an exemplary evaluation of this specification using the synchronous semantics from Section 4.1. The $\mathsf{default}$ first creates a 0 on z. Then an event on a is processed, i. e. its value gets incremented. Next, we receive a trigger on x causing the **last** to produce the last known value on z, which is the 0 created earlier by the $\mathsf{default}$. Then we receive a 4 on y, which is added to this 0 causing a new event on z. Next, we receive two synchronous events on x and y, so the 4 produced on z earlier is reproduced with the current timestamp by the **last** and added to the 2 on y producing a 6 on z. Finally, there is another event on a causing a final event on b.

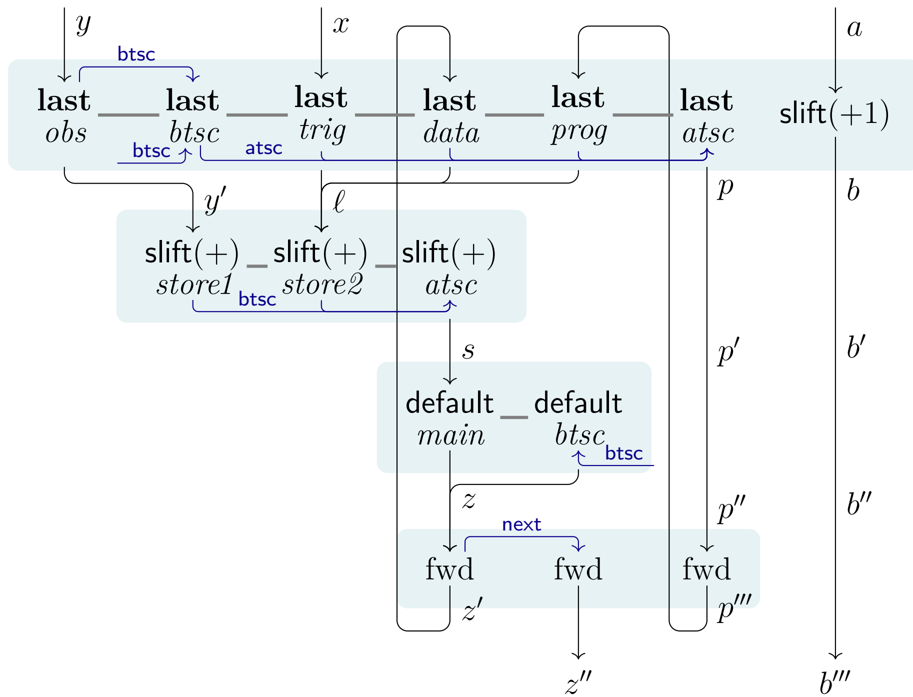


Figure 5.15.: EPU network diagram for the recursive example.

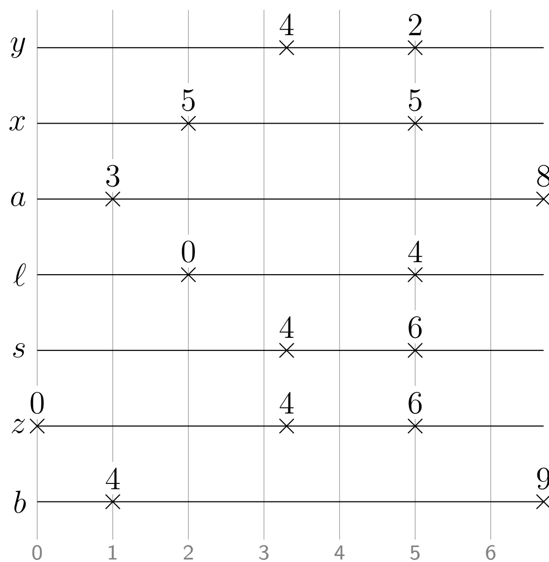


Figure 5.16.: Exemplary evaluation of the recursive example using the synchronous semantics.

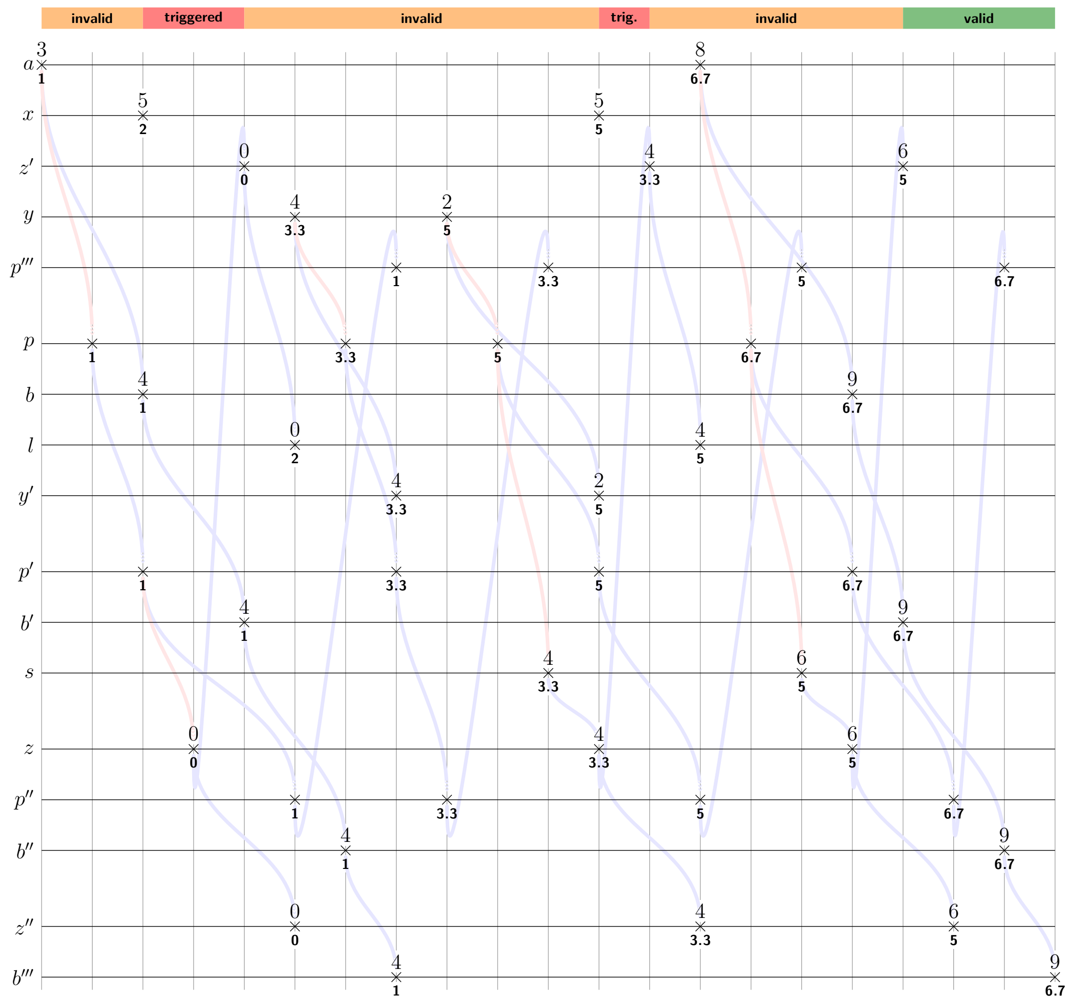


Figure 5.17.: Exemplary evaluation of the EPU network derived from the recursive example using the same input as in Figure 5.16.

Figure 5.17 shows the same input being processed by the EPU network. Again this stream diagram is based on the EPU network execution semantics given above and was generated using the EPU simulation discussed in Section 5.3.4. In addition to EPU stream diagrams used earlier in this chapter in this diagram the state of the commands for the blocking **last** is indicated by the bar above the first EPU: If the state is triggered, the EPU is blocked, i. e. the blocking counter is incremented above zero, and the EPU no longer accepts incoming messages, increasing the current timestamp. Note how in both cases, an incoming recursive message delivers the missing data we were waiting for. Then the state goes back to invalid, the blocking counter is decremented, and the blocking is disabled.

5.6.2. Expressiveness

Theorem 5.16. *Every timestamp-conservative TeSSLa specification can be evaluated on EPUs.*

Proof. As a direct consequence of Theorem 3.51 (Signal Lift and Default) from Section 3.4.2 and Lemma 3.98 (Expressiveness of Timestamp Conservative TeSSLa) from Section 3.6 we get: The basic operators slift, **last**, **time**, default and **unit** are sufficient to express every timestamp-conservative TeSSLa specification. In the previous sections we have shown how to evaluate TeSSLa specifications build from these operators on EPUs. □

Note that the proof of Theorem 3.51 requires data types being extended with $\perp$. This extension can be encoded into integer data types by interpreting an unused value as $\perp$. However, this is a rather theoretical result because this encoding and especially the constructions used in the proof are very inefficient on the EPUs because they use up too many commands. For that reason, additional translations for derived operators into EPU networks have been given in Section 5.4.

5.7. Fulfilling Hardware Restrictions

The last sections discussed how to translate TeSSLa specifications into EPU networks which were introduced as formal EPU model in Section 5.3. This section will discuss additional restrictions and limitations of the actual hardware that must be considered when encoding such an EPU network into actual hardware EPUs.

A central concept of the hardware EPUs is the encoding of conditions. All conditional assignments and outputs of an EPU are expressed in the form of a condition.

These conditions are configured in a separate condition configuration and are referred to only by their ID in the command's configuration.

The following constraints on EPU commands must be fulfilled on actual hardware:

- A command must use at most 3 data memory values, which can be read and written.
- A command must use at most 8 flag memory values, i. e. Boolean values, which can be read and written. These flags must be grouped into flag words of 8 bit, i. e. flags cannot be shared with other commands in different subsets.
- A command must use only one arithmetic computation.
- A command can only use one condition to decide whether to send out an event or not. The value of an outgoing message sent by a command can be taken from many sources, e. g. the memory or an arithmetic computation. However, this choice must be made static at compile-time, and the only decision made at runtime is whether to send a message or not. Note that this restriction does not limit the possible values of an outgoing message.
- A command can only use one condition to decide whether to update a memory cell or leave it unchanged.
- A command can only have three different new values for the entire flag word, with one option being leaving the flag word unchanged.
- The conditions must be expressible; see below on how conditions are encoded.

In addition to these constraints on the individual EPU commands, the size of the memories and FIFOs are further limitations. In detail, the following limitations of the EPUs must be considered when assigning EPU commands to EPUs:

– The size of the ASIC and BSIC FIFOs,
– the size of the command memory,
– the size of the data and flag memory and
– the number of available channels.

In the next sections, we will discuss how to fulfil the constraints regarding the individual EPU commands, how to encode conditions and how to fulfil the memory constraints of the EPUs.

5.7.1. Splitting Up EPU Commands

In order to fulfil the constraints regarding the individual EPU commands, some EPU commands must be split up into multiple EPU commands: A single EPU command can be replaced with two (or more) EPU commands linked together using the next command. The original input message is available to all EPU commands. Memory adjustments of previous commands are already available when executing the next commands, so basically, an EPU command can be split up into two consecutive commands at any point.

Example 5.17 (Splitting Up EPU Commands)**.** As an example on how to split up EPU commands we reconsider the command c_{btsc} used in Definition 5.14 for the translation of the TeSSLa specification

$$z = \mathsf{slift}(ite)(w, x, y).$$

The command c_{btsc} is given by

$$
\begin{aligned}
c_{btsc} \colon\ & m'_{send} = \mathit{false} \\
& \text{if } m_1 \neq \bot \wedge m_2 \neq \bot \wedge m_3 \neq \bot \wedge m_{send} \text{ then} \\
& \qquad o = \begin{cases} m_1 & \text{if } m_1 \neq 0, \\ m_2 & \text{otherwise.} \end{cases}
\end{aligned}
$$

There are three possible outcomes regarding the outgoing message o: Sending out m_1, sending out m_2 or doing nothing. According to the constraints above we must decide statically where the value should come from and the ALU is already busy deciding $m_1 \neq 0$, so we need to split this command up into c_{btsc1} and c_{btsc2} defined as follows:

$$
\begin{aligned}
c_{btsc1} &: \text{if } m_1 \neq \bot \wedge m_2 \neq \bot \wedge m_3 \neq \bot \wedge m_{send} \wedge m_1 \neq 0 \text{ then } o = m_2 \\
c_{btsc2} &: \text{if } m_1 \neq \bot \wedge m_2 \neq \bot \wedge m_3 \neq \bot \wedge m_{send} \wedge m_1 = 0 \text{ then } o = m_3 \\
& \quad m'_{send} = \mathit{false}
\end{aligned}
$$

The precondition of m_1, m_2 and m_3 being initialised and m_{send} being set is now checked by both commands. What to send is decided by $m_1 \neq 0$ or its negation $m_1 = 0$ in the second command. ⌟

5.7.2. Condition Configuration

As mentioned in the constraints above, all conditional assignments and outputs of an EPU must be expressible as a condition configuration. The result flag of the

ALU and all the flags of the flag memory are available as inputs to the condition logic. The output of the condition logic is used to decide if an output event is sent or memory is updated. The condition logic realises an expression of the form

$$((x \otimes x) \otimes (x \otimes x)) \otimes ((x \otimes x) \otimes (x \otimes x)),$$

where each $\otimes$ is either $\wedge$ or $\vee$ and each x is any of the flags, optionally negated.

For simplicity, we discuss the condition configuration for condition logic trees of depth 3. The EPUs on the actual hardware are equipped with trees of depth 4. Note that this only changes the number of different formulas that can be expressed, not the number of available inputs. In both cases, all nine flags can be used as an input for a condition. Figure 5.18 shows a schematic representation of a condition logic tree of depth 3 and the variables used to configure the tree.

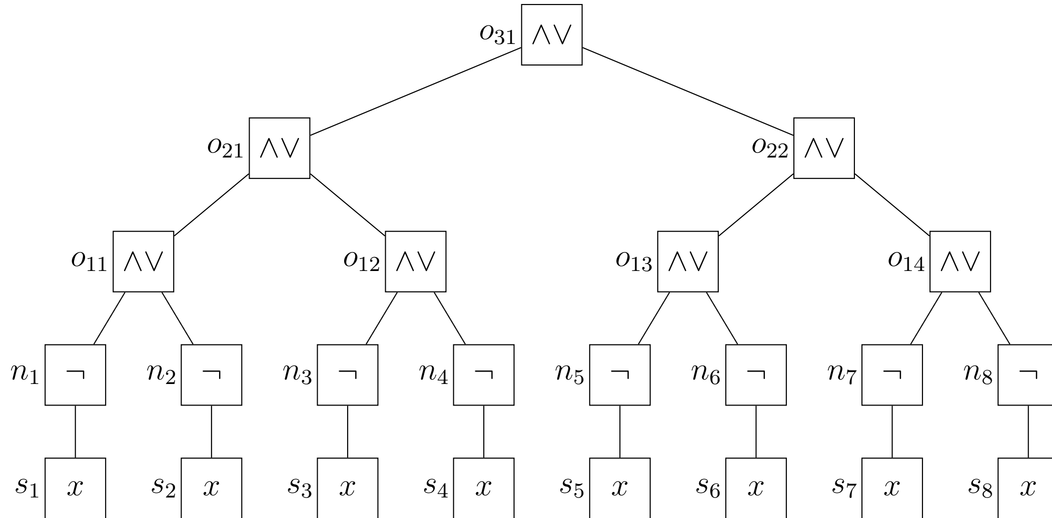


Figure 5.18.: Schematic representation of an condition logic tree of depth 3.

The tuple $\boldsymbol{o} = (o_{31}, o_{21}, o_{22}, o_{11}, o_{12}, o_{13}, o_{14}) \in \mathbb{B}^7$ specifies if the Boolean operator is $\wedge$ (false) or $\vee$ (true). The tuple $\boldsymbol{n} = (n_1, n_2, \dots, n_8) \in \mathbb{B}^8$ specifies if the negation is enabled (true) or not (false). The tuple $\boldsymbol{s} = (s_1, s_2, \dots, s_8) \in \{1, 2, \dots, 9\}^8$ specifies which of the 9 flags (flag word and ALU flag) are assigned to the inputs.

We will discuss two different approaches to fit Boolean functions $f\colon \mathbb{B}^9 \to \mathbb{B}$ into a configuration $\boldsymbol{o}, \boldsymbol{n}, \boldsymbol{s}$ for this condition tree: Using an SMT solver and a faster and simpler greedy approach.

Using an SMT Solver

Let $f\colon \mathbb{B}^9 \to \mathbb{B}$ be the given function with the arguments $\boldsymbol{x} = x_1, x_2, \ldots, x_9$. Then we want to find an assignment to the variable tuples $\boldsymbol{o}, \boldsymbol{n}, \boldsymbol{s}$ such that

$$\forall\, \boldsymbol{x} \in \mathbb{B}^9\colon\ f(\boldsymbol{x}) = \gamma$$

with

$$\begin{aligned}\gamma := \alpha(o_{31}, \alpha(o_{21}, \alpha(o_{11}, \beta(n_1, s_1), \beta(n_2, s_2)),\\ \alpha(o_{12}, \beta(n_3, s_3), \beta(n_4, s_4))),\\ \alpha(o_{22}, \alpha(o_{13}, \beta(n_5, s_5), \beta(n_6, s_6)),\\ \alpha(o_{14}, \beta(n_7, s_7), \beta(n_7, s_7))))\end{aligned}$$

and

$$\begin{aligned}\alpha(o_{ij}, a, b) &:= (\neg o_{ij} \wedge a \wedge b) \vee (o_{ij} \wedge a) \vee (o_{ij} \wedge b)\\ \beta(n_i, s_i) &:= n_i \veebar \left(\bigvee_{k \in \{1,2,\ldots,9\}} s_i = k \wedge x_k \right)\end{aligned}$$

The operator $\veebar$ denotes the exclusive disjunction, i. e. the XOR operation.

These conditions can be encoded into and solved with an SMT solver whose output is an assignment of the configuration variables of the condition logic tree such that the logic formula represented by the tree is equivalent to the given formula.

Note that this is not an optimisation problem. Any solution is a good solution because we only need a valid configuration for the condition logic tree. This configuration is written into the memory and interpreted by the FPGA accordingly. Since we are not reconfiguring the FPGA itself, the path length of the routes on the FPGA is fixed and independent of the concrete configuration.

Example 5.18 (Computing a Condition Configuration Using an SMT Solver)**.** Consider the function

$$f(\boldsymbol{x}) := x_1 \vee \Big(x_2 \vee ((x_3 \wedge x_4) \vee \neg(x_3 \vee x_4))\Big),$$

which is equivalent to $x_0 \vee x_1 \vee (x_2 \leftrightarrow x_3)$. If we insert this into the SMT formula presented above a possible solution would be

$$\begin{aligned}\boldsymbol{o} &= (\mathit{false}, \mathit{true}, \mathit{true}, \mathit{true}, \mathit{true}, \mathit{true}, \mathit{true})\\ \boldsymbol{n} &= (\mathit{false}, \mathit{true}, \mathit{false}, \mathit{false}, \mathit{false}, \mathit{false}, \mathit{false}, \mathit{true})\\ \boldsymbol{s} &= (3, 4, 1, 2, 4, 2, 1, 3)\end{aligned}$$

which represents this formula

$$f_1(\boldsymbol{x}) := \big((x_3 \vee \neg x_4) \vee (x_1 \vee x_2)\big) \wedge \big((x_4 \vee x_2) \vee (x_1 \vee \neg x_3)\big).$$

The condition tree is shown in Figure 5.19. ⌟

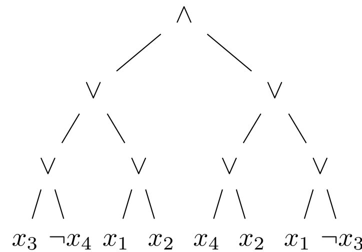


Figure 5.19.: Condition tree for the condition configuration discussed in Example 5.18.

Using a Simple Greedy Algorithm

In many cases, the following simple algorithm is sufficient, faster and produces a more human-readable result. The condition tree shown in Figure 5.18 can already express Boolean formulas, including disjunction, conjunction and negation of the leaves. So it only remains to convert given formulas to negation normal form and balance them to fit in trees of limited depth. While the following steps are not guaranteed to find a solution if one exists, they are sufficient enough for all practical examples:

1. Replace constants with tautologies or contradictions, i. e. replace true with $x_1 \vee \neg x_1$ and replace false with $x_1 \wedge \neg x_1$.
2. Convert the formula to negation normal form.
3. Balance the formula: Form maximal subformulas using only the same operator ($\wedge$ or $\vee$) starting with the tree's leaves, i. e. the literals in the formula. Build balanced trees for those subformulas. Now consider these subformulas as fixed and iterate the process until the tree is balanced. Always consider the depth of subtrees when balancing subformulas over already balances subtrees.
4. Fill up the tree: If balancing the tree does not produce a fully balanced tree, fill up the tree either by duplicating subtrees or adding tautologies.

5. Assert depth: If the tree's depth is too high (more than three levels of operators), this algorithm cannot fit the given function. If it is too small (less than three levels), add layers either by duplicating subtrees or adding tautologies.

Example 5.19 (Computing a Condition Configuration Using a Simple Greedy Algorithm)**.** Consider the same function used in the last example:

$$f(\boldsymbol{x}) := x_1 \vee \Big(x_2 \vee ((x_3 \wedge x_4) \vee \neg(x_3 \vee x_4))\Big).$$

The expression does not contain any constant. We first translate the expression into negation normal form:

$$f_1(\boldsymbol{x}) := x_1 \vee \Big(x_2 \vee ((x_3 \wedge x_4) \vee (\neg x_3 \wedge \neg x_4))\Big).$$

Next we identify subexpressions using the same operator: The two conjunctions at the bottom of the tree are already perfectly balanced, but the disjunction at the top of the tree can be better balanced:

$$f_2(\boldsymbol{x}) := (x_1 \vee x_2) \vee \Big((x_3 \wedge x_4) \vee (\neg x_3 \wedge \neg x_4)\Big).$$

As a final step we add a conjunction with the tautology $\neg x_1 \vee x_1$ to the subtree on the left in order to perfectly balance the tree:

$$f_3(\boldsymbol{x}) := \Big((\neg x_1 \vee x_1) \wedge (x_1 \vee x_2)\Big) \vee \Big((x_3 \wedge x_4) \vee (\neg x_3 \wedge \neg x_4)\Big).$$

This formula now fits exactly into the condition tree and we get the following condition configuration:

$$\begin{aligned}
\boldsymbol{o} &= (\mathit{true}, \mathit{false}, \mathit{true}, \mathit{true}, \mathit{true}, \mathit{false}, \mathit{false}) \\
\boldsymbol{n} &= (\mathit{true}, \mathit{false}, \mathit{false}, \mathit{false}, \mathit{false}, \mathit{false}, \mathit{true}, \mathit{true}) \\
\boldsymbol{s} &= (1, 1, 1, 2, 3, 4, 3, 4)
\end{aligned}$$

The tree representations for these functions is shown in Figure 5.20. ⌟

5.7.3. Placement of EPU Commands in the Network

If the mapping results in an EPU network using more than the available memory for data, flags or FIFOS, some EPU commands must be moved to the next EPU following these considerations:

- One can only move entire command groups, i. e. a group of commands that shares the same data and flag memory, indicated as grey memory connections in the EPU network diagrams.

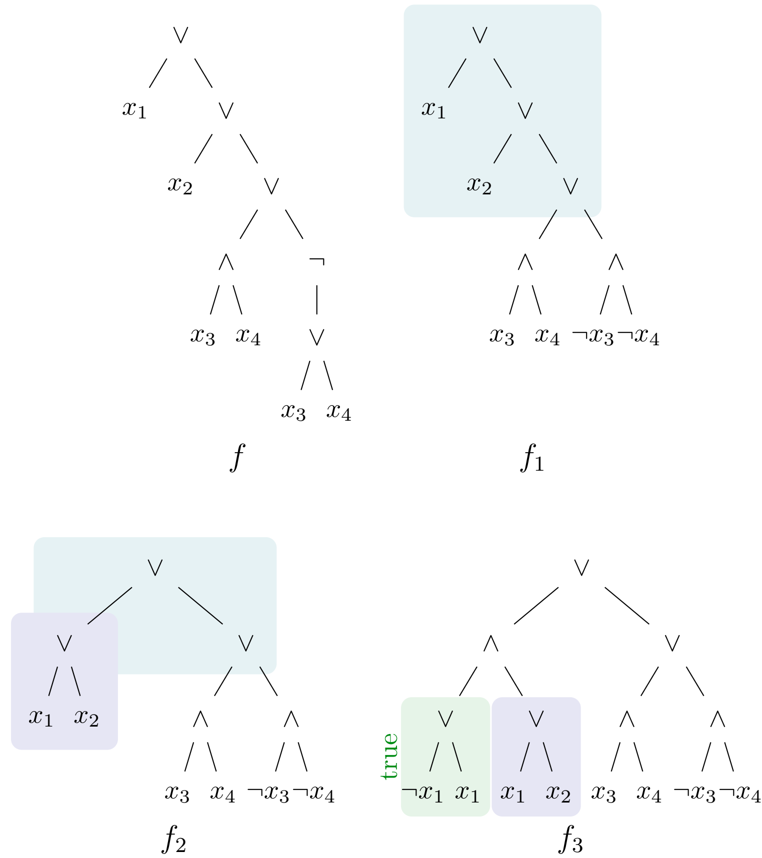


Figure 5.20.: Tree representations for the functions discussed in Example 5.19. In f_1 the maximal subtree with same operators is highlighed and in f_2 balanced. In f_2 additionally the not yet perfectly balanced subtree is highlighted and moved one level down in f_3.

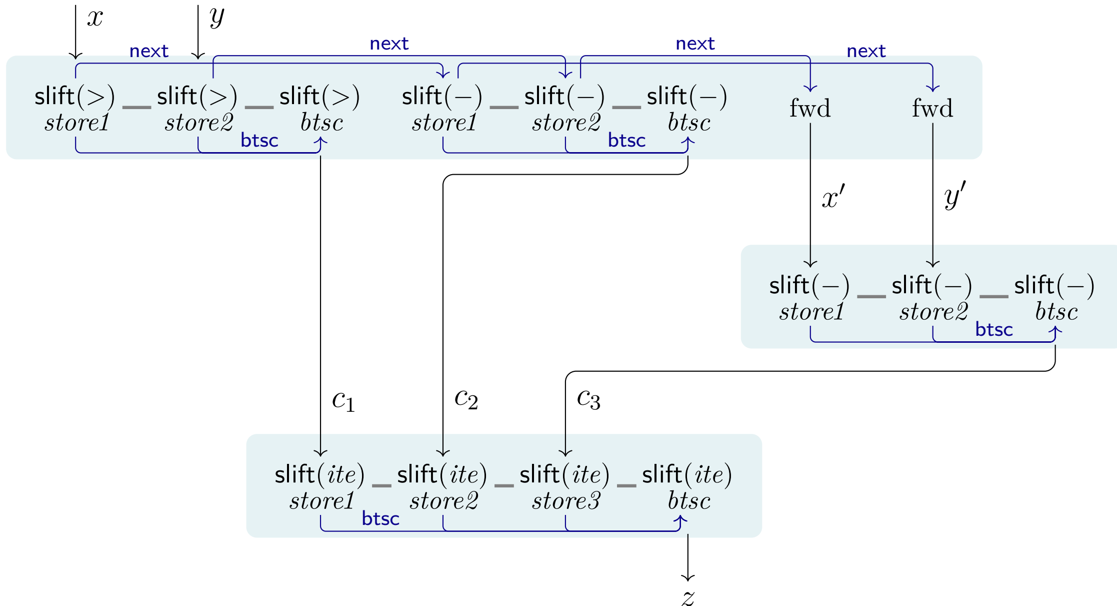


Figure 5.21.: Adjusted EPU network diagram for the absolute value example from Figure 5.11 in Section 5.5.1.

- Dependent command groups must be moved to later EPUs, too.
- If a command group is moved to a later EPU, which got its input via a next chain, then one might need to insert additional forward commands as discussed in Section 5.5.

As an example of such adjustments reconsider the absolute value example from Figure 5.11 in Section 5.5.1. For the sake of the example, we assume that the commands for the third **slift** do not fit on the first EPU. Figure 5.21 adjusts the EPU network diagram accordingly: The three commands for that **slift** are moved to the next EPU. As a consequence, the commands for the **slift**(ite) are also moved to a later EPU because it depends on the output of the **slift**. In the original EPU network, the inputs x and y are dispatched to the three **slift** using a next command chain. This only works for commands on the same EPU, so in order to send these messages to the third **slift**, too, they are dispatched to fwd commands using the next chain. These fwd commands forward the messages to the third **slift** on the second EPU. The second EPU bypasses messages, which are sent from the first EPU and targeted to the third EPU.

Additional Restrictions for Recursive Cycles

With recursive cycles, one has to consider an additional restriction: The number of parallel recursions with different targets must be smaller or equal to the number of available recursive channels. The optimisation problem becomes more complex with this additional restriction because one has to decide which command group should be moved to a later EPU to fulfil all constraints.

We formalise the constraints in order to use an SMT solver to check if an EPU network can be transformed into an equivalent network that fulfils the hardware restrictions:

We assume the set of all EPU commands $c \in \mathbb{C}$ to be partitioned into command groups $g \in \mathbb{G} = 2^{\mathbb{C}}$. For every command group, the variable $v_g \in \mathbb{E}$ encodes on which EPU all commands of this command group are located. We analyse all target edges in a given EPU network and distinguish them into forward and backward edges:

- The set $D \subseteq \mathbb{G} \times \mathbb{G}$ of *forward edges* between command groups contains all combination of command groups g_1, g_2 such that there are $c_1 \in g_1$ and $c_2 \in g_2$ with $\mathit{target}(c_1) = c_2$ and $\mathit{epu}(c_1) < \mathit{epu}(c_2)$.
- The set $D_R \subseteq \mathbb{G} \times \mathbb{G}$ of *backward edges* between command groups contains all combination of command groups g_1, g_2 such that there are $c_1 \in g_1$ and $c_2 \in g_2$ with $\mathit{target}(c_1) = c_2$ and $\mathit{epu}(c_1) > \mathit{epu}(c_2)$.

For every backward edge $(g_1, g_2) \in D_R$ we introduce a variable $h_{g_1,g_2} \in \{1, 2, 3, 4\}$ indicating the recursive channel used for this backward edge. Finally we encode the switch boxes (see Section 5.2) as follows: For every EPU $e \in \mathbb{E}$ and every channel $h \in \{1, 2, 3, 4\}$ we introduce a variable $s_{e,h} \in \{\mathit{straight}, \mathit{out}, \mathit{in}\}$ indicating the state of this switch box.

We are now looking for an assignment of the variables v_g, h_{g_1,g_2} and $s_{e,h}$ for all $g, g_1, g_2 \in \mathbb{G}, e \in \mathbb{E}$ and $h \in \{1, 2, 3, 4\}$ which fulfils the following constraints:

- Every forward edge must remain forward, i. e. for all $(g_1, g_2) \in D$ we have $v_{g_1} < v_{g_2}$ and
- every backward edge has a channel, i. e. for all $(g_1, g_2) \in D_R$ we need the following switch box configuration on the assigned channel: out on the EPU of g_1, in on the EPU of g_2 and straight for all EPUs in between:

$$s_{v_{g_1},h_{g_1,g_2}} = \mathit{out} \wedge s_{v_{g_2},h_{g_1,g_2}} = \mathit{in} \wedge \bigwedge_{g_1 < v < g_2} s_{v,h_{g_1,g_2}} = \mathit{straight}$$

In order to encode the second property into an SMT solver, the formula can be unrolled: All possible recursive channels can be generated in advance and then only checked if any of these recursive channels matches the current variable assignment.

In order to find a proper mapping of the commands onto the EPUs, we encode the other EPU network constraints regarding the available memory for flags, data and FIFOs, too. These additional constraints check how much memory the commands consume, which are included in the command groups.

5.7.4. Enqueuing Commands

In the definition of the execution of the formal EPU modal given in Section 5.3.1 every EPU is equipped with two sets storing the commands being enqueued via BTSC or ATSC. As already introduced in Section 5.2 these sets are implemented as FIFOs on the actual hardware. Implementing a set in hardware would introduce additional overhead to maintain the invariance that an element can only be once in a set. There are multiple approaches, e. g. sorting the set elements with every insertion or comparing every newly inserted element with all elements in the set. Any generic solution would use numerous resources, which is why we use the existing flags and conditions to ensure that specific BTSC or ATSC commands only get enqueued once. An EPU command of the actual hardware can specify two additional conditions specifying if the BTSC or ATSC command, respectively, is enqueued or not. With the introduction of an additional flag to remember if the command was already enqueued, we can use this mechanism to prevent a command from being enqueued twice. The enqueued command then resets the flag so that it can be enqueued again for the next timestamp.

5.8. Practical Simplifications

This section discusses several practical engineering tweaks used in translating TeSSLa specifications to EPU configurations.

5.8.1. Flow Graph Optimisations

We start with adjustments on the level of the specification's flow graph. Although these adjustments are performed on the flow graph before its translation into the EPU network, they are especially relevant for the EPU backend, because the EPU backend is the only implementation which uses the translation result directly without further optimisations.

Unary filter. The binary operation filter can be replaced with a unary slift if both inputs of the filter are derived from the same stream. For example $\mathsf{filter}(\mathsf{const}(x, c), c)$ is a common pattern to replace the true events on $c \in \mathcal{S}_{\mathbb{B}}$ with the constant value x and filter out the false events. This specification is equivalent to $\mathsf{slift}(f)(c)$ with $f\colon \mathbb{B} \to \mathbb{D}_{\perp}$ given by

$$f(a) = \begin{cases} x & \text{if } a = \mathit{true}, \\ \perp & \text{otherwise.} \end{cases}$$

Further, $\mathit{filter}(c, c)$ is a special case of that pattern which filters a Boolean stream $c \in \mathcal{S}_{\mathbb{B}}$ for true events. This specification is equivalent to $\mathsf{slift}(f)(c)$ with $f\colon \mathbb{B} \to \mathbb{D}_{\perp}$ given by

$$f(a) = \begin{cases} a & \text{if } a = \mathit{true}, \\ \perp & \text{otherwise.} \end{cases}$$

Integrate Constants Into slift. Constant values $d \in \mathbb{D}$ in specifications are implicitly understood as $\mathsf{const}(d) \in \mathcal{S}_{\mathbb{D}}$, which is then translated as $\mathsf{default}(\mathsf{nil}, d)$. This operator becomes an EPU command sending a single message before the first timestamp change of its EPU. If this stream is used as input of an slift it can be integrated directly into the lifted function and thereby reduce the arity of the operator.

For example $\mathsf{slift}(f)(x, \mathsf{default}(\mathit{nil}, y))$ for a stream $x \in \mathcal{S}_{\mathbb{D}}$, a constant $y \in \mathbb{D}$ and a binary function $f\colon \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ can be equivalently as a unary $\mathsf{slift}(g)(x)$ with $g\colon \mathbb{D} \to \mathbb{D}$ given by $g(a) = f(a, y)$. Translating this into an EPU commands no longer requires any BTSC commands waiting for a possible event on the second input stream because we now know that every incoming event on x immediately creates an outgoing event.

The same approach can be applied for the ternary $\mathsf{slift}(\mathit{ite})(w, x, y)$ with streams $w \in \mathcal{S}_{\mathbb{B}}, x \in \mathcal{S}_{\mathbb{D}}$ and a constant $y \in \mathbb{D}$. The resulting $\mathsf{slift}(\mathit{ite})(w, x, \mathsf{default}(\mathsf{nil}, y))$ can be expressed as $\mathsf{slift}(g)(w, x)$ with $f\colon \mathbb{B} \times \mathbb{D} \to \mathbb{D}$ given by $f(a, b) = \mathit{ite}(a, b, y)$. In case of $\mathsf{slift}(\mathit{ite})(w, \mathsf{default}(\mathsf{nil}, x), \mathsf{default}(\mathsf{nil}, y))$ it can even become a unary $\mathsf{slift}(g)(w)$ with $g\colon \mathbb{B} \to \mathbb{D}$ given by $g(a) = \mathit{ite}(a, x, y)$.

Similarly $\mathsf{merge}(x, y)$ for a stream $x \in \mathcal{S}_{\mathbb{D}}$ and a constant $y \in \mathbb{D}$ can be replaced with $\mathsf{default}(x, y)$ with requires fewer synchronisation logic.

5.8.2. EPU Network Optimisations

The following adjustments are applied after the translation of the specification's flow graph into an EPU network. They are based on integrating commands into the target command in the next EPU to reduce the specification depth and cannot be expressed on the flow graph.

Integrate time. Instead of translating the time operator into a custom command in nearly all cases, the target command can be adjusted to use the time t instead of the events value i. This adjustment cannot be expressed on the TeSSLa flow graph because the **time** operator is used there to express accessing an event's timestamps instead of its values.

Integrate Constants Into last. In the same way that constant values $d \in \mathbb{D}$ could be integrated into a following slift they can also be integrated into a following **last**. However, the resulting construct has no longer a trivial TeSSLa operator expressing its semantics. Hence, this integration optimisation only makes sense directly on the EPU network:

The specification $\textbf{last}(\mathsf{default}(\mathsf{nil}, d), r)$ can be translated as the commands for **last** where $m_{current}$ is initialised with d and c_{btsc} is initially enqueued in the BTSC FIO. Note that initially setting $m_{current} = d$ would be wrong because a **last** never produces an event when triggered at timestamp 0. This distinction between events at timestamp 0 and all later events is what makes this case unique compared to the integration into slift discussed in the previous section.

5.8.3. Translating Recursive Specifications

The translation scheme for recursions presented in Section 5.6 works for arbitrary cycles in the specification's flow graph. A typical pattern for simple recursive specifications, however, is shown in Figure 5.22 on the left and consist of the following elements: A **last** operator which is used to copy the old aggregated value to the current timestamp for each event on the trigger stream r, some arbitrary operators depicted as cloud C which perform computations based on the old aggregated value and some new incoming events represented by an observed stream o, and a default operator, which defines a base case needed to get the recursion started.

The specification

$$z = \mathsf{default}(\textbf{last}(z, x) + y, 0)$$

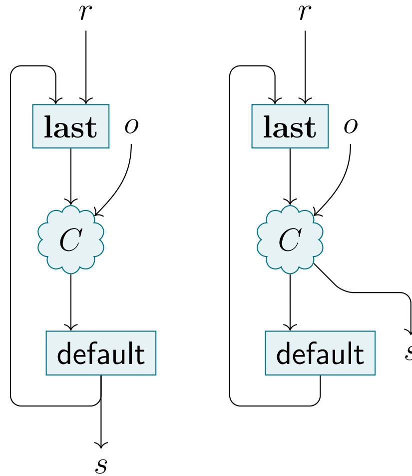


Figure 5.22.: Abstract flow graph of recursive specifications suitable for a simple translation.

from Section 5.6.1 falls into this category, too: The **last** reproduces the old aggregated value at the current timestamp for every event on x, then the latest value on y is added to the aggregated value and finally the default injects the base case of 0 at timestamp into the cycle. We will therefore revisit this example in this section and discuss how to simplify the translation by removing the default operator from the cycle. If the default is the last operator whose output is directly fed into the recursive value input of the **last**, then the default can be integrated into the EPU commands created from the recursive **last**.

For recursive specifications matching this pattern, we can adjust the memory initialisation. The initial memory tuple $\boldsymbol{m}$ for the EPU commands used to translate a recursive **last** was given in Definition 5.15 as

$$\boldsymbol{m} = (m_{data}, m_{prev}, m_{time}, m_{state}) = (\bot, \bot, -\infty, \mathit{valid}).$$

If we already apply the effect of the default with a constant value $k \in \mathbb{D}$ we adjust this to

$$\boldsymbol{m} = (m_{data}, m_{prev}, m_{time}, m_{state}) = (k, \bot, 0, \mathit{valid}).$$

With the initialisation m_{time} and $m_{prev} = \bot$ we still ensure that the **last** never produces an event at timestamp 0 because c_{trig} only sends out events if $m_{state} = \mathit{valid}$ and either $m_{time} = t \wedge m_{prev} \neq \bot$ or $m_{time} < t \wedge m_{value} \neq \bot$ which is both not initially fulfilled.

The second adjustment is that we do not need to initially enqueue c_{btsc} in that case because there is no longer a need to inform the default about a timestamp

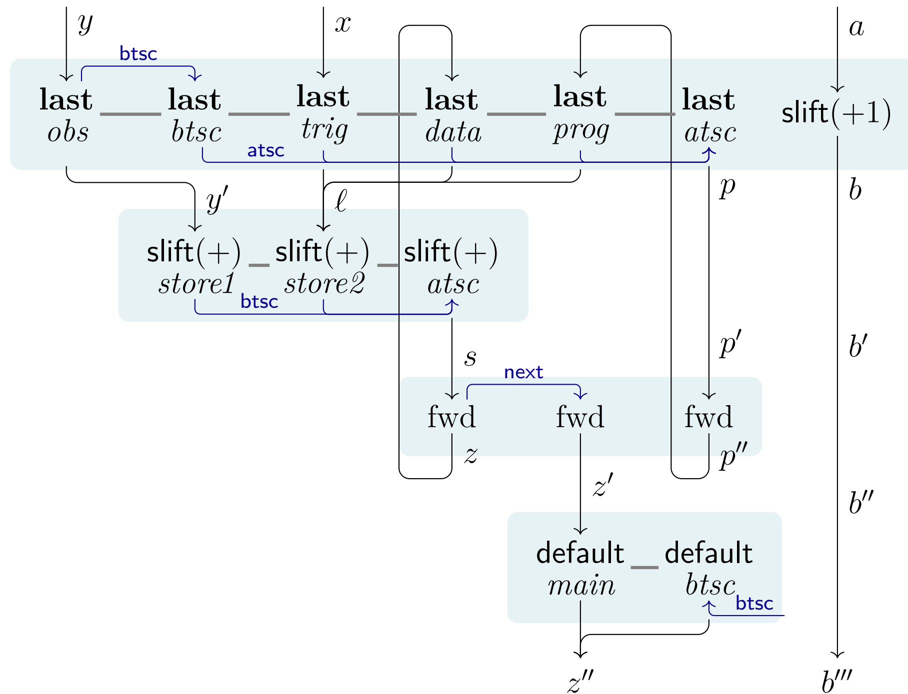


Figure 5.23.: EPU network diagram for the simplified translation of the recursive example.

change by a progress message. The adjusted EPU network diagram for the example specification is shown in Figure 5.23.

The EPU commands for the default are still there but are located after the recursion now. This adjustment reduces the number of EPUs involved in the recursion by one. The default is still required after the recursion to send out the initial output of 0 at timestamp 0. In the case of the flow graph shown in Figure 5.22 on the right, where the output stream s is derived directly from the internal specification C and the default is not involved, the default can be eliminated.

5.9. Optimising Simple Recursions

Looking at the EPU commands defined so far – except those designed to implement recursive cycles – there is a common pattern: Storing data on arrival and processing it before the timestamp increment. This pattern implements the sync operator, which performs the synchronisation required for the signal view. In combination with an operation performed before the timestamp increases, this pattern realises the semantics of the slift operator. This section aims to generalise this pattern further

by allowing the execution of arbitrary functions at the arrival of events and when computing the output event before the timestamp increase. The TeSSLa operator foldLift provides a formal TeSSLa semantics to this store and BTSC mechanism of the EPU commands.

The foldLift operator is based on the foldn operator defined in Definition 3.42 in Section 3.3.5. The foldn operator takes a function f, which is folded over the events of the input streams. This function is split up into several smaller functions to simplify the mapping onto the mechanisms available on the EPUs:

- For every input stream, there is a function $s_i \colon \mathbb{D} \times M_i \to M_i$ which is called for every event on the input stream x_i. The foldLift does not directly aggregate the output stream but instead aggregates over a tuple of memory cells from the partitioned memory domain $M = \prod_{i=1}^{n} M_i$.
- For every timestamp with an event on at least one of the input streams, the function $u \colon M \to M$ is called once to update the entire memory.
- For every timestamp with an event on at least one of the input streams, the function $q \colon M \to \mathbb{D}_\perp$ is called to derive the output event from the current memory valuation.

From the perspective of the TeSSLa semantics, this separation complicates the definition but does not limit the expressiveness in comparison to foldn. However, it makes it possible to directly translate expressions defined using foldLift to EPU commands. Every s_i only uses its local memory, and u and f are only called once per relevant timestamp. This restriction makes the semantics independent of the execution order of the individual s_i, making the TeSSLa operator's semantics and the corresponding EPU commands equivalent. Otherwise, the semantics of the EPU commands would depend on the order of arrival of the events with similar timestamps. In many cases, this is not a problem, but it depends on the specific functions if the semantics realised by the EPU commands are still expressible in TeSSLa.

The foldLift operator and the corresponding EPU commands allow evaluating recursions on a single EPU. Instead of using additional synchronisation mechanisms to integrate recursive messages into the outer pipeline, we can now evaluate a recursion in a single step of the pipeline. We will see in the evaluation in Chapter 8 that this improves performance for small recursions with high event rates because the additional synchronisation overhead is reduced. However, for complex recursions, this approach might be even slower because the EPU must compute the entire recursion in a chain of next commands before any other message can be processed, so the EPU may become a bottleneck. Note that the foldLift is not sufficient to gain the full expressiveness because mutual recursive expressions cannot be directly expressed using foldn without the introduction of additional data types such as tuples, which the EPUs do not support.

We now define the foldLift operator as a special case of the foldn operator and then give the corresponding EPU commands:

Definition 5.20 (Semantics of the Operator foldLift)**.** Let

$$M = \prod_{i=1}^{n} M_i$$

be a tuple of arbitrary memory data types. The operator

$$\mathsf{foldLift}\colon M \times \prod_{i=1}^{n} (M_i \times \mathbb{D} \to M_i) \times (M \to M) \times (M \to \mathbb{D}_\perp) \to \mathcal{S}_\mathbb{D}^n \to \mathcal{S}_\mathbb{D}$$

is defined as

$$\mathsf{foldLift}(\boldsymbol{a}, \boldsymbol{s}, u, q)(\boldsymbol{x}) = \textbf{lift}(q)(\mathsf{foldn}(\boldsymbol{a}, g)(\boldsymbol{x})).$$

It takes an initial memory tuple $\boldsymbol{a} \in M$, a corresponding tuple $\boldsymbol{s}$ of store functions $s_i\colon M_i \times \mathbb{D} \to M_i$ as well as a final memory update function $u\colon M \to M$ and an output computation function $q\colon M \to \mathbb{D}_\perp$. The aggregating function $g\colon M \times (\mathbb{D}_\perp)^n \to M$ is given by

$$g(\boldsymbol{m}, \boldsymbol{x}) = u(\boldsymbol{m}')$$

with $\boldsymbol{m}'$ being

$$m'_i = \begin{cases} s_i(m_i, x_i) & \text{if } x_i \neq \perp, \\ m_i & \text{otherwise.} \end{cases}$$

⌟

Definition 5.21 (Commands for foldLift)**.** Let $\boldsymbol{x} \in \mathcal{S}_\mathbb{D}^n$ be a tuple of streams and the derived stream $z \in \mathcal{S}_\mathbb{D}$ given by $z = \mathsf{foldLift}(\boldsymbol{a}, \boldsymbol{s}, u, q)(\boldsymbol{x})$. This specification is translated into the EPU commands c_k for all $0 \leq k < n$ and the EPU command c_{btsc} given by:

$$\begin{aligned} c_k\colon m'_k &= s_k(m_k, i) \\ c_{btsc}\colon \boldsymbol{m}' &= u(\boldsymbol{m}) \\ o &= q(\boldsymbol{m}') \end{aligned}$$

The EPU's memory tuple is initialised as follows:

$$\boldsymbol{m} = \boldsymbol{a}.$$

The corresponding EPU network diagram is given in Figure 5.24. ⌟

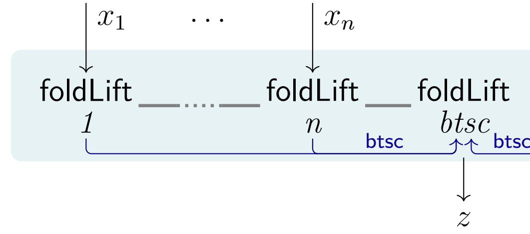


Figure 5.24.: EPU network diagrams for the commands for foldLift.

One can see the idea of the foldLift operator in the above definition of its EPU commands and the corresponding EPU network diagram: It can be imagined as turning around the responsibilities and represent what an EPU can do as a TeSSLa operator. This approach allows us to have an operator with a formal semantics that can express EPU commands.

In the following, we give some examples of how foldLift can be used to translate TeSSLa specifications to EPU commands.

Example 5.22 (Express slift Using foldLift)**.** Let $\boldsymbol{x} \in \mathcal{S}_{\mathbb{D}}^n$ be a tuple of streams and $f\colon \mathbb{D}^n \hookrightarrow \mathbb{D}$ a partial function on the data domain. The following equivalence follows from the above definitions:

$$\mathsf{slift}(f)(\boldsymbol{x}) \equiv \mathsf{foldLift}(\bot, \boldsymbol{s}, u, q)(\boldsymbol{x})$$

with the initial memory tuple $\bot \in (\mathbb{D}_\bot)^n$, the store functions $s_i\colon \mathbb{D} \times \mathbb{D} \to \mathbb{D}$, the memory update function $u\colon \mathbb{D}^n \to \mathbb{D}^n$ and the output computation function $q\colon \mathbb{D}^n \to \mathbb{D}_\bot$ given by:

$$\begin{aligned} s_i(m_i, d) &= d \\ u(\boldsymbol{m}) &= \boldsymbol{m} \\ q(\boldsymbol{m}) &= f(\boldsymbol{m}) \end{aligned}$$

This usage of foldLift does not aggregate values over multiple events because the old value m_i is ignored in s_i. As a result, we only store the old value until it is used, which is the semantics of sync which is embedded in slift.

Next we show how the recursive aggregation operators count, sum, minimum and maximum defined in Section 3.3.5 can be expressed using foldLift:

Example 5.23 (Express count Using foldLift)**.** Let $x \in \mathcal{S}_{\mathbb{D}}$ be a stream. The following equivalence follows from the above definitions:

$$\mathsf{count}(x) \equiv \mathsf{foldLift}(0, s, u, q)(x)$$

with the initial memory value $0 \in \mathbb{D}$, the store function $s\colon \mathbb{D} \times \mathbb{D} \to \mathbb{D}$, the memory update function $u\colon \mathbb{D} \to \mathbb{D}$ and the output computation function $q\colon \mathbb{D} \to \mathbb{D}_\bot$ given by:

$$\begin{aligned} s(m, x) &= m + 1 \\ u(m) &= m \\ q(m) &= m \end{aligned}$$

⌟

The sum operator could be expressed similarly using the store function $s(m, x) = m + x$ instead.

Example 5.24 (Express minimum Using foldLift). Let $x \in \mathcal{S}_\mathbb{D}$ be a stream. The following equivalence follows from the above definitions:

$$\mathsf{minimum}(x) \equiv \mathsf{foldLift}(\bot, s, u, q)(x)$$

with the initial memory value $\bot \in \mathbb{D}_\bot$, the store function $s\colon \mathbb{D}_\bot \times \mathbb{D} \to \mathbb{D}_\bot$, the memory update function $u\colon \mathbb{D}_\bot \to \mathbb{D}_\bot$ and the output computation function $q\colon \mathbb{D}_\bot \to \mathbb{D}_\bot$ given by:

$$\begin{aligned} s(m, x) &= \begin{cases} x & \text{if } m = \bot, \\ \min(m, x) & \text{otherwise.} \end{cases} \\ u(m) &= m \\ q(m) &= m \end{aligned}$$

⌟

As already mentioned in Section 5.4 on the definition of the EPU commands for TeSSLa operators, we use data types with the additional value $\bot$ for memory cells only to improve the readability. In the actual implementation, this is a memory cell containing the data values and an additional flag indicating if the value was initialised or not.

The maximum operator could be expressed similarly using max instead of min.

As a final example we have a look at how to express resetCount defined in Definition 3.46 using foldLift. This example is interesting because its two input streams combine multiple input streams with an actual aggregation.

Example 5.25 (Express resetCount Using foldLift). Let $x, r \in \mathcal{S}_\mathbb{D}$ be two streams. The following equivalence follows from the above definitions:

$$\mathsf{resetCount}(x, r) \equiv \mathsf{foldLift}((0, 0), (s_x, s_r), u, q)(x, r)$$

with the initial memory tuple $(0,0) \in \mathbb{D}^2$, the two store functions $s_x, s_r \colon \mathbb{D} \times \mathbb{D} \to \mathbb{D}$, the memory update function $u \colon \mathbb{D}^2 \to \mathbb{D}^2$ and the output computation function $q \colon \mathbb{D}^2 \to \mathbb{D}_\perp$ given by:

$$
\begin{aligned}
s_x(x, m_x) &= 1 \\
s_r(r, m_r) &= 0 \\
u(m_x, m_r) &= (0, m_r + m_x) \\
q(m) &= m_r
\end{aligned}
$$

⌟

In this example, we make use of the individual functions to express the complex semantics of a resetCount with simple functions: The memory tuple $m = (m_x, m_r)$ stores the current increment in m_x which is either 0 or 1 and the current value in m_r. With this convention s_x can set m_x to 1 for every incoming event on x in order to count that event and s_r resets the counter m_r to 0. The memory update function than adds m_x to m_r and resets m_x to 0. The output is then m_r, which is always the current counter value.

Practical Simplifications

Looking at the above examples, one can see that the differentiation into a separate store function, a memory update function and an output computation function does not make much sense in the unary case with only one input stream. For these simple aggregation operators count, sum, minimum and maximum we always have

$$
\begin{aligned}
u(m) &= m \\
q(m) &= m.
\end{aligned}
$$

The three functions are evaluated one after another for every event on the one input stream. Evaluating u and q in an additional command with BTSC execution is not necessary in that case.

As a further simplification the initial evaluation at timestamp 0 can be removed if it has no effect, i. e. $u(\boldsymbol{a}) = \boldsymbol{a}$ and $q(\boldsymbol{a}) = \perp$. In that case, we do not schedule c_{btsc} initially for BTSC execution.

For example in case of slift we have $\boldsymbol{a} = \perp$, $u(\perp) = \perp$ and $q(\perp) = \perp$. In case of count however we have $a = 0$ and $u(0) = 0$, but $q(0) = 0 \neq \perp$ because count produces and output stream which is initialised with 0 even if the input stream does not have an event at timestamp 0.

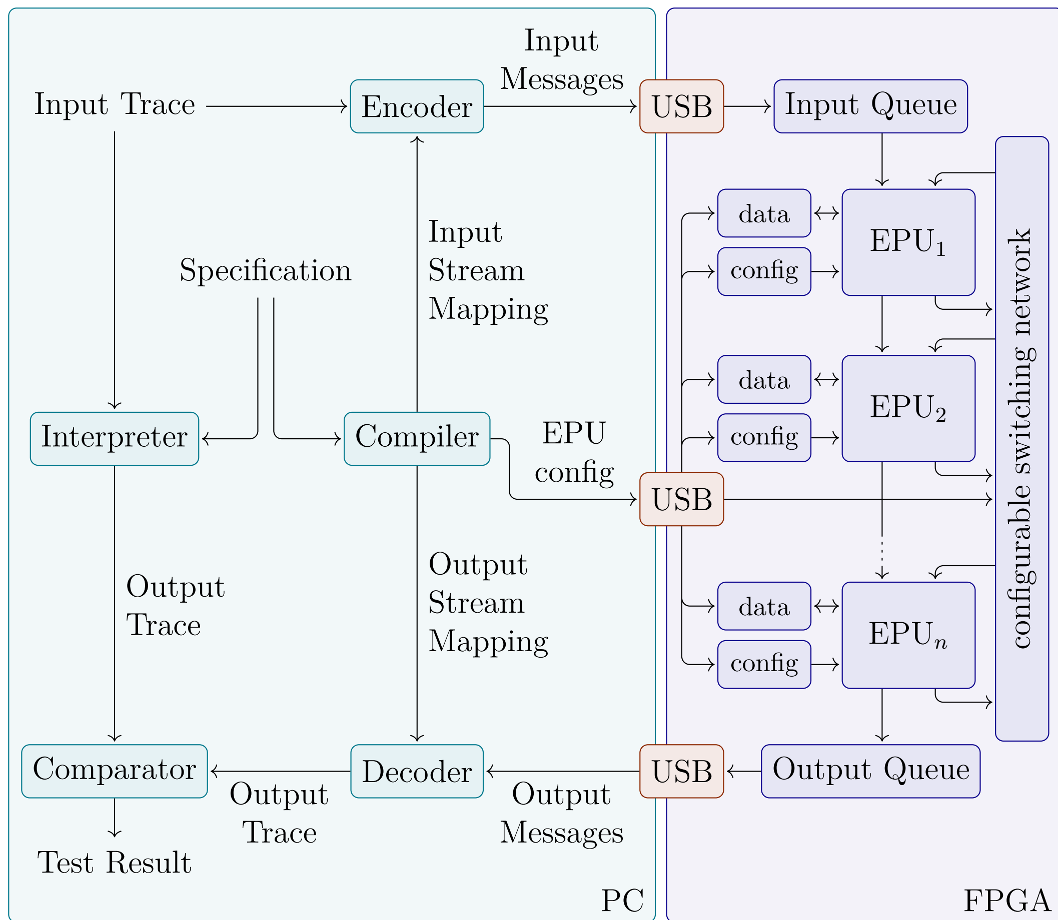


Figure 5.25.: Architectural overview diagram of the EPU hardware and test workflow on the PC communicating with the EPUs.

5.10. Integration and Test Setup

A pipeline of EPUs is synthesised onto an FPGA and then at runtime configured with an EPU configuration. While synthesising an FPGA image can take several minutes up to hours, this reconfiguration is fast because we only write a small configuration data of a few kilobytes through a USB connection into memory on the FPGA. Figure 5.25 shows on the right an architectural overview of the pipeline of EPUs on an FPGA. The EPU configuration is written through the USB connection represented by the USB node in the centre of the diagram. As shown in Figure 5.4 in Section 5.2, every EPU has a command memory and a data & flag memory.

- The command memory contains the command configuration containing the EPU commands and the operation configuration, as well as the condition configuration, which is referenced in the command configuration as described in Section 5.7. The command memory is written during the configuration and only read by the EPU.
- The data memory is initialised during the configuration and read and written by the attached EPU during the execution. It contains the data and flag memory as well as the BTSC and ATSC FIFOs.
- The switching network controls the connections of the EPUs, which are going against the direction of the outer pipeline. These connections are used to realise the cyclic paths in the dependency graphs of recursive TeSSLa specifications. The switch boxes are configured as described in Section 5.7. The switch boxes' configuration is written during the configuration and cannot be changed during the execution of the EPUs.

On the left of Figure 5.25 the integration test workflow for hardware EPUs is shown. A specification is compiled into an EPU configuration and an input and output stream mapping on the PC connected to the FPGA via USB. The input mapping indicates which input streams are mapped to which commands on which EPUs. Output messages are sent to a virtual EPU located after the final EPU of the pipeline. The output mapping maps those messages and their target command ID back to streams.

For the integration test, the input trace is encoded and sent via USB to the input queue on the FPGA. Messages from the input queue are then fed into the pipeline of EPUs. Messages from the output queue are read through USB and are then decoded on the PC. The output trace is then compared with an output trace generated by the interpreter, executed on the same specification and trace. The interpreter is used here again as a reference implementation in a similar way as in Section 4.5.

Transmitting input and output messages through a USB connection from the PC to the FPGA and back is a bottleneck of the setup depicted in Figure 5.25. In production, especially the input messages are usually either provided by the same FPGA containing the EPUs or transmitted through specialized high-speed connections. This test setup, however, serves the two purposes of correctness tests and performance evaluation: For the correctness tests, input messages are created on the PC and sent to the EPUs to compare the output of the EPUs with the interpreter's output. The input and output queues are included in the test setup, in particular for the performance evaluation in Chapter 8: All input messages are written into the input queue, then the EPUs process the input messages and write their output into the output queue, and finally, the output queue is read out by the PC. With this approach, the execution time of the EPUs is measured independently of the USB connection.

5.11. Conclusion

This chapter introduced the first of two hardware backends for TeSSLa presented in this thesis: We map a specification's flow graph onto a linear sequential pipeline of EPUs that run in parallel. Data flow processors inspired the design of the individual EPUs of the pipeline because the address of the incoming message determines the executed command. A TeSSLa specification is compiled into an EPU network realising the synchronous semantics for the specification that was introduced in Section 4.1. Recursive specification, i. e. cycles in the flow graph, are supported on the EPUs by additional outputs which are not connected to the next EPU but earlier EPUs in the pipeline. For the translation of recursive specifications onto the EPUs, a single **last** operator is translated into several different EPU commands on at least three EPUs. However, with this construction, we gain the full expressiveness of all timestamp-conservative TeSSLa specifications on the EPUs. Simple recursive specifications that are not mutual recursive can be expressed using the foldLift operator, which avoids this complex construction. It can be translated into fewer commands on a single EPU. The evaluation in Chapter 8 shows that this approach can improve the performance for small recursions with high event rates in comparison with the non-optimised translation.

The formal execution model for EPU commands and EPU networks were tested with a simulation. Integration tests of the EPU compiler and the EPU hardware have been performed by comparing the EPU's output with the interpreter using a hardware simulation and actual hardware provided by Accemic. An empirical evaluation of the EPU's performance in comparison with the software compiler from Chapter 4 and the FPGA synthesis in Chapter 7 is discussed in Chapter 8.

6 Implementing Asynchronous TeSSLa

In the last chapters, we studied synchronous approaches to implement the TeSSLa semantics. The synchronous monitors use a single global current timestamp indicating the synchronous progress for all streams of the specification. For the FPGA synthesis discussed in Chapter 7, a different approach is taken to support asynchronous evaluations. The monitoring semantics defined in Section 3.5 already support asynchronous evaluations in that every stream can have a different progress. The progress of a derived stream is not restricted by a global synchronisation but only determined by the application of its defining TeSSLa operators and the progress of its arguments. However, the monitoring semantics are not implemented directly, but this chapter introduces the abstract monitoring semantics on abstract monitoring streams. This semantics are shown to be an abstraction of the monitoring semantics. This abstraction differs from the synchronous semantics given in Section 4.1 in that they preserve more of the asynchronicity of the monitoring semantics. The abstract monitoring streams can still have different progress and the individual abstract operators are shown to be perfect abstractions of their counterparts on monitoring streams.

A monitoring stream was defined in Section 3.5 as a possibly infinite set of streams. This set can be seen as the set of all possible continuations of a given prefix. A refinement relation and hence a smallest element, the entirely unknown stream, was defined. Starting from this smallest element, one can compute the fixed point by applying the function step by step; see Lemma 3.81 (Construction of the Least Fixed Point) in Section 3.5.4.

Actual implementations cannot directly handle the set of infinitely many streams. The abstract monitoring streams introduced in this chapter provide an abstract representation of such a set of streams. This abstraction ignores that monitoring streams can have multiple sequences of perfect knowledge, i. e. sequences where all the streams in the set are equal. For the online monitoring to work starting with the smallest element and refining this stepwise, we only need to represent the initial sequence of full knowledge. The progress of a monitoring stream was defined in Definition 4.1 in Section 4.1.1 as the timestamp, until which all its streams are equal. This progress can either be inclusive or exclusive, i. e. at this particular timestamp, the streams can still be equal (inclusive progress), or the streams can be equal up to but not including this particular timestamp (exclusive progress). Thus, abstract

monitoring streams explicitly encode the progress and contain of the sequence of events up to the progress.

With this informal idea of abstract monitoring streams, we can now discuss a first example, illustrating the difference of

- the TeSSLa monitoring semantics on monitoring streams,
- the synchronous monitoring on synchronised streams and
- the abstract monitoring semantics on abstract monitoring streams.

The main difference between monitoring streams and abstract monitoring streams is that the abstract monitoring streams do not encode any information about how the stream can continue after its progress.

Example 6.1 (Comparison of Different Abstractions of Monitoring Streams and Semantics)**.** Let $v, r \in \mathcal{P}_{\mathbb{D}}$ be two free monitoring streams and let $\ell \in \mathcal{P}_{\mathbb{R}}$ be a derived monitoring stream given by:

$$\ell = \mathbf{last}(v, r)$$

Figure 6.1 shows the application of this specification to exemplary input streams on the left in black. Let $q \in \mathcal{Q}_2$ a synchronised stream given as the synchronous abstraction of v, r:

$$q = \alpha((v, r)).$$

The synchronised stream $\ell_q \in \mathcal{Q}_1$ is derived from q by application of the synchronous monitoring to q and shown in the middle of Figure 6.1 in red. Let $v_p, r_p \in \mathcal{P}_{\mathbb{D}}$ be abstract monitoring streams (defined below in this chapter) given as the abstraction of v, r:

$$v_p = \alpha(v) \text{ and } r_p = \alpha(r).$$

The abstract monitoring stream $\ell_p \in \mathcal{P}_{\mathbb{D}}$ is derived from v_p and r_p using the abstract monitoring semantics (defined below in this chapter) and shown in the right part of Figure 6.1 in blue.

The monitoring streams v and r have a minimal exclusive progress of 2. Thus, the synchronised stream abstracted from v and r has a common exclusive progress of 2. Due to the synchronisation any information about the behaviour of the streams beyond that progress is lost.

The abstract monitoring streams v_p and r_p can encode the different progresses of v and r. However, the monitoring semantics yield more details on ℓ than the abstract monitoring semantics on ℓ_p: We can see on the monitoring stream ℓ that that $\mathbf{last}(v, r)$ can only generate events when v has events. If there is no event on v,

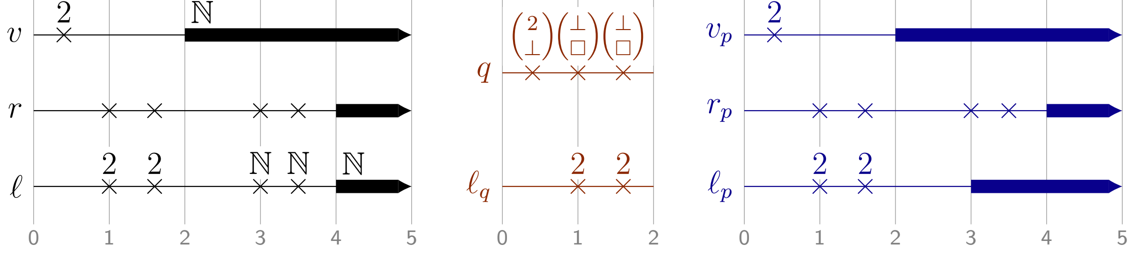


Figure 6.1.: Exemplary evaluation of the specification $\ell = \textbf{last}(v, r)$ with monitoring semantics on the left in black in comparison with two abstractions: The abstraction to synchronous monitoring is shown in the middle in red. The abstraction to abstract monitoring semantics is shown on the right in blue.

then there cannot be an event on ℓ. For the two events at and after timestamp 3, ℓ encodes that there is an event with unknown value, because v is only known until timestamp 2. The abstraction ℓ_p only encodes that there is no event up to but not including 3 without any details on how the stream continues after timestamp 3. ⌟

6.1. Abstract Monitoring Streams

Abstract monitoring streams are introduced as streams in [CHL+18]. Their definition is reproduced here with minor adjustments to analyse them as an abstraction of monitoring streams.

Definition 6.2 (Abstract Monitoring Streams [CHL+18])**.** An *abstract monitoring stream* over a time domain $\mathbb{T}$ and a data domain $\mathbb{D}$ is a finite or infinite sequence over timestamped elements from the data domain:

$$s = \langle (t_0, d_0), (t_1, d_1), \ldots \rangle \in \mathcal{R}_{\mathbb{D}} \subseteq (\mathbb{T}_\infty \times \mathbb{D} \cup \{\bot, _\})^\infty.$$

Every such sequence $s \in \mathcal{R}_{\mathbb{D}}$ fulfils the following properties:

- The sequence s contains at least one element,
- the timestamps are strictly increasing, i. e. $\forall\, 0 < i < |s| \colon t_{i-1} < t_i$,
- the symbols $\bot$ indicating inclusive progress and $_$ indicating exclusive progress occur only in the last element of s,
- the symbolic timestamp ∞ only occurs as $(\infty, _)$ in s. ⌟

In the same way as already introduced for $\mathcal{S}^n_{\mathbb{D}}$ (see Section 3.2.2) we indicate the Cartesian product $\mathcal{R}_{\mathbb{D}_1} \times \mathcal{R}_{\mathbb{D}_2} \times \ldots \times \mathcal{R}_{\mathbb{D}_n}$ of abstract monitoring streams over these data domains with the notation $\mathcal{R}^n_{\mathbb{D}}$.

An abstract monitoring stream can either be a finite or an infinite sequence of tuples of timestamps and values. In the finite case, special symbols can be used in the last tuple to indicate additional progress. Similar to the synchronised streams introduced in Section 4.1.2 the final tuple $(t, \perp)$ indicates inclusive progress, i. e. the knowledge that there exists no extension with an event until and including t. The final tuple $(t, _)$ indicates exclusive progress, i. e. the knowledge that extension cannot have events before t but might have an event at t. The special final tuple $(\infty, _)$ is used to encode full knowledge, i. e. there exists no extensions for this stream.

Similarly to Definition 3.15 (Timestamps of a Stream) in Section 3.2.1 and Definition 3.59 (Timestamps of a Monitoring Stream) in Section 3.5.1, we define on abstract monitoring streams:

Definition 6.3 (Timestamps of an Abstract Monitoring Stream)**.** For an abstract monitoring stream $s \in \mathcal{R}_\mathbb{D}$ we define $T(s) \subseteq \mathbb{T}$ to be the set of timestamps carrying events in s:

$$T(s) := \{t \in \mathbb{T} \mid s \text{ contains } (t, d) \text{ with } d \in \mathbb{D}\}.$$

We further define $T(\boldsymbol{s}) := \bigcup_{1 \leq i \leq k} T(s_i)$ for the timestamps of all events occurring in $\boldsymbol{s} \in \mathcal{R}^n_\mathbb{D}$. ⌟

As abstract counterpart for Definition 4.1 (Progress of a Monitoring Stream) from Section 4.1.1 we define similar to Definition 4.4 (Progress of a Synchronised Stream) from Section 4.1.2:

Definition 6.4 (Progress of an Abstract Monitoring Stream)**.** Let $s \in \mathcal{R}_\mathbb{D}$ be an abstract monitoring stream. Then the *progress* of s is the supremum of all timestamps used in s if it exists, or ∞ otherwise. If s ends in $(t, _)$ or if there is no maximal timestamp we call the progress *exclusive*, otherwise *inclusive*. ⌟

Example 6.5 (Abstract Monitoring Streams)**.** The stream without any progress

$$s_0 = \langle (0, _) \rangle$$

consists of a single tuple indicating an exclusive progress of 0. The empty sequence is not a valid abstract monitoring stream. The empty stream with infinite progress

$$s_1 = \langle (\infty, _) \rangle$$

does not contain any events and can not be extended because the progress is already infinite. Both streams do not contain any timestamps:

$$T(s_0) = T(s_1) = \emptyset.$$

This stream with three events

$$s_2 = \langle(1,2),(3,5),(7,1)\rangle$$

has an inclusive progress of 7. Its extension

$$s_3 = \langle(1,2),(3,5),(7,1),(9,_)\rangle$$

has an exclusive progress of 9 and its extension

$$s_4 = \langle(1,2),(3,5),(7,1),(9,\bot)\rangle$$

has an inclusive progress of 9. All three streams have three timestamps:

$$T(s_2) = T(s_3) = T(s_4) = \{1,3,7\}.$$

We depict abstract monitoring streams using the same conventions as for their concrete counterparts. We no longer depict specific possible values next to the black bars because every value is possible after the progress for abstract monitoring streams. See Figure 6.2 for the visualisations of the streams defined above.

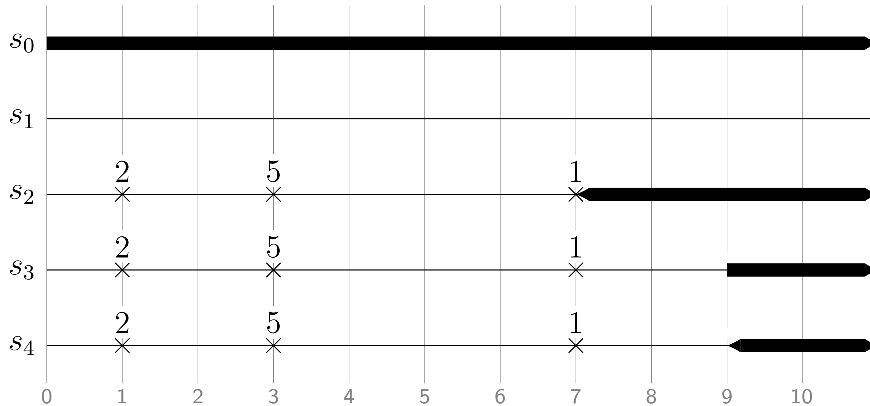


Figure 6.2.: Visualisation of the streams defined in Example 6.5.

⌟

In general we can distinguish the following cases for an abstract monitoring stream $s \in \mathcal{R}_{\mathbb{D}}$:

- A finite stream ending with (t,d) with a data value $d \in \mathbb{D}$ has inclusive progress of t.

- A finite stream ending with $(t, \bot)$ has inclusive progress of t. The special symbol $\bot$ indicates the absence of an event and represents inclusive progress without a terminal event.
- A finite stream ending with $(t, _)$ has exclusive progress of t. The special symbol $_$ indicates exclusive progress.
- An infinite stream with timestamps growing beyond any bound has exclusive progress of ∞.
- An infinite stream with timestamps converging in a way that there is a timestamp t such that $\forall\, t' \in T(s)\colon t' < t$ has exclusive progress of t.

Definition 6.6 (Abstraction Function for Abstract Monitoring Streams)**.** Let $s \in \mathcal{P}_{\mathbb{D}}$ be a monitoring stream with progress t and r be the sequence of all events (t', d) from s with $t' \leq t$. The *abstraction function* $\alpha\colon \mathcal{P}_{\mathbb{D}} \to \mathcal{R}_{\mathbb{D}}$ is given by

$$\alpha(s) := \langle (0, _) \rangle \quad \text{if } r \text{ is the empty sequence}$$

and otherwise by

$$\alpha(s) := \begin{cases} r & \text{if } s \text{ and } r \text{ have the same progress,} \\ r \,\&\langle (t, _) \rangle & \text{if } s \text{ has exclusive progress,} \\ r \,\&\langle (t, \bot) \rangle & \text{otherwise.} \end{cases}$$

⌟

The sequence of events r is already an abstract monitoring stream if it contains at least one event. If the sequence is finite, then r always has inclusive progress because r ends with an event. The other two cases extend r to encode the progress of s.

In Definition 3.57 in Section 3.5.1 a monitoring stream is defined as a set of streams: $\mathcal{P}_{\mathbb{D}} = 2^{\mathcal{S}_{\mathbb{D}}} \backslash \emptyset$. The function α defined above thus fits the concept of an abstraction function which provides an abstract representation for a set of elements. The progress of a monitoring stream is defined in Definition 4.1 in Section 4.1.1 such that all events with timestamps smaller or equal to the progress are common to all streams of the monitoring stream. By definition, the abstraction function α preserves the progress of the monitoring stream: The progress of an abstract monitoring stream is a perfect abstraction of the progress of a monitoring stream.

Definition 6.7 (Concretisation Function for Abstract Monitoring Streams)**.** Let $r \in \mathcal{R}_{\mathbb{D}}$ be an abstract monitoring stream with progress t and $s \in \mathcal{P}_{\mathbb{D}}$ be a monitoring stream such that $\alpha(s) = r$. The *concretisation function* $\gamma\colon \mathcal{R}_{\mathbb{D}} \to \mathcal{P}_{\mathbb{D}}$ is given by

$$\gamma(r) := \begin{cases} s & \text{if } r \text{ has infinitely many events,} \\ s|_{<t} & \text{if } r \text{ has finitely many events and exclusive progress,} \\ s|_{\leq t} & \text{otherwise.} \end{cases}$$

⌟

In the above definition we assume $s|_{<\infty} := s$ to simplify the notation. There is only one possibility to choose an $s \in \mathcal{P}_\mathbb{D}$ such that $\alpha(s) = r$ if r has infinitely many events. However, there are many possibilities to choose an $s \in \mathcal{P}_\mathbb{D}$ if r has only finitely many events, but since we only consider $s|_{\leq t}$ or $s|_{<t}$, the function γ is still well-defined.

In a similar way how streams $s \in \mathcal{S}_\mathbb{D}$ can be seen as functions $s\colon \mathbb{T} \to \mathbb{D}_\perp$ (see Definition 3.21 in Section 3.2.1) and synchronised streams $q \in \mathcal{Q}_k$ as functions $q\colon \mathbb{T} \to \mathbb{D}_\perp^k \cup \{?\}$ (see Definition 4.9 in Section 4.1.2) we can see abstract monitoring streams as functions, too:

Definition 6.8 (Functional View of Synchronised Streams)**.** Let $r \in \mathcal{R}_\mathbb{D}$ be an abstract monitoring stream with progress t. Then $f_r\colon \mathbb{T} \to \mathbb{D} \cup \{\perp, ?\}$ is its *functional view*. For any timestamp $t' \in \mathbb{T}$ we have

$$f_r(t') = \begin{cases} d & \text{if } r \text{ contains } (t', d) \text{ with } d \in \mathbb{D}, \\ \perp & \text{if } t' \notin T(r) \text{ and } t' < t \text{ if } t \text{ exclusive or } t' \leq t \text{ if } t \text{ inclusive}, \\ ? & \text{otherwise.} \end{cases}$$

⌟

As with streams and synchronised streams, the function maps a timestamp t to a value $d \in \mathbb{D}$ if r has an event with value d at time t. All timestamps t without an event at t are mapped to $\perp$ if t is lower than the progress of r. All timestamps after the progress of the stream are mapped to the symbol ?.

If the usage is clear from the context we use r to refer to f_r.

The TeSSLa monitoring semantics are defined in Definition 3.67 in Section 3.5.2 using the least fixed point. The least fixed point depends on the refinement relation on monitoring streams defined in Definition 3.63 in Section 3.5.1 as inverse subset relation. Since we want to define the abstract TeSSLa monitoring semantics as an abstraction of the TeSSLa monitoring semantics, we need a relation on abstract monitoring streams that corresponds to the refinement relation on monitoring streams:

Definition 6.9 (Prefix Relation on Abstract Monitoring Streams [CHL+18])**.** Let $s, r \in \mathcal{R}_\mathbb{D}$ be two abstract monitoring streams. We define the *prefix relation* $\sqsubseteq \subseteq \mathcal{R}_\mathbb{D} \times \mathcal{R}_\mathbb{D}$ as follows:

$$s \sqsubseteq r :\Longleftrightarrow \forall t \in \mathbb{T}\colon s(t) \in \{r(t), ?\}.$$

⌟

This prefix relation matches the refinement relation on monitoring streams from Definition 3.63 in Section 3.5.1, i. e. for two abstract monitoring streams $s, r \in \mathcal{R}_{\mathbb{D}}$ we have

$$s \sqsubseteq r \iff \gamma(s) \sqsubseteq \gamma(r).$$

The prefix relation forms a partial order $(\mathcal{R}_{\mathbb{D}}, \sqsubseteq)$. If we use the inverse prefix relation on abstract monitoring streams and the inverse refinement relation on monitoring streams, it follows directly from the definitions above:

Lemma 6.10 (Galois Connection for Abstract Monitoring Streams)**.** *The abstraction function $\alpha\colon \mathcal{P}_{\mathbb{D}} \to \mathcal{R}_{\mathbb{D}}$ and the concretisation function $\gamma\colon \mathcal{R}_{\mathbb{D}} \to \mathcal{P}_{\mathbb{D}}$ are a Galois connection between $(\mathcal{P}_{\mathbb{D}}, \sqsupseteq)$ and $(\mathcal{R}_{\mathbb{D}}, \sqsupseteq)$.*

The following example illustrates the Galois connection:

Example 6.11 (Galois Connection Between Monitoring Streams and Abstract Monitoring Streams)**.** The functions α and γ are a Galois connection between $(\mathcal{P}_{\mathbb{D}}^{k}, \sqsupseteq)$ and $(\mathcal{R}_{\mathbb{D}}, \sqsupseteq)$, i. e. we have

$$\forall\, a \in \mathcal{P}_{\mathbb{D}}, b \in \mathcal{R}_{\mathbb{D}} \colon \alpha(a) \sqsupseteq b \iff a \sqsupseteq \gamma(b).$$

Let $a \in \mathcal{P}_{\mathbb{D}}$ be the following monitoring stream:

$$a = \{\langle (1,1), (2,1) \rangle, \langle (1,1), (2,2) \rangle\}$$

We get the following abstract monitoring stream $\alpha(a) \in \mathcal{R}_{\mathbb{D}}$:

$$\alpha(a) = \langle (1,1), (2, _) \rangle$$

Next, we choose another abstract monitoring stream $b \in \mathcal{R}_{\mathbb{D}}$ which is a prefix of $\alpha(a)$, i. e. $\alpha(a) \sqsupseteq b$:

$$b = \langle (1,1), (1.5, \bot) \rangle$$

We get the following monitoring stream $\gamma(b) \in \mathcal{P}_{\mathbb{D}}$:

$$\gamma(b) = \{(1,1)\}|_{\leq 1.5}$$

Now we can see that $\gamma(b)$ is in fact a prefix of a, i. e. $a \sqsupseteq \gamma(b)$. ⌟

6.2. Abstract TeSSLa Operators

The abstract TeSSLa operators defined in this section are introduced as regular TeSSLa operators in [CHL+18]. Their definition is reproduced here with minor adjustments to analyse them as an abstraction of the TeSSLa operators defined in Section 3.2 on monitoring streams.

The TeSSLa semantics on monitoring streams provide maximal progress. It considers all possible continuations of all streams for the entire specification. The abstract TeSSLa operators on abstract monitoring streams provide maximal progress per operator, i. e. it does not consider all streams and especially not the lifted function in case of the **lift** operator.

Definition 6.12 (Semantics of the Abstract Operator $\mathbf{unit}^\#$ [CHL+18])**.** The abstract operator $\mathbf{unit}^\# \in \mathcal{R}_\mathbb{U}$ is given by

$$\mathbf{unit} := \langle (0, \Box), (\infty, _) \rangle.$$

The $\mathbf{unit}^\#$ operator is the stream with a single unit event at timestamp zero and infinite progress. The only addition compared to the **unit** operator on streams is the infinite progress.

Definition 6.13 (Semantics of the Abstract Operator $\mathbf{time}^\#$ [CHL+18])**.** The abstract operator $\mathbf{time}^\# \colon \mathcal{R}_\mathbb{D} \to \mathcal{R}_\mathbb{T}$ is given by $\mathbf{time}^\#(s) := z$ with the abstract monitoring stream z being defined as follows:

$$z(t) = \begin{cases} t & \text{if } t \in T(s), \\ s(t) & \text{otherwise.} \end{cases}$$

The **time** operator on streams explicitly states $\perp$ in the case $t \notin T(s)$ while this $\mathbf{time}^\#$ operator refers to $s(t)$ which can be either $\perp$ or ?. This way, the $\mathbf{time}^\#$ operator preserves the progress of its input stream.

Definition 6.14 (Semantics of the Abstract Operator $\mathbf{lift}^\#$ [CHL+18])**.** The abstract operator $\mathbf{lift}^\# \colon (\mathbb{D}^n \rightarrowtail \mathbb{D}) \to (\mathcal{P}^n_\mathbb{D} \to \mathcal{P}_\mathbb{D})$ is given by $\mathbf{lift}^\#(f)(s_1, \ldots, s_n) := z$. The function f must not generate new events, i. e. , must fulfil $f(\perp, \ldots, \perp) = \perp$. The abstract monitoring stream z being defined as follows:

$$z(t) = \begin{cases} f(s_1(t), \ldots, s_n(t)) & \text{if } s_1(t) \neq ?, \ldots, s_n(t) \neq ?, \\ ? & \text{otherwise.} \end{cases}$$

The $\mathbf{lift}^\#$ operator lifts a function f on values to a function on streams by applying f to the stream's values for every timestamp. Again the only extension compared to the operator on streams is the handling of the progress: The progress of z is the minimal progress of all input streams because an event on any of the input streams at timestamp t can directly affect the output stream at timestamp t.

Definition 6.15 (Semantics of the Abstract Operator $\mathbf{last}^\#$ [CHL+18])**.** The abstract operator $\mathbf{last}^\# \colon \mathcal{S}_\mathbb{D} \times \mathcal{S}_{\mathbb{D}'} \to \mathcal{S}_\mathbb{D}$ is given by $\mathbf{last}^\#(v, r) := z$ with the abstract monitoring stream z being defined as follows:

$$z(t) = \begin{cases} d & \text{if } t \in T(r) \wedge \exists\, t' < t \colon \mathit{isLast}(t, t', v, d), \\ \bot & \text{if } r(t) = \bot \wedge \mathit{defined}(z, t), \\ & \text{or } \mathit{uninitialised}(v, t), \\ ? & \text{otherwise,} \end{cases}$$

with the auxiliary definitions:

$$\begin{aligned} \mathit{isLast}(t, t', v, d) &:= v(t') = d \wedge \forall\, t'' \colon t' < t'' < t \Rightarrow v(t'') = \bot, \\ \mathit{defined}(z, t) &:= \forall\, t' < t \colon z(t') \neq ? \text{ and} \\ \mathit{uninitialised}(v, t) &:= \forall\, t' < t \colon v(t') = \bot. \end{aligned}$$

⌟

The predicate $\mathit{isLast}(t, t', v, d)$ for two timestamps $t, t' \in \mathbb{T}$, a stream $v \in \mathcal{S}_\mathbb{D}$ and a value $d \in \mathbb{D}$ holds if (t', d) is the last event on v until t. (*isLast* is the same as in Definition 3.27 for **last** from Section 3.2.3.) The predicate $\mathit{defined}(z, t)$ for a stream $z \in \mathcal{S}_\mathbb{D}$ and a timestamp $t \in \mathbb{T}$ holds if z is defined at least until t (exclusive). The predicate $\mathit{uninitialised}(v, t)$ for a stream $v \in \mathcal{S}_\mathbb{D}$ and a timestamp $t \in \mathbb{T}$ holds if v has no event at least until t (exclusive).

The **last** operator interprets the events on its input stream v as values and the events on r as triggers. It outputs an event with the previous value on v for every event on r. Again this only differs from the **last** operator on streams in the handling of the progress. The addition of the third case, which outputs ?, makes it necessary to explicitly state when $z(t)$ is $\bot$.

With the definitions of the abstract operators $\mathbf{unit}^\#$, $\mathbf{time}^\#$, $\mathbf{lift}^\#$ and $\mathbf{last}^\#$ together with the derived operators from Section 3.3 we can now handle timestamp-conservative TeSSLa specifications. The following example reconsiders Example 3.73 (Counting) from Section 3.5.3 using abstract monitoring streams instead of monitoring streams. Note how the online monitoring in the form of a stepwise refinement still works the same as in the original example.

Example 6.16 (Counting With Abstract Monitoring Streams)**.** Let $x \in \mathcal{R}_{\mathbb{U}}$ be a free input stream and $\ell, i, z \in \mathcal{R}_{\mathbb{N}}$ be bound derived streams given by the following specification:

$$\ell = \mathbf{last}^{\#}(z, x)$$
$$i = \mathbf{lift}^{\#}(inc)(\ell)$$
$$z = \mathsf{merge}^{\#}(i, 0)$$

The function $inc\colon \mathbb{Z} \to \mathbb{Z}$ increments an integer, i. e. $inc(i) = i + 1$ for any $i \in \mathbb{Z}$.

The visualisation in Figure 6.3 shows the abstract monitoring semantics on the left and the monitoring semantics already shown and discussed in Example 3.73 on the right in grey.

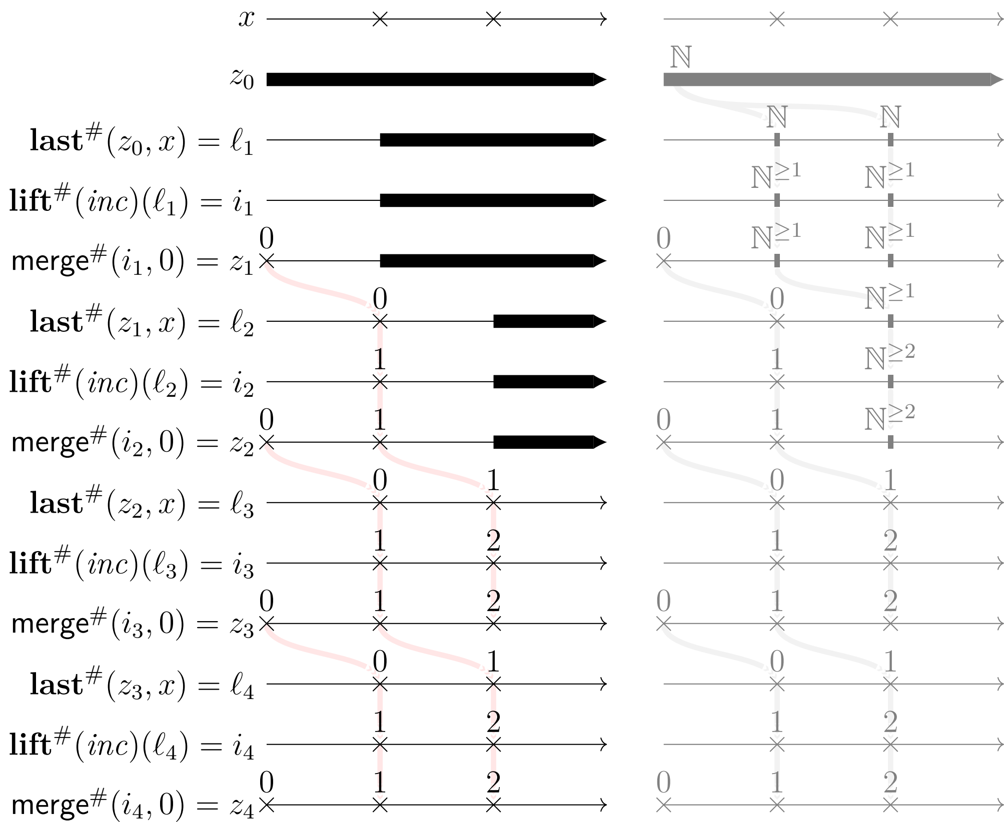


Figure 6.3.: Comparison of the abstract monitoring semantics applied to the specification from Example 6.16 on the left and the monitoring semantics already shown and discussed in Example 3.73 on the right in grey.

Because of $z_4 = z_3$ we reached the fixed point and can conclude $z = z_3$. ⌟

The progress is always exclusive in the above example. Example 6.22 illustrates that inclusive progress is needed, too, especially for the **delay** operator to be used in recursive equations.

6.2.1. Delay

In the case of the **delay** operator, the abstraction becomes more complicated. If one would define the abstract operator based on the operator on streams like we did for the other operators, the resulting abstract $\mathbf{delay}^{\#}$ operator would not work as expected.

Example 6.17 ($\mathbf{delay}^{\#}$ introduces additional fixed points)**.** Consider the following TeSSLa specification known from Example 3.78 (Period) in Section 3.5.3:

$$
\begin{aligned}
c &= \mathsf{const}(2, z)\\
d &= \mathbf{delay}(c)\\
z &= \mathsf{merge}(d, \mathbf{unit})
\end{aligned}
$$

Now assume the streams $z, d \in \mathcal{R}_{\mathbb{U}}$ and $c \in \mathcal{R}_{\mathbb{R}}$ being abstract monitoring streams. Then we would start with an initial $z_0 = \langle (0, _) \rangle$, i. e. the stream without any progress. Next we would compute $\mathsf{const}^{\#}(2, z_0) = \langle (0, _) \rangle = c_1$ which would be again the stream without any progress because $\mathbf{lift}^{\#}$ preserves the progress. Now $\mathbf{delay}^{\#}(c_1)$ does not have the information that all events on c_1 carry the data value 2 because we do not have any information about how the stream continues after its progress available in the abstract monitoring streams. ⌟

One could try to enrich the abstraction with more information, but while this seems simple for this example, it might become challenging in the general case. The main issue why the $\mathbf{delay}^{\#}$ cannot produce any progress larger than its input progress is because every event on its input can cancel and override the currently active delay. We thus define the $\mathbf{delayR}^{\#}$ operator with an additional input stream called reset: The $\mathbf{delayR}^{\#}$ only accepts new delays if there is a simultaneous event on the reset stream or the output stream. In other words, new delays are only accepted at the timestamp when the current delay has been expired or was cancelled by an event on the reset stream. With this adjustment, we know that there will be no relevant event on the input until the current delay is over, and the $\mathbf{delayR}^{\#}$ can generate enough progress to drive the recursion.

Using **delayR** instead of **delay** does not limit the expressiveness (see Lemma 6.20 below), but solves the issue of multiple fixed points due to the abstraction (see Theorem 6.27 below): Similar to the monitoring semantics defined in Definition 3.67

in Section 3.5.2, the abstract monitoring semantics will be defined as the least fixed point of the equation system later in Definition 6.25. Without the additional reset argument of the **delayR** operator, this fixed point in the abstract monitoring semantics would not be unique. An abstraction preserves all fixed points, i. e. every fixed point in the concrete case is a fixed point in the abstraction. However, this is not true the other way around: The abstraction might introduce additional fixed points, as shown in the example above: The minimal fixed point would be the empty stream, but there would be many additional fixed points.

Next, we first define the new **delayR** operator on streams and reason that while it might be less comfortable to use, it does not affect the expressiveness of TeSSLa:

Definition 6.18 (Semantics of the Operator **delayR**)**.** The operator **delayR** with the signature $\mathcal{S}_{\mathbb{T}\setminus\{0\}} \times \mathcal{S}_{\mathbb{U}} \to \mathcal{S}_{\mathbb{U}}$ is given by $\mathbf{delayR}(d, r) := z$ with the stream z being defined as follows:

$$z(t) = \begin{cases} \square & \text{if } \exists\, t' < t\colon \mathit{set}(d, t, t') \wedge \mathit{setable}(z, r, t') \wedge \mathit{noreset}(r, t', t), \\ \bot & \text{otherwise,} \end{cases}$$

with the following auxiliary functions

$$\begin{aligned} \mathit{set}(d, t, t') &:= d(t') = t - t', \\ \mathit{setable}(z, r, t') &:= z(t') = \square \vee r(t') = \square \text{ and} \\ \mathit{noreset}(r, t, t') &:= \forall\, t''\colon t < t'' < t' \Rightarrow r(t'') = \bot. \end{aligned}$$

⌟

The predicates *set* and *noreset* are the same as in the definition of **delay** in Definition 3.28 from Section 3.2.3. The additional predicate $\mathit{setable}(z, r, t')$ for two streams $z, r \in \mathcal{S}_{\mathbb{U}}$ and a timestamp $t' \in \mathbb{T}$ holds if z or r has an event at timestamp t'. Note that we only consider those t', which are strictly smaller than the current timestamp t. That allows us to use $z(t')$ to define $z(t)$.

The operator takes a delay stream d and a reset stream r. It emits a unit event in the resulting stream after the delay passes. Every event on the reset stream resets any delay. New delays can only be set together with a reset event or an emitted output event. The 0 is not allowed as an event's value on the delay input. In the traditional **delay** operator, the 0 was used to reset the delay without setting a new one, but in this **delayR** operator, an event on the reset stream is used for this purpose.

We now consider TeSSLa specifications using **delayR** instead of **delay**:

Definition 6.19 (Progressing TeSSLa)**.** We call a TeSSLa specification *progressing* if it only uses the operators **unit**, **lift**, **last** and **delayR** and operators derived from these.

⌟

The traditional **delay** operator without the reset parameter and any operator derived from it are explicitly not allowed in progressing TeSSLa specifications.

The statements made in Lemma 3.99 (Expressiveness of TeSSLa) in Section 3.6 still holds for progressing TeSSLa:

Lemma 6.20 (Expressiveness of Progressing TeSSLa)**.** *For a function $f\colon \mathcal{S}_{\mathbb{D}}^k \to \mathcal{S}_{\mathbb{D}}^n$ on monitoring streams there exists a progressing TeSSLa specification φ such that $\hat{\boldsymbol{f}}_\varphi \equiv f$ iff*

a) f is Scott-continuous and preserves full knowledge,
b) f has maximal refinement, and
c) f is future independent.

Proof. The construction in the proof of Lemma 3.99 can be slightly adjusted to use **delayR** instead of **delay**: Replace the line

$$d = \mathbf{delay}(\mathbf{lift}(\tilde{u})(m))$$

with the new line

$$d = \mathbf{delayR}(\mathbf{lift}(\tilde{u})(m), \mathsf{merge}(\mathbf{unit}, x_1, \dots, x_k)).$$

Since the entire specification contains only one **delayR** operator and all the other operators are timestamp conservative, we know that there can only be timestamps at time 0, coming from the external streams $x_1, \dots, x_k$ or at times where the **delayR** generated them. All this is covered by this new usage of **delayR**. □

As a consequence of Lemma 6.20 shown above, Theorem 3.102 (Expressiveness of TeSSLa) from Section 3.6 holds for progressing TeSSLa, too.

Although a single **delayR** operator is sufficient to gain the full expressiveness for every TeSSLa specification as shown in the above proof, this might not be the typical use of **delay** operators in TeSSLa specifications. Typical applications use the **delay** operator either in a non-recursive fashion as a timer that generates events after a timeout or in recursive expressions to generate repeated event patterns. In the first case of non-recursive specifications, the **delay** can be replaced with **delayR** simply by using the existing input stream as reset stream, too. The second case is elaborated in Example 6.22 (Period With Abstract Monitoring Streams) and Example 6.23 (Variable Frequency Period With Abstract Monitoring Streams) below.

With the expressiveness of progressing TeSSLa being established we can now move on and abstract the new **delayR** operator:

Definition 6.21 (Semantics of the Abstract Operator $\mathbf{delayR}^{\#}$ [CHL+18])**.** The abstract operator $\mathbf{delayR}^{\#}\colon \mathcal{R}_{\mathbb{T}\setminus\{0\}} \times \mathcal{R}_{\mathbb{U}} \to \mathcal{R}_{\mathbb{U}}$ is defined by $\mathbf{delayR}^{\#}(d,r) := z$ with the abstract monitoring stream z being defined as follows:

$$z(t) = \begin{cases} \Box & \text{if } \exists\, t' < t\colon \mathit{set}(d,t,t') \wedge \mathit{setable}(z,r,t') \wedge \mathit{noreset}(r,t',t), \\ \bot & \text{if } \mathit{defined}(z,t) \\ & \text{and } \forall\, t' < t\colon \mathit{unset}(d,t,t') \vee \mathit{unsetable}(z,r,t') \vee \mathit{reset}(r,t',t), \\ ? & \text{otherwise,} \end{cases}$$

with the following auxiliary definitions:

$$\begin{aligned} \mathit{set}(d,t,t') &:= d(t') = t - t', \\ \mathit{setable}(z,r,t') &:= z(t') = \Box \vee r(t') = \Box, \\ \mathit{noreset}(r,t,t') &:= \forall\, t''\colon t < t'' < t' \Rightarrow r(t'') = \bot, \\ \mathit{defined}(z,t) &:= \forall\, t' < t\colon z(t') \neq ?, \\ \mathit{unset}(d,t,t') &:= d(t') \neq t - t' \wedge d(t') \neq ?, \\ \mathit{unsetable}(z,r,t') &:= z(t') = \bot \wedge r(t') = \bot, \text{ and} \\ \mathit{reset}(r,t,t') &:= \exists\, t''\colon t < t'' < t' \Rightarrow r(t'') = \Box. \end{aligned}$$

⌟

The predicates *set*, *setable* and *noreset* are the same as in Definition 6.18 for **delayR** from Section 6.2.1. The predicate *defined* is the same as in Definition 6.15 for $\mathbf{last}^{\#}$ from Section 6.2. The predicates *unset*, *unsetable* and *reset* are the opposites of *set*, *setable* and *noreset*, respectively. However, they are not their negation because they disallow the unknown case, i. e. they require the involved streams to have sufficient progress to make their statement: The predicate $\mathit{unset}(d,t,t')$ for a stream $d \in \mathcal{R}_{\mathbb{T}\setminus\{0\}}$ and two timestamps $t, t' \in \mathbb{T}$ holds if the stream d does not contain an event at timestamp t' with a delay pointing to the current timestamp t. The predicate $\mathit{unsetable}(z,r,t')$ for two streams $z, r \in \mathcal{R}_{\mathbb{U}}$ and a timestamp $t' \in \mathbb{T}$ holds if both streams z and r do not have an event at timestamp t', i. e. event on d at t' are ignored. The predicate $\mathit{reset}(r,t,t')$ for a stream $r \in \mathcal{R}_{\mathbb{U}}$ and two timestamps $t, t' \in \mathbb{T}$ holds if there is an event on r between t' and the current timestamp t (both exclusive).

The operator takes a delay stream d and a reset stream r. It emits a unit event in the resulting stream after the delay passes. Every event on the reset stream resets any delay. New delays can only be set together with a reset event or an emitted output event. Again, the main addition is the third case producing ?, which requires an explicit definition when the output is $\bot$.

With the $\mathbf{delayR}^{\#}$ being defined, we can now come back to the specification known from Example 3.78 (Period) in Section 3.5.3 and show how computing the recursive specification step by step works:

Example 6.22 (Period With Abstract Monitoring Streams)**.** Let $z, d \in \mathcal{R}_{\mathbb{U}}$ and $c \in \mathcal{R}_{\mathbb{R}}$ be derived streams given by the following specification:

$$c = \mathsf{const}^{\#}(2, z)$$
$$d = \mathbf{delayR}^{\#}(c, \mathbf{unit}^{\#})$$
$$z = \mathsf{merge}^{\#}(d, \mathbf{unit}^{\#})$$

The diagram in Figure 6.4 shows the abstract monitoring semantics on the left, and the diagram from Example 3.78 on the right in grey for comparison.

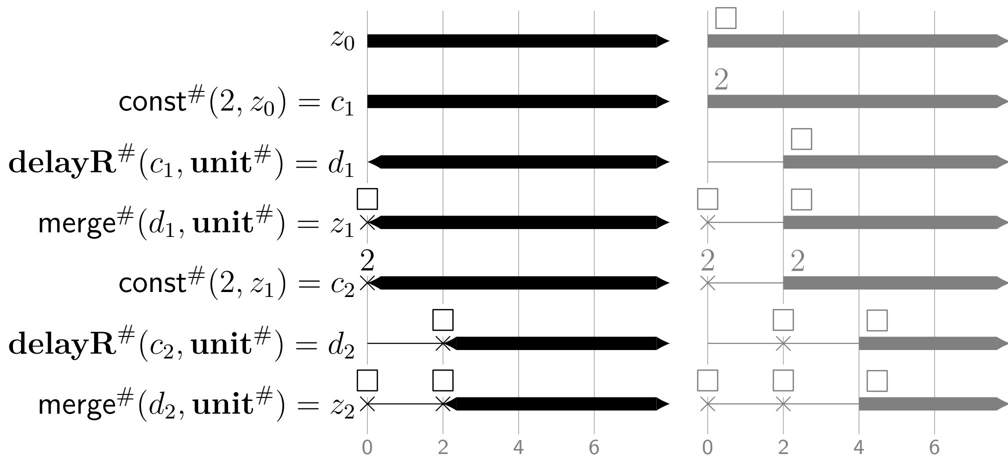


Figure 6.4.: Comparison of the abstract monitoring semantics on the left, and the monitoring semantics from Example 3.78 on the right in grey for comparison.

The diagram on the left uses the abstract monitoring semantics, and the grey diagram on the right uses the monitoring semantics and **delay** instead of **delayR**. The main difference is that the monitoring streams on the right can encode that all possible events on c_1 have the value 2. The abstract monitoring streams cannot encode this information, and hence d_1 only has inclusive progress of 0 because, without the information that all events on c_1 have a value of 2, we only know for sure that there cannot be an event at timestamp 0. After merging with the $\mathbf{unit}^{\#}$ stream and applying the $\mathsf{const}^{\#}$ operator again we get c_2. Now the **delayR** has the additional knowledge that there will be no interruption until it outputs the event at timestamp 2 because its reset stream $\mathbf{unit}^{\#}$ is fully known and has no events. So the $\mathbf{delayR}^{\#}$ now produces inclusive progress until timestamp 2. This example shows how the additional reset stream input of the **delayR** replaces the information that was lost by representing the monitoring stream as an abstract monitoring stream. ⌟

We can make two observations from the above example:

1. z_1 has fewer progress than in the concrete case. We will show after the following example that the individual operators are perfect abstractions of their concrete counterparts. However, this property is not closed under composition, i. e. the composition of two TeSSLa operators can be abstracted by composing the corresponding two abstract operators, but this is not necessarily a perfect abstraction of the composition. We will discuss this further in Example 6.29.

2. While c_1 has an exclusive progress of 0, d_1 has an inclusive progress of 0. This inclusive progress is needed for the **merge** to generate an initial event in z_1. Without the capability to express inclusive progress, we would reach a fixed point after one iteration, the fully unknown stream z_0.

In the above example the reset was a constant stream with infinite progress. The following more complex example demonstrates how a delay can be interrupted using an external input stream as reset stream. The specification is known from Example 3.80 (Variable Frequency Period) in Section 3.5.3:

Example 6.23 (Variable Frequency Period With Abstract Monitoring Streams). Let $x \in \mathcal{R}_{\mathbb{R}^+}$ be a free input stream and $\ell, z \in \mathcal{R}_{\mathbb{R}^+}$ and $d \in \mathcal{R}_{\mathbb{U}}$ be derived streams given by the following specification:

$$d = \mathbf{delayR}^{\#}(z, x)$$
$$\ell = \mathbf{last}^{\#}(x, d)$$
$$z = \mathsf{merge}^{\#}(x, \ell)$$

Figure 6.5 shows the abstract monitoring semantics on the diagram from Example 3.80 on the right in grey. The grey diagram on the right uses the monitoring semantics and **delay** instead of **delayR**. As in Example 3.80 the red 1.5 and the black 1.5 are the exact same value. They are only coloured differently to indicate the source of the event.

The main difference to the previous Example 6.22 is that the delay can now be interrupted by events on the input stream x. Hence the input stream x is used as reset stream for the $\mathbf{delayR}^{\#}$. The stream d_3 only has inclusive progress until the timestamp 5 because there is an event with that timestamp on x. On the right, the **delay** considers all possible values of the events on z_3. It generates progress until timestamp 6.5 because the possible values on z_2 switch from 3 to 1.5 after the event on the input stream x. So again, the additional reset stream compensates for the additional information available on the monitoring streams on the right. ⌟

As a direct consequence of the definition of the abstract TeSSLa operators above we can conclude:

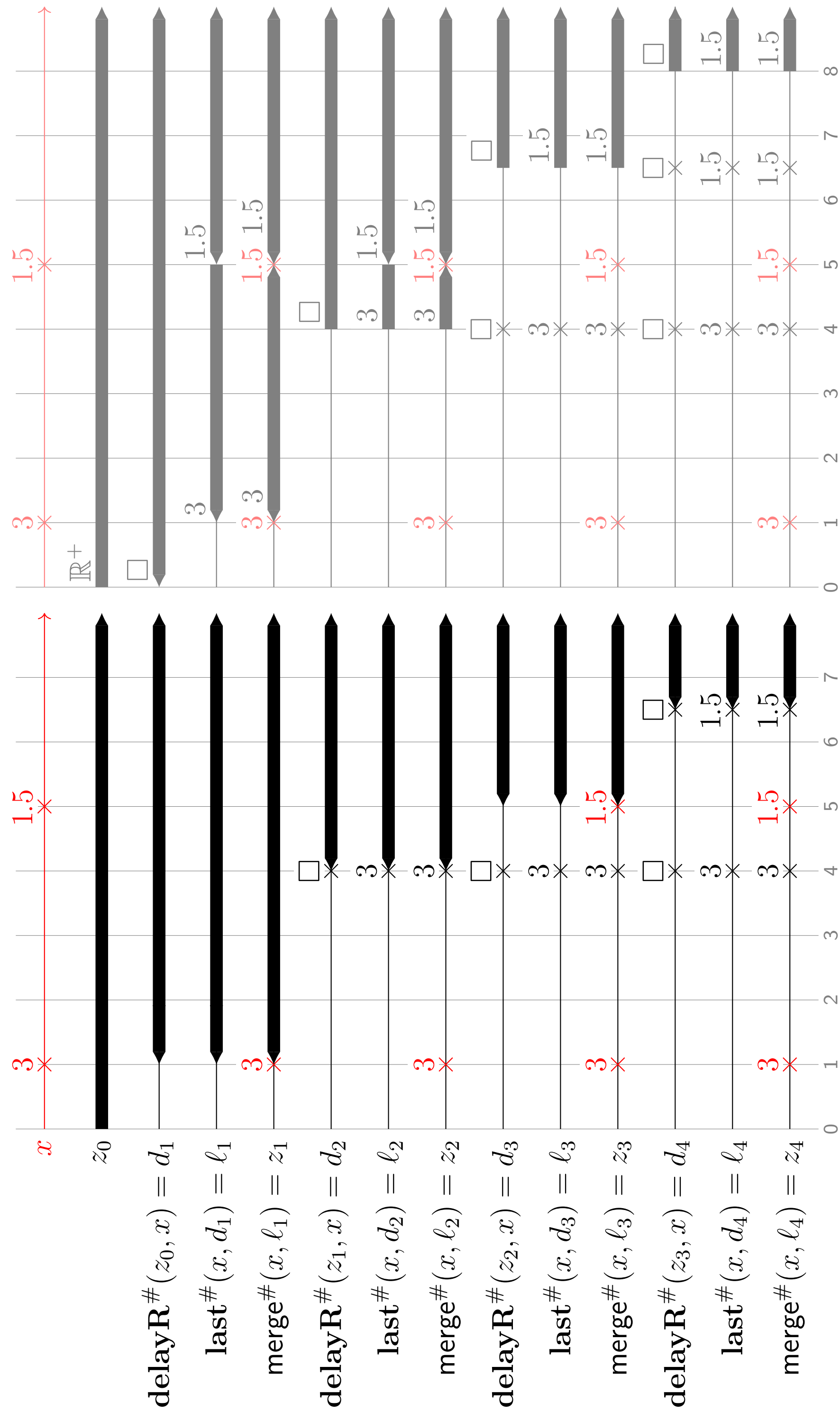


Figure 6.5.: Variable frequency period with abstract monitoring streams. For comparison the already known monitoring semantics is reproduced above in grey.

Lemma 6.24 (Perfectness of the Abstract TeSSLa Operators)**.** *The abstract TeSSLa operators on abstract monitoring streams* $\mathbf{unit}^\#$*,* $\mathbf{time}^\#$*,* $\mathbf{lift}^\#$*,* $\mathbf{last}^\#$ *and* $\mathbf{delayR}^\#$ *are a perfect abstraction of their counterparts on monitoring streams* **unit***,* **time***,* **lift***,* **last** *and* **delayR***.*

The abstract $\mathbf{lift}^\#$ operator is a perfect abstraction of the **lift** operator if we do not consider the lifted function f and are looking for an abstraction that works for any lifted function. If we consider $\mathbf{lift}(f)$ for a concrete function $f\colon \mathbb{D}^n \rightarrowtail \mathbb{D}$ on the data domain then $\mathbf{lift}^\#(f)$ might not be a perfect abstraction. For example consider the function $f\colon \mathbb{D} \rightarrowtail \mathbb{D}$ with $f(x) = \bot$ which removes all events. In this case, $\mathbf{lift}(f)$ would be $\langle(\infty, _)\rangle$, i. e. the empty stream with infinite progress because there will never be an event on this stream. However, $\mathbf{lift}^\#(f)$ does not produce more progress than the input stream already has.

While this seems easily fixable in the above example, the general case can become arbitrarily complex and involves static analysis of the function f. For that reason, we stick with $\mathbf{lift}^\#$ which is the perfect abstraction of **lift** for the general case of unknown lifted functions.

6.3. Abstract TeSSLa Semantics

In the above examples, the abstract TeSSLa operators were already used in complex specifications. The following definition formally defines the abstract monitoring semantics analogously to Definition 3.67 (TeSSLa Monitoring Semantics) from Section 3.5.2. This semantics and the results regarding its fixed point are introduced as TeSSLa semantics in [CHL+18] and are reproduced here with minor adjustments for the sake of completeness.

Definition 6.25 (TeSSLa Abstract Monitoring Semantics [CHL+18])**.** Let φ be a progressing TeSSLa specification with k free streams $\boldsymbol{y} = (y_1, y_2, \ldots, y_k)$ and n bound streams $\boldsymbol{z} = (z_1, z_2, \ldots, z_n)$. The equations $z_i = f_i(\boldsymbol{y})(\boldsymbol{z})$ for $1 \leq i \leq n$ of the specification φ can be applied to abstract monitoring streams by replacing f_i with $f_i^\#$. In combination we get the fixed point equation

$$\boldsymbol{z} = \boldsymbol{f}^\#(\boldsymbol{y})(\boldsymbol{z})$$

with

$$\boldsymbol{f}^\#\colon \mathcal{R}_{\mathbb{D}}^k \times \mathcal{R}_{\mathbb{D}'}^n \to \mathcal{R}_{\mathbb{D}'}^n.$$

The *abstract monitoring semantics* of φ are a function $\boldsymbol{f}_\varphi^\#\colon \mathcal{R}_{\mathbb{D}}^k \to \mathcal{R}_{\mathbb{D}'}^n$ given as the least fixed point of $\boldsymbol{f}^\#$ over the bound streams $\boldsymbol{z}$:

$$\boldsymbol{f}_\varphi^\#(\boldsymbol{y}) := \mu \boldsymbol{f}^\#(\boldsymbol{y}).$$

┘

Next, we show that Lemma 3.81 (Construction of the Least Fixed Point) from Section 3.5.4 can be modified such that it applies to the least fixed point used in the above definition of the abstract monitoring semantics and states that this fixed point exists and can be constructed:

Lemma 6.26 (Construction of the Least Fixed Point [CHL+18])**.** *The least fixed point used in the definition of the abstract TeSSLa monitoring semantics exists and can be constructed:*

$$\boldsymbol{f}^{\#}_{\varphi}(\boldsymbol{y}) = \mu \boldsymbol{f}^{\#}(\boldsymbol{y}) = \bigvee \{ (\boldsymbol{f}^{\#}(\boldsymbol{y}))^n (\langle (0, _) \rangle, \langle (0, _) \rangle, \dots, \langle (0, _) \rangle) \mid n \in \mathbb{N} \}$$

Proof. In order to apply the proof of the original Lemma 3.81 we make sure that several concepts used in it still work on the abstract monitoring stream and the abstract operators:

- The abstract TeSSLa operators $\textbf{unit}^{\#}$, $\textbf{time}^{\#}$, $\textbf{lift}^{\#}$, $\textbf{last}^{\#}$ and $\textbf{delayR}^{\#}$ are Scott-continuous in the same way as their concrete counterparts.
- Hence, for a given tuple of input streams $\boldsymbol{y} \in \mathcal{S}_{\mathbb{D}}$ the function $\boldsymbol{f}^{\#}(\boldsymbol{y})$ is Scott-continuous.
- The partial order $(\mathcal{R}_{\mathbb{D}}, \sqsubseteq)$ is a dcpo with the least element $(0, _)$, i. e. the empty stream without any progress, in the same way as $(\mathcal{P}, \sqsubseteq)$ is a dcpo with the least element $\mathcal{S}_{\mathbb{D}}$, i. e. the set of all possible streams.

□

Based on this we can now show that Theorem 3.82 (Uniqueness of the Fixed Point in the Monitoring Semantics) from Section 3.5.4 can be adjusted to hold on the abstract TeSSLa monitoring semantics, too:

Theorem 6.27 (Uniqueness of the Fixed Point [CHL+18])**.** *If a progressing TeSSLa specification φ is well-formed, then the fixed point $\mu \boldsymbol{f}^{\#}(\boldsymbol{y})$ used in the abstract TeSSLa monitoring semantics is unique.*

Proof. The proof of the original Theorem 3.82 can be applied to the adjusted theorem by replacing the mentioned functions and relations with their abstract counterparts and replacing **delay** with **delayR**. We only need to reconsider the final argument regarding the nature of the operators $\textbf{last}^{\#}$ and $\textbf{delayR}^{\#}$: Both operators are defined in a way that they refine their input further on pre-fixed points until the fixed point is reached: An output event at timestamp t is defined in both operators independent of their input streams at timestamp t. Due to the addition of the reset input stream to the $\textbf{delayR}^{\#}$ operator, this argument still holds. □

See Examples 6.16, 6.22 and 6.23 for illustrations how $\textbf{last}^\#$ and $\textbf{delay}^\#$ generate progress if there is enough progress on trigger or reset, respectively.

6.3.1. Quality of the TeSSLa Abstract Monitoring Semantics

Theorem 6.28 (TeSSLa Abstract Monitoring Semantics is an Abstraction)**.** *The TeSSLa abstract monitoring semantics $\boldsymbol{f}^\#_\varphi \colon \mathcal{R}^k_\mathbb{D} \to \mathcal{R}^n_{\mathbb{D}'}$ is an abstraction of the TeSSLa monitoring semantics $\hat{\boldsymbol{f}}_\varphi \colon \mathcal{P}^k_\mathbb{D} \to \mathcal{P}^n_{\mathbb{D}'}$ for any progressing TeSSLa specification φ with k free stream variables.*

Proof. Let φ be a progressing TeSSLa specification with k free stream variables. From Lemma 6.10 (Galois Connection for Abstract Monitoring Streams) in Section 6.1 we know that α and γ are a Galois connection between $(\mathcal{P}_\mathbb{D}, \sqsupseteq)$ and $(\mathcal{R}_\mathbb{D}, \sqsupseteq)$. This property is closed under Cartesian product such that we have a Galois connection between $(\mathcal{P}^k_\mathbb{D}, \sqsupseteq)$ and $(\mathcal{R}^k_\mathbb{D}, \sqsupseteq)$, too. From Lemma 6.24 (Perfectness of the Abstract TeSSLa Operators) in Section 6.2.1 we know that the abstract TeSSLa operators are abstractions or their concrete counterparts, so for an abstract TeSSLa operator $f^\#_i$ and its concrete counterpart $\hat{f}_i$ we have

$$\forall\, s \in \mathcal{R}_\mathbb{D} \colon f_i(\gamma(s)) \sqsupseteq \gamma(f^\#_i(s)).$$

This property is compositional and closed under the Cartesian product, and by Lemma 6.26 we know that for any tuple of input streams $\boldsymbol{s} \in \mathcal{R}^k_\mathbb{D}$ we can construct $\boldsymbol{f}^\#_\varphi(\boldsymbol{s})$ through a Kleene chain of Cartesian products of TeSSLa operators. Hence we get

$$\forall\, \boldsymbol{s} \in \mathcal{R}^k_\mathbb{D} \colon \hat{\boldsymbol{f}}_\varphi(\gamma(s)) \sqsupseteq \gamma(\boldsymbol{f}^\#_\varphi(\boldsymbol{s})). \qquad \square$$

Note that we only use the fact that the operators are an abstraction, not a perfect abstraction. We have already seen in Example 6.22 that the perfectness of the abstraction is not closed under composition. The following example illustrates further cases where the abstract monitoring semantics is not a perfect abstraction:

Example 6.29 (Abstract TeSSLa Monitoring Semantics is Not a Perfect Abstraction)**.** Consider the following TeSSLa specification with the input stream $x \in \mathcal{S}_\mathbb{U}$:

$$y = \mathsf{count}(x)$$
$$z = \mathsf{filter}(y, y \leq 3)$$

In this example, we use the implicit conversions and applications defined in Section 3.3.7, which allow us to write $y \leq 3$ for an slift which applies the comparison with 3 to every event's data value on the stream y.

This specification counts the number of events on the input stream and lets the first three events pass. All later events will be filtered out. As a result, after the third event on the output stream, we can be sure that there will be no more events on the output stream z. The monitoring semantics reflect this by returning a fully known stream after reading the third input event because they consider all possible continuations of the partially known input streams. The abstract monitoring semantics, however, cannot consider this because the abstract monitoring streams contain no information about possible events after the current progress of the stream. As a result, the abstract monitoring semantics do not generate more progress on the output stream than what is already available on the input stream. ⌟

Although it might be possible for a given TeSSLa specification to manually come up with a perfect abstraction for the entire specification, this is generally not feasible because it would require a complex analysis of the entire specification. For practical implementations, it is much better to be compositional than perfect. As one can see in the above example, we only lose precision regarding the progress, and we gain a set of simple abstract operators which can be used as basic building blocks for every TeSSLa specification.

Before we continue in the next chapter using the abstract monitoring semantics to build an imperative asynchronous execution model, we first consider some final aspects about the relation of the TeSSLa monitoring semantics and the abstract TeSSLa monitoring semantics:

The following definition is the abstract counterpart of Definition 3.84 (Preserving Full Knowledge) from Section 3.5.5. Monitoring streams are called fully known if they are singleton sets because then no further refinement is possible. We call abstract monitoring streams fully known if their progress is infinite. In terms of the prefix relation, those streams are maximal.

Definition 6.30 (Preserving Full Knowledge on Abstract Monitoring Streams)**.** We say a function $f\colon \mathcal{R}_{\mathbb{D}}^k \to \mathcal{R}_{\mathbb{D}}^n$ over tuples of abstract monitoring streams *preserves full knowledge* if it maps tuples of fully known streams to tuples of fully known streams, i. e. if the input is a tuple of abstract monitoring streams with infinite progress then the output tuple consists only of abstract monitoring streams with infinite progress, too. ⌟

Recall that every stream can be seen as a monitoring stream with full knowledge. We defined the implicit conversion, which converts a stream into a monitoring stream by making it a singleton set. In the same way, we define the following implicit conversion for abstract monitoring streams: Let $s \in \mathcal{S}_{\mathbb{D}}$ be a stream. We then implicitly add $(\infty, _)$ if s is finite and used as an abstract monitoring stream. We

get $\gamma(s) = \{s\}$, i. e. if we use a stream as an abstract monitoring stream, its concrete counterpart is a fully known stream.

The following lemma is the abstract counterpart of Lemma 3.85 (Relation Between TeSSLa Monitoring Semantics and TeSSLa Semantics) from Section 3.5.5 and makes the relation between the abstract monitoring semantics and the monitoring semantics and hence the quality of the abstraction more precise: For all streams, i. e. monitoring streams with full knowledge, the abstract monitoring semantics behaves the same as the monitoring semantics.

Lemma 6.31 (Relation Between the Abstract TeSSLa Monitoring Semantics and the TeSSLa Semantics)**.** *Let φ be a well-formed progressing TeSSLa specification with the semantics $\boldsymbol{f}_\varphi$ and the abstract monitoring semantics $\boldsymbol{f}^\#_\varphi$. We then have*

$$\forall\, \boldsymbol{a} \in \mathcal{S}^n_{\mathbb{D}} \colon \boldsymbol{f}^\#_\varphi(\boldsymbol{a}) = \boldsymbol{f}_\varphi(\boldsymbol{a}).$$

Proof. The proof of Lemma 3.85 directly carries over to this abstract counterpart of the lemma using the fixed point used to define the TeSSLa abstract monitoring semantics in Definition 6.25 instead of the fixed point from the TeSSLa monitoring semantics in the original proof. □

Note that the main equality in the lemma above only holds with the implicit conversions from streams to fully known abstract monitoring streams.

It directly follows from the above lemma that the abstract TeSSLa semantics preserves full knowledge.

Intuitively the lemma states that the only difference between the TeSSLa operators and the abstract TeSSLa operators is the handling of the progress. If there is enough progress, then they behave the same. This result says something about the quality of this abstraction: The abstraction of the individual operators is perfect, but as demonstrated in Example 6.29 the perfectness of the individual operator's abstraction is not compositional, i. e. the abstraction of the TeSSLa monitoring semantics is no longer perfect. With this lemma, we know that it only differs from the perfect abstraction regarding the progress on not yet fully known input streams, not regarding the events. So we can safely use this abstract monitoring semantics because eventually, we get the exact result.

6.3.2. Equivalence of TeSSLa Specifications

While monitoring-equivalence and equivalence of TeSSLa specification is the same property, the equivalence of TeSSLa specifications regarding the abstract monitoring

semantics is weaker: Abstract monitoring equivalence implies equivalence. As a direct consequence of Lemma 6.31 we get:

Corollary 6.32 (Equivalence of Abstract TeSSLa Specifications)**.** *If two TeSSLa specifications φ and ψ are equivalent with regard to their abstract monitoring semantics, i. e.* $\boldsymbol{f}^{\#}_{\varphi} \equiv f^{\#}_{\psi}$*, then they are monitoring-equivalent and thus equivalent.*

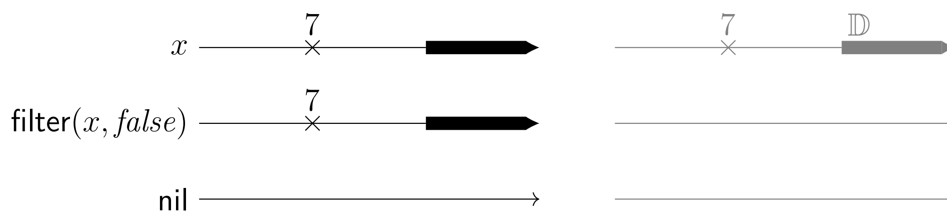
Intuitively equivalence of two TeSSLa specifications with regard to their abstract monitoring semantics means they produce the same output for any input, which includes those streams with infinite progress, which are the ones relevant for the TeSSLa semantics.

Equivalent TeSSLa specifications are not always equivalent with regard to their abstract monitoring semantics. They can have different behaviour regarding the progress in the abstraction. The following example demonstrates a case where equivalent TeSSLa specifications do not behave the same using the abstract monitoring semantics because the progress differs:

Example 6.33 (Equivalent TeSSLa Specifications With Different Progress in the Abstract TeSSLa Monitoring Semantics)**.** We know that for any monitoring stream $x \in \mathcal{S}_{\mathbb{D}}$ we have

$$\mathsf{filter}(x, \mathit{false}) \equiv \mathsf{nil},$$

because the filter removes all elements from x with the condition being always false. The following diagram compares the monitoring semantics and the abstract monitoring semantics:



In the abstract monitoring semantics on the left, the filter only generates as much progress as the input stream x provides. The monitoring semantics on the right, however, considers all possible streams, and because all possible streams never contain any event, the output of the filter is equivalent to nil. ⌟

6.4. Conclusion

This chapter introduced the abstract monitoring semantics on abstract monitoring streams used for the FPGA synthesis discussed in the next chapter. The monitoring

streams are a possibly infinite set of streams, and the monitoring semantics are defined by applying the TeSSLa operators to each of these streams individually. With the straightforward definition of this semantics in Section 3.5 we gained the ability to express individual progress for every monitoring stream which supports asynchronous evaluations. The abstract monitoring semantics is an abstraction of the monitoring semantics that

a) preserves the support for asynchronous evaluation that the monitoring semantics provide, and
b) simplifies the operators to allow implementation on hardware.

The **delay** operator was extended to the **delayR** operator with an additional reset argument. By splitting up the inputs of the **delay** into one stream setting delays and another stream resetting them, we can express the behaviour of **delay** in case of recursive definitions on abstract monitoring streams.

The abstract monitoring semantics cannot preserve the maximal progress provided by the monitoring semantics, but it provides maximal progress per operator. Intuitive, this means that there is only a difference in the derived streams regarding the progress, and only in cases where the input lacks sufficient progress. This sweet spot satisfies the two goals of simple compositional operators that can be implemented on hardware and support for asynchronous evaluation of the flow graph on the hardware.

7 FPGA Synthesis

This chapter introduces another approach how to map the flow graph of a TeSSLa specification onto processing hardware: Chapter 5 presented a linear sequential pipeline of EPUs running in parallel and mapped the flow graph onto this linear pipeline. This chapter covers the synthesis of a TeSSLa specification directly on an FPGA. This synthesis places every operator of the flow graph individually on the FPGA such that all synthesised operators are executed independently in parallel. Chapter 6 introduced the abstract monitoring semantics on abstract monitoring streams as an abstraction of the TeSSLa monitoring semantics that is suited for this parallel approach: The abstraction preserves the ability of the monitoring semantics to produce different progress on the individual streams.

The main difference between the EPU architecture and the synthesis discussed in this chapter is the absence of a global current timestamp. In order to keep the EPU pipeline synchronous, every event is sent through the entire pipeline until it reaches its target. Messages cannot bypass EPUs on the pipeline because otherwise, the EPUs might miss timestamp increments. With the individual progress per stream provided by the abstract monitoring streams, we can utilise the main benefit of an FPGA: It processes multiple data values independently in parallel on different areas of the FPGA. On the other hand, additional synchronisation overhead is required for every operator.

The FPGA synthesis is based on the following principles:

- *Message Passing.* The operators receive, process and pass on timestamps and values along the edges of the flow graph. Data can only be transmitted if the recipient is ready to receive. Otherwise, the sender has to wait. Waiting operators can cause a back pressure along the edges of the flow graph in the opposite direction. Queues are included in the graph to store data and relieve the backpressure for certain paths.
- *Compositional Flow Graphs.* The synthesised monitor consists only of the synthesised operators connected according to the flow graph. There is no central orchestration or scheduling. Every operator is executed purely based on the available input data. The operators and the flow graph are compositional, i. e. every connection between operators directly corresponds to an edge of the flow graph and can be interpreted as an abstract monitoring stream.

- *Asynchronous Evaluation.* The lack of a central orchestration also implies the absence of any global synchronisation. The streams are synchronised locally for every operator. The benefit of this approach is that operations on one path do not necessarily influence independent operations on other paths. This independence can be especially beneficial if different event streams have different event rates. The drawback is the additional need for synchronisation logic in every operator. We will investigate ways to reorganise the flow graph to mitigate this effect in Section 7.5 (Tuplification Optimisation).
- *Logical Timestamps.* The absence of a global orchestration in combination with asynchronous evaluation raises the need to encode the order of events explicitly: The synchronisation of events on different streams is performed based on their logical timestamps. Every event is encoded and passed along the edges of the flow graph in the form of a timestamp and a corresponding data value. With this approach, TeSSLa's ability to insert additional events between existing events with the **delay** operator can be implemented naturally.

The basic idea of evaluating TeSSLa specifications using asynchronous message passing along the specification's flow graph was already studied in [LSS+18, LSS+20] However, these publications use Erlang actors to encode the flow graph in software and do not address hardware synthesis. Further, they use a preliminary version of TeSSLa that does not support recursive equations. This chapter generalises and extends the approach to support cycles in flow graphs and the necessary synchronisation.

As already discussed in the introduction of this thesis, the main difference between TeSSLa and other synchronous languages such as Esterel and Lustre is TeSSLa's support for asynchronous evaluations due to its utilisation of logical timestamps. Synthesising Lustre on hardware [RH91] always follows the idea of a Synchronous evaluation. The continuous time is represented as a sequence of instants, and for every instant, inputs are read, and new outputs are computed. These instants are realised as clock cycles on the hardware. The same holds for Esterel's hardware synthesis [Ber16, HN03]: The evaluation happens in general in a synchronous fashion. The entire specification is evaluated for every instant. The logical timestamps of TeSSLa, on the other hand, can be seen as a higher abstraction from the actual hardware. While Esterel can be seen as a language to program FPGAs, because the Esterel specification describes the timing behaviour of the hardware [HN03], TeSSLa's timestamps are entirely independent of the hardware's clock ticks.

For the purpose of this thesis, an FPGA can be considered an array of programmable logic blocks and programmable interconnects between these logic blocks. [Max04] Figure 7.1 shows a top-down view of such a simple, generic FPGA architecture and Figure 7.2 shows an exemplary logic block: The lookup table (LUT) can be

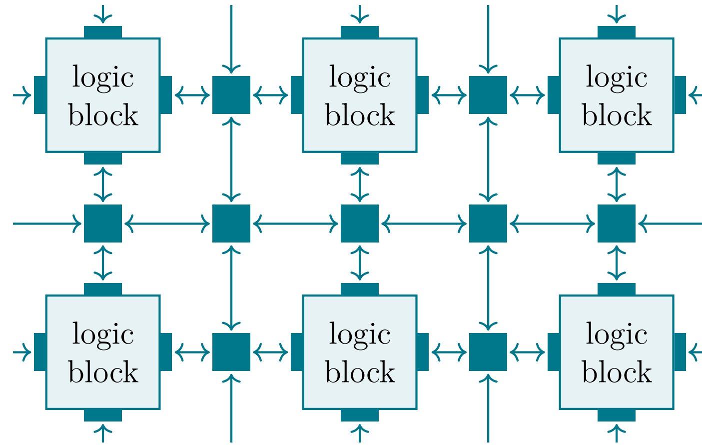


Figure 7.1.: Top-down view of a simple, generic FPGA architecture [Max04, Figure 3-20].

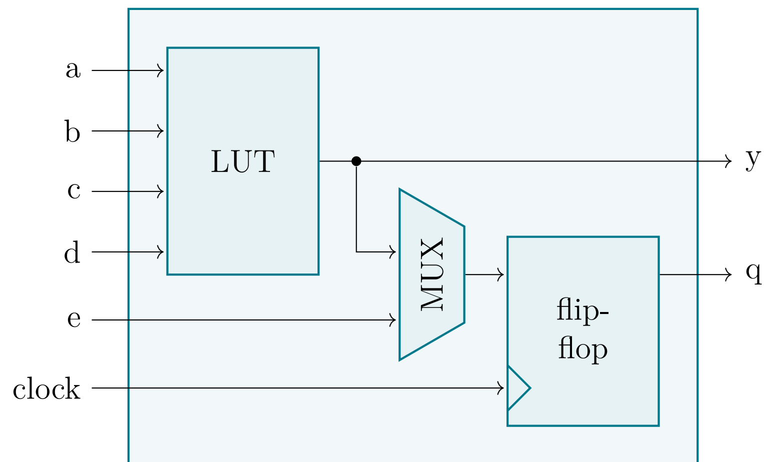


Figure 7.2.: Simplified logic block, diagram based on [Max04, Figure 4-7].

configured to realise an arbitrary 4-ary boolean function, and the flip-flop can store one bit. The multiplexer (MUX) connects the flip-flop's input either with the output of the LUT or an independent input. In modern FPGAs, the design is way more advanced. See [XILa, XILb] for detailed informations about Xilinx' logic blocks.

FPGAs are programmed by describing registers and logical functions connecting their inputs and outputs. On a higher level, arithmetic operations and comparison on integers are available, too. Such a hardware description is then compiled into an FPGA image by tools like Xilinx Vivado [XILe]. For a given clock speed, the logical functions connecting flip-flops must not exceed a certain complexity such that the involved LUTs realising the function are converging fast enough into a stable value until the end of the clock cycle. That also depends on how the functions can be mapped to logic blocks on the physical chip and how long the corresponding physical paths are.

The TeSSLa operators are designed to have explicit memory usage (see Section 3.4) which supports implementations with finite memory. Compared with the synchronous operator functions used for the software compiler and the EPUs, the asynchronous operators introduced in the last section are more complex, because they need to consider inputs of different progress and computer the resulting progress. Unfortunately, the computation of the progress performed by the $\mathbf{delay}^\#$ operator introduced in the previous chapter cannot be implemented with finite memory. Therefore, we define some simpler versions of the $\mathbf{delay}^\#$ and $\mathbf{last}^\#$ operators in Section 7.1. These are no longer perfect abstractions because they do not always generate the maximal possible progress. However, with this slight adjustment, they are much better suited for hardware implementations.

Section 7.2 introduces operator networks as a formal model which abstracts away from the technical details of an FPGA and Section 7.3 describes how TeSSLa specifications are compiled into operator networks.

Section 7.4 brings back the implementation details of actual FPGAs, Section 7.5 adds optimisations that increase throughput and reduce resource utilisation, and Section 7.6 explains the test and integration setup used the evaluate the compiler and the synthesised FPGA images.

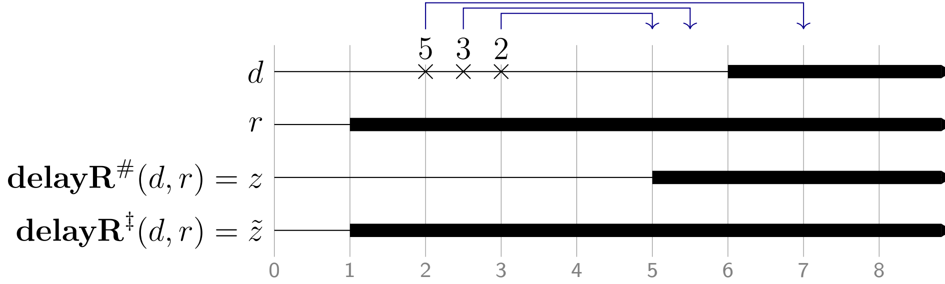
7.1. Finite Memory

Before we discuss how to implement the abstract operators for the FPGA synthesis, we first revisit the $\mathbf{delayR}^\#$ operator defined in the last chapter. The following example demonstrates why we cannot implement this operator with finite memory:

Example 7.1 (**delayR**$^{\#}$ Needs Infinite Memory)**.** Let $d \in \mathcal{R}_{\mathbb{R}>0}$ and $r \in \mathcal{R}_{\mathbb{U}}$ be two abstract input streams. Further, let the derived abstract monitoring stream $z \in \mathcal{R}_{\mathbb{U}}$ be defined by

$$z = \mathbf{delayR}^{\#}(d, r).$$

The following diagram shows a possible evaluation:



The additional stream $\tilde{z}$ is discussed after the definition of **delayR**‡ below. The blue arrows indicate the delays. The **delay** (and thus the **delayR**) operator is designed such that there is always only one active delay. (See the discussion of the alternative mdelay in Section 3.4.5.) However, the reset stream r has fewer progress in the above situation than the delay stream d. In such a situation, it is not (yet) clear if the events on d must be considered or not because there might be a corresponding event on r. The maximal possible progress on the output stream z depends on all possible active delays. We have to consider all possible delays. In the example above the progress is

$$\min\{2 + 5, 2.5 + 3, 3 + 2\} = 5.$$

Note that the progress on z is exclusive: With an event on r at timestamp 3 there could be an event on z at timestamp 5. However, considering all possible delays on d, there is no way how r could be extended to create an event on z with an earlier timestamp. There exists no limit regarding the number of events on d that must be considered to compute the progress of z. Infinite memory is needed to track all the events occurring on d after the progress on r. ⌟

Next, we define a variant of the **delayR**$^{\#}$ operator, the **delayR**‡ operator, which still has all the necessary properties for the considerations in the previous chapter but produces slightly less progress. We do this by restricting the progress of $\tilde{z}$ to the progress of r:

Definition 7.2 (Semantics of the Abstract Operator **delayR**‡)**.** The abstract operator **delayR**‡: $\mathcal{R}_{\mathbb{T}\setminus\{0\}} \times \mathcal{R}_{\mathbb{D}} \to \mathcal{R}_{\mathbb{U}}$ is defined by **delayR**$^{\ddagger}(d, r) := z$ with the

abstract monitoring stream z being defined in the same way as for $\mathbf{delayR}^{\#}$ with one adjustment: For the case $\bot$ we add the additional requirement

$$\land\ \mathit{defined}(r, t)$$

at the outermost level of the condition. ⌟

The effect of this adjustment is depicted in the stream diagram of the previous example as stream $z = \mathbf{delayR}^{\ddagger}(d, r)$: The progress of r limits the progress of z. This restriction drastically simplifies the progress output and makes it possible to implement the operator with finite memory: If there is sufficient progress on d, the progress of r is passed on.

Note that implementations with finite memory would be possible with less restrictive adjustments. In the example above, the progress of $\tilde{z}$ could be extended up to 2 without the need to consider any of the events on d because there are no events on d until 2. However, the $\mathbf{delayR}^{\ddagger}$ operator was chosen because of its straightforward rules regarding the output of progress.

Next, we define a similar simplification for the $\mathbf{last}^{\#}$ operator. Although this operator can already be implemented with finite memory, we simplify the edge case when there has not been any event that can be reproduced by a trigger yet. Again this does not change the behaviour of the operator with regards to the events:

Definition 7.3 (Semantics of the Abstract Operator $\mathbf{last}^{\ddagger}$)**.** The abstract operator $\mathbf{last}^{\ddagger}\colon \mathcal{S}_{\mathbb{D}} \times \mathcal{S}_{\mathbb{D}'} \to \mathcal{S}_{\mathbb{D}}$ is given by $\mathbf{last}^{\ddagger}(v, r) := z$ with the abstract monitoring stream z being defined in the same way as for $\mathbf{last}^{\#}$ with one adjustment: For the case $\bot$ we remove the condition

$$\text{or } \mathit{uninitialised}(v, t).$$

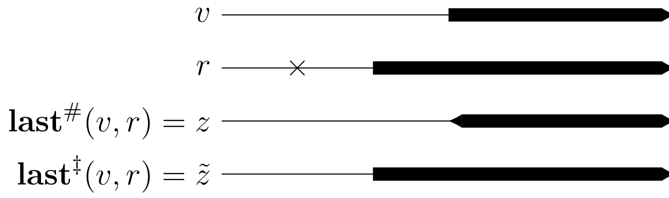
⌟

The following example illustrates the difference between $\mathbf{last}^{\#}$ and $\mathbf{last}^{\ddagger}$:

Example 7.4 (Comparison of $\mathbf{last}^{\#}$ and $\mathbf{last}^{\ddagger}$)**.** Let $v \in \mathcal{R}_{\mathbb{N}}$ and $r \in \mathcal{R}_{\mathbb{U}}$ be two abstract input streams and the derived abstract monitoring stream $z \in \mathcal{R}_{\mathbb{N}}$ defined by

$$z = \mathbf{last}^{\#}(v, r).$$

The following diagram shows a possible evaluation:



The stream v has more progress than r, and due to the absence of events on v, even in the case of an event on r, there would be no event on z. The stream $\tilde{z}$ shows the simpler version of the operator, which passes on the progress of r if there is sufficient progress on v but never exceeds the progress of r. ⌟

We conclude this section with the definition of the adjusted abstract monitoring semantics using the two operators $\mathbf{delayR}^{\ddagger}$ and $\mathbf{last}^{\ddagger}$ introduced above:

Definition 7.5 (Adjusted TeSSLa Abstract Monitoring Semantics)**.** The *adjusted abstract monitoring semantics* $\boldsymbol{f}^{\ddagger}_{\varphi}$ of a TeSSLa specification φ are obtained like the abstract monitoring semantics defined in Definition 6.25 from Section 6.3 but with $\mathbf{delayR}^{\ddagger}$ and $\mathbf{last}^{\ddagger}$ instead of $\mathbf{delayR}^{\#}$ and $\mathbf{last}^{\#}$, respectively. ⌟

7.2. Operator Networks

We introduce operator networks as a formal model which abstracts away many implementation details of actual FPGAs. Operator networks consist of two elements: Operators reading inputs and writing outputs and channels connecting these operators and providing inputs and outputs to them. Abstract monitoring streams as defined in Definition 6.2 from Section 6.1 are encoded in channels based on their sequential notation: Timestamps and values are transmitted in an alternating fashion. This approach has the benefit of naturally combining progress information and event's timestamps:

- A timestamp denotes progress, i. e. the fact that there was no event up to (but not including) the timestamp.
- A value denotes an event. It refers to the last timestamp preceding the value on the channel. The combination of value and timestamp form an event.

The absence of an event is not explicitly encoded, i. e. timestamps can be followed by further timestamps on a channel. This encoding simplifies the implementation of recursive cycles: Compared to the EPUs (see Section 5.6) no additional progress messages are needed.

In some circumstances we still need to explicitly encode the absence of events; see Example 6.16 (Counting With Abstract Monitoring Streams) and Example 6.22

(Period With Abstract Monitoring Streams) from Section 6.2.1 as well as the discussion in Section 7.3.4. For that purpose, we extend the time domain to encode inclusive and exclusive progress as different values. Inclusive progress is a special timestamp that an event cannot follow. While regular timestamps only encode exclusive progress, i. e. they might be followed by a value indicating an event at that timestamp, inclusive timestamps encode inclusive progress, i. e. the guarantee that there is no event at that timestamp.

Definition 7.6 (Extended Time Domain)**.** Every time domain $\mathbb{T}$ can be extended into an *extended time domain* $\mathbb{T}_i$ as follows:

$$\mathbb{T}_i = \mathbb{T} \cup \{t' \mid t \in \mathbb{T}\} \cup \{\infty\}.$$

On the extended time domain the following additional conventions for any $a, b \in \mathbb{T}$ apply:

$$a < \infty, \quad a < b' :\Leftrightarrow a \leq b, \quad a' < b :\Leftrightarrow a < b, \quad a' < b' :\Leftrightarrow a < b,$$
$$a \cdot b' := (a \cdot b)', \quad a + b' := (a + b)', \quad a' \cdot b' := (a \cdot b)', \quad a' + b' := (a + b)'$$

We call timestamps of the form t' for any $t \in \mathbb{T}$ inclusive. The operator $\mathit{incl}\colon \mathbb{T}_i \to \mathbb{T}_i$ with $\mathit{incl}(t) := t'$ and $\mathit{incl}(t') := t'$ for any $t \in \mathbb{T}$ and $\mathit{incl}(\infty) := \infty$ takes any timestamp from the extended time domain and makes it inclusive with the exception of ∞ which cannot be inclusive. ⌟

Every operator has zero or more inputs and one output (see Definition 7.8 below). A channel connects operators as follows:

Definition 7.7 (Channel)**.** A *channel* connects one output of one operator (called the channel's *source*) with one or more inputs (called the channel's *sinks*) from one or more operators. The following operation is available for the output connected to the source of the channel:

– $\mathit{submit}(x)$ for any $x \in \mathbb{T}_i$ or any $x \in \mathbb{D}$ sends a timestamp or a data value, respectively, into the channel. This operation blocks until the channel is ready, i. e. there is currently no timestamp or value in the channel waiting to be received.

The following operations are available for every input i connected to a sink of the channel:

– $\mathit{ts}(i) \in \mathbb{B}$ indicates if a timestamp is available at the channel attached to this input.
– $\mathit{val}(i) \in \mathbb{B}$ indicates if a data value is available at the channel attached to this input.

– *consume*(i) consumes the timestamp or data value at the channel attached to this input.

A channel can hold one timestamp or data value, so for one input i, the predicates *ts*(i) and *val*(i) are mutually exclusive. Every timestamp or data value can be consumed once at every sink. A new timestamp or data value cannot be submitted to the channel until every sink has consumed the current one. The current value of a channel can be accessed by reading the corresponding input i. Reading a channel if neither a timestamp nor a data value is available results in undefined behaviour. ⌟

A channel operator reads from the input channels and writes to its output channel:

Definition 7.8 (Channel Operator)**.** A *channel operator* consists of

– zero or more inputs,
– exactly one output,
– zero of more variables with their initial values and
– a behaviour.

A behaviour is a deterministic finite sequence of conditional instructions using the operations and predicates of the inputs and outputs and reading and writing the variables. Writing a variable has an immediate effect on all following reading instructions. ⌟

If we combine channel operators with channels, we get an operator network:

Definition 7.9 (Operator Network)**.** An *operator network* is a finite graph of channel operators connected with channels. Every channel connects exactly one output of one operator to one or more inputs of one or more operators. Every input or output not connected to any channel is considered as an input or output, respectively, of the network. ⌟

The graph might contain cycles that may lead to deadlocks (see below).

A single operator without any channels forms a *trivial* operator network.

Definition 7.10 (Scheduling)**.** A *scheduling* s for an operator network with the operators $O = \{o_1, o_2, \ldots, o_n\}$ is an infinite sequence of non-empty subsets of O:

$$s \in (2^O \backslash \emptyset)^\omega.$$

A scheduling is called *fair* if every operator occurs infinitely often in the sequence. ⌟

Example 7.11 (Fair Scheduling)**.** Assume an operator network with the operators o_1, o_2 and o_3. The schedulings $s_1 = (\{o_1\}, \{o_2\}, \{o_3\})^\omega$ and $s_2 = \{o_1, o_2, o_3\}^\omega$ are trivially fair. ⌟

In the case of trivial operator networks, there is only one scheduling. That scheduling is always fair.

Definition 7.12 (Operator Network Function)**.** An *operator network function* f with the signature $\mathcal{R}_\mathbb{D}^k \to \mathcal{R}_\mathbb{D}^n$ maps a tuple of abstract monitoring input streams to a tuple of abstract monitoring output streams. The function f is derived from an operator network by applying a scheduling s to the network: All operator's variables are initialised according to their behaviour. Then for every entry s_i of the schedule s the operators in s_i are executed, i. e. all conditional instructions in its behaviour are executed in order. The side effects of the operations *submit* and *consume* of the channels are applied after all operators from s_i are executed. In case of a blocked call the execution of the operator terminates early and continues with this instruction the next time the operator is executed.

We assume that every input stream is mapped to one input of the network, and every output stream is mapped to one output of the network. Inputs can always be consumed until the end of their stream is reached. If an input is consumed then the next tuple from the stream is submitted to the input according to the following mapping:

- Regular events (t, d) for $t \in \mathbb{T}$ and $d \in \mathbb{D}$ are submitted as t and d after another.
- Inclusive progress $(t, \bot)$ for $t \in \mathbb{T}$ is submitted as $incl(t)$.
- Exclusive progress $(t, _)$ for $t \in \mathbb{T}_\infty$ is submitted as t.

Outputs can always be submitted. Submitting an output extends the corresponding stream as follows:

- Inclusive timestamps are appended as inclusive progress to the stream.
- Non-inclusive timestamps are appended as exclusive progress to the stream.
- Data values are appended as an event using the progress of the current stream as the timestamp. ⌟

Note that if an operation during the execution of an entry s_i of a schedule submits data into a channel, this data is not available before s_{i+1}, i. e. other operations from s_i cannot read it. The same applies to the consumption of data from a channel: If operations from s_i consumes data from all sinks of a channel, the channel becomes ready for another submission not before s_{i+1}, i. e. other operations from s_i cannot submit data into it.

Although the operator network function is defined for every combination of operator network and scheduling, we will only consider operator networks translated from TeSSLa specifications and fair schedulings in the next section. We will show that the operator network function of an operator network translated from a TeSSLa specification is equivalent to the specification's abstract monitoring semantics for any fair scheduling.

An infinite amount of execution steps defines the operator network functions because every scheduling is an infinitely long sequence. However, the output of the operator network function might still be a stream with finitely many events because most operators only submit output if they are provided with an input (see next section). Finite schedulings would be sufficient to generate such output streams with only finitely many events, but since TeSSLa can generate Zeno streams (see Section 3.4.4), the operator networks are defined with infinite schedulings to express this behaviour.

7.3. Translating TeSSLa to Operator Networks

In the following we present channel operators for the abstract TeSSLa operators $\mathbf{unit}^{\#}$, $\mathbf{time}^{\#}$, $\mathbf{lift}^{\#}$, $\mathbf{last}^{\ddagger}$ and $\mathbf{delayR}^{\ddagger}$. Lemma 6.20 (Expressiveness of Progressing TeSSLa) from Section 6.2.1 states that these operators are sufficient to gain TeSSLa's full expressiveness. Section 7.3.2 then addresses the translation of a TeSSLa specification into an operator network: The operators are translated into their corresponding channel operators, and the specification's flow graph is encoded using channels connecting these operators. We show the correctness of this translation and finally give an example.

The channel operators for $\mathbf{unit}^{\#}$, $\mathbf{time}^{\#}$, $\mathbf{lift}^{\#}$, $\mathbf{last}^{\ddagger}$ discussed below are based on an earlier version presented in [Buc20]. The implementation discussed in that thesis neither supports the **delay** operator nor inclusive progress. Further, it does not contain any formal representation of the operators apart from their actual implementation.

7.3.1. Imperative Semantics of the Operators

Definition 7.13 (Channel Operator for $\mathbf{unit}^{\#}$)**.** The channel operator for $\mathbf{unit}^{\#}$ has no inputs, uses the variable $\mathit{state} \in \{\mathit{DONE}, \mathit{VALUE}, \mathit{TIME}\}$ initialised with $\mathit{state} = \mathit{VALUE}$ and has the following behaviour:

if $state = VALUE$ **then**
  $submit(\square); state := TIME$

else if $state = TIME$ **then**
  $submit(\infty); state := DONE$ ⌟

The channel operator for **unit**$^{\#}$ has an internal state used to submit the value $\square$ in its first execution, then the timestamp ∞ and then does nothing for all subsequent executions.

Definition 7.14 (Channel Operator for **time**$^{\#}$)**.** The channel operator for **time**$^{\#}$ has the input a, uses the variable $time \in \mathbb{T}_i$ initialised with $time = 0$ and has the following behaviour:

if $ts(a)$ **then**
  $time := a; submit(a); consume(a)$

if $val(a)$ **then**
  $submit(time); consume(a)$ ⌟

The channel operator for **time**$^{\#}$ directly passes on every timestamp and stores the latest passed timestamp in its local memory. Every value is directly passed on, too, but the value is replaced with the latest passed timestamp.

Definition 7.15 (Channel Operator for Unary **lift**$^{\#}(f)$)**.** The channel operator for unary **lift**$^{\#}(f)$ with a function $f\colon \mathbb{D}^n \rightarrowtail \mathbb{D}$ on the data domain has the input a and the following behaviour:

if $ts(a)$ **then**
  $submit(a)$
  $consume(a)$

if $val(a)$ **then**
  if $f(a) \neq \bot$ **then** $submit(f(a))$
  $consume(a)$ ⌟

The channel operator for a unary **lift**$^{\#}(f)$ works very similar to the one for **time**$^{\#}$: Incoming timestamp and values are directly passed on. Instead of replacing values with the last timestamp, the function f is applied to the value.

Definition 7.16 (Channel Operator for binary $\mathbf{lift}^{\#}(f)$)**.** The channel operator for binary $\mathbf{lift}^{\#}(f)$ with a function $f\colon \mathbb{D}^2 \rightarrowtail \mathbb{D}$ on the data domain has the inputs a and b, uses variable $\mathit{progress} \in \mathbb{T}_i$ with the initialisation $\mathit{progress} = 0$ and has the following behaviour:

if $\mathit{ts}(a) \land \mathit{ts}(b)$ **then**
  if $a \leq b$ **then**
    $\mathit{submit}(a); \mathit{consume}(a); \mathit{progress} := a$
    if $a = b$ **then** $\mathit{consume}(b)$
  else
    $\mathit{submit}(b); \mathit{consume}(b); \mathit{progress} := b$

else if $\mathit{ts}(a) \land \mathit{val}(b)$ **then**
  if $f(\bot, b) \neq \bot$ **then** $\mathit{submit}(f(\bot, b))$
  $\mathit{consume}(b)$
  if $a = \mathit{incl}(\mathit{progress})$ **then** $\mathit{consume}(a)$

else if $\mathit{val}(a) \land \mathit{ts}(b)$ **then**
  if $f(a, \bot) \neq \bot$ **then** $\mathit{submit}(f(a, \bot))$
  $\mathit{consume}(a)$
  if $b = \mathit{incl}(\mathit{progress})$ **then** $\mathit{consume}(b)$

else if $\mathit{val}(a) \land \mathit{val}(b)$ **then**
  if $f(a, b) \neq \bot$ **then** $\mathit{submit}(f(a, b))$
  $\mathit{consume}(a); \mathit{consume}(b)$ ⌟

The channel operator for binary $\mathbf{lift}^{\#}(f)$ performs the following synchronisation: Timestamps are only consumed if both inputs are a timestamp. In that case, the smaller timestamp is consumed (or both if they are equal). The input streams are kept synchronised, and if a value is present on any of the inputs, it can be directly processed. If only one input is a value, there cannot be a corresponding event with the same timestamp on the other stream, and the operator passes $\bot$ to the function for that input. In the case of inclusive progress, the progress is consumed if it fits the current progress to ensure that the operator always consumes all processed progress (see Definition 7.25 (Progress Consuming Channel Operators) below).

Every n-ary $\mathbf{lift}^{\#}(f)$ can be split up into nested binary $\mathbf{lift}^{\#}$ applications using Lemma 3.48 (Associativity of **lift**) from Section 3.4.1. We use a tuplification function $u\colon \mathbb{D}^n_\bot \to (\mathbb{D}^n_\bot)_\bot$ with $u(\boldsymbol{d}) = (\boldsymbol{d})$, which converts the input values into a single tuple of values. Let $f\colon \mathbb{D}^3 \rightarrowtail \mathbb{D}$ be a ternary function on the data domain and $a, b, c \in \mathcal{S}_\mathbb{D}$ streams:

$$\mathbf{lift}(f)(a, b, c) = \mathbf{lift}(f)(\mathbf{lift}(u)(a, b), c)$$

We abuse notation and apply the same function f to three arguments of type $\mathbb{D}_\perp$ on the left and two arguments of type $(\mathbb{D}^2_\perp)_\perp$ and $\mathbb{D}_\perp$, respectively, on the right. These changes of the function's signature must be considered in actual implementations. For efficiency reasons, explicit channel operators for $\mathbf{lift}^\#(f)$ of higher arity can be defined by generalising the binary operator given above.

Definition 7.17 (Channel Operator for $\mathbf{last}^\ddagger$)**.** The channel operator for $\mathbf{last}^\ddagger$ has the inputs *trigger* and *value*. It uses the variables

$$\mathit{triggerTime}, \mathit{valueTime} \in \mathbb{T}_i \text{ and } \mathit{store} \in \mathbb{D}$$

initialised with

$$\mathit{triggerTime} = \mathit{valueTime} = 0 \text{ and } \mathit{store} = 0.$$

It has the following behaviour:

if $\mathit{ts}(\mathit{value})$ **then**
  $\mathit{valueTime} := \mathit{value}; \mathit{consume}(\mathit{value})$

if $\mathit{ts}(\mathit{trigger})$ **then**
  $\mathit{triggerTime} := \mathit{trigger}; \mathit{submit}(\mathit{trigger}); \mathit{consume}(\mathit{trigger})$

if $\mathit{val}(\mathit{value}) \wedge \mathit{valueTime} < \mathit{triggerTime}$ **then**
  $\mathit{store} := \mathit{value}; \mathit{consume}(\mathit{value})$

if $\mathit{val}(\mathit{trigger}) \wedge \mathit{valueTime} \geq \mathit{triggerTime}$ **then**
  if $\mathit{store} \neq \perp$ **then** $\mathit{submit}(\mathit{store})$
  $\mathit{consume}(\mathit{trigger})$
  if $\mathit{val}(\mathit{value}) \wedge \mathit{valueTime} = \mathit{triggerTime}$ **then**
    $\mathit{store} := \mathit{value}; \mathit{consume}(\mathit{value})$ ⌟

Timestamps on both inputs are always consumed and stored in the corresponding variables *triggerTime* and *valueTime*. Every timestamp on *trigger* is immediately passed on because the progress of $\mathbf{last}^\ddagger$ only depends on the progress of *trigger*. A value on *value* is stored into *store* and consumed if $\mathit{valueTime} < \mathit{triggerTime}$ because $\mathbf{last}^\ddagger$ outputs the last known value, i. e. the previous value in cases of simultaneous events on *trigger* and *value*. In the other case $\mathit{valueTime} \geq \mathit{triggerTime}$, events on *trigger* are processed and the last known value stored in *store* is submitted. In the case of simultaneous events on *trigger* and *value*, the value can be updated after consuming the *trigger* in the same execution.

Definition 7.18 (Channel Operator for **delayR**‡)**.** The channel operator for **delayR**‡ has the inputs *delay* and *reset*. It uses the variables

$$\begin{aligned}
&\mathit{delayTime}, \mathit{resetTime}, \mathit{nextTime} \in \mathbb{T}_i,\\
&\mathit{state} \in \{\mathit{DONE}, \mathit{TIME}, \mathit{VALUE}\} \text{ and}\\
&\mathit{init} \in \mathbb{B}
\end{aligned}$$

with the initialisation

$$\begin{aligned}
&\mathit{delayTime} = \mathit{resetTime} = 0,\\
&\mathit{nextTime} = \infty,\\
&\mathit{state} = \mathit{DONE} \text{ and}\\
&\mathit{init} = \mathit{false}\,.
\end{aligned}$$

It has the following behaviour:

if $\mathit{ts}(\mathit{delay})$ **then**
  $\mathit{consume}(\mathit{delay}); \mathit{delayTime} := \mathit{delay}$

if $\mathit{val}(\mathit{reset})$ **then**
  $\mathit{consume}(\mathit{reset}); \mathit{state} := \mathit{DONE};$
  $\mathit{nextTime} := \mathit{resetTime}; \mathit{resetTime} := \mathit{incl}(\mathit{resetTime})$

if $\mathit{val}(\mathit{delay})$ **then**
  if $\mathit{delayTime} < \mathit{resetTime} \wedge \mathit{state} = \mathit{DONE}$ **then**
    $\mathit{consume}(\mathit{delay})$
    if $\mathit{delayTime} = \mathit{nextTime}$ **then**
      $\mathit{nextTime} := \mathit{nextTime} + \mathit{delay}; \mathit{state} := \mathit{TIME}$

if $\mathit{ts}(\mathit{reset})$ **then**
  $\mathit{resetTime} := \mathit{reset}$
  if $\mathit{nextTime} <= \mathit{reset} \wedge \mathit{state} = \mathit{TIME}$ **then**
    $\mathit{submit}(\mathit{nextTime}); \mathit{state} := \mathit{VALUE}$
  else if $\mathit{nextTime} <= \mathit{reset} \wedge \mathit{state} = \mathit{VALUE}$ **then**
    $\mathit{submit}(\Box); \mathit{state} := \mathit{DONE}$
  else if $\mathit{nextTime} < \mathit{delayTime} \vee \mathit{reset} \leq \mathit{nextTime}$ **then**
    $\mathit{consume}(\mathit{reset})$
    if $\mathit{reset} \neq \mathit{nextTime}$ **then** $\mathit{submit}(\mathit{incl}(\mathit{reset}))$

else if $\mathit{resetTime} = 0 \wedge \neg\mathit{init}$ **then**
  $\mathit{init} := \mathit{true}; \mathit{submit}(\mathit{incl}(0))$ ┘

The internal synchronisation of the channel operator for $\mathbf{delayR}^{\ddagger}$ is more complex than that of the operators already discussed because this operator can generate additional events. Generating an additional event in this encoding means submitting a timestamp followed by a value. (In this case, the value of always $\square$.) During one execution of the channel operator, it can only submit either a timestamp or a value. The variable *state* determines should be submitted next: A timestamp ($TIME$) or a value ($VALUE$). The operator only submits when a timestamp is present at the *reset* input, which is greater than or equal to the timestamp of the event which should be generated (stored in $nextTime$). If this condition is not fulfilled, the scheduled event might be cancelled by an event on *reset*. The timestamp on *reset* is always stored in $resetTime$ but only consumed if $nextTime < delayTime$ or $resetTime \leq nextTime$, i. e. if there is no event on *delay* at or before the next scheduled event or the input progress on *reset* is not sufficient to generate the next scheduled event.

Timestamps on *delay* are always consumed and stored in $delayTime$. A value on *reset* is always consumed and processed by resetting the *state* and the $nextTime$. The $resetTime$ is marked as inclusive to indicate that there cannot be another reset event at this timestamp. This ensures $delayTime < resetTime$ in case of simultaneous events at *reset* and *delay*.

Values on *delay* are only consumed if there is sufficient progress on *reset* ($delayTime < resetTime$) no event is currently scheduled ($state = DONE$). A new delay is only scheduled if $delayTime = nextTime$, i. e. either if the last delay ended at the timestamp of the new delay or if an event on *reset* at that timestamp has set the $nextTime$ to its timestamp. In both cases the $nextTime$ is increased by the given delay and the *state* is set to $TIME$. Note that setting the *state* to $TIME$ might cause a timestamp to be submitted during the same execution later on if $ts(reset)$.

The last two lines of the behaviour submit an initial inclusive progress of 0, which is needed to get recursions started; see Example 6.22 (Period With Abstract Monitoring Streams) from Section 6.2.1 and Lemma 7.28 (Channel Operators for $\mathbf{last}^{\ddagger}$ and $\mathbf{delayR}^{\ddagger}$ are Progress Increasing) below.

With the above definitions and descriptions of the channel operators, we can conclude the following correctness statement:

Lemma 7.19 (Correctness of the TeSSLa Channel Operators)**.** *Let o be one of the operators $\mathbf{unit}^{\#}$, $\mathbf{time}^{\#}$, $\mathbf{lift}^{\#}$, $\mathbf{last}^{\ddagger}$ or $\mathbf{delayR}^{\ddagger}$. Let f be the operator network function for the trivial operator network consisting only of the channel operator for o. We then have $o \equiv f$.*

7.3.2. Translating TeSSLa Specifications to Operator Networks

Definition 7.20 (Translating TeSSLa Specifications to Operator Networks)**.** Let φ be a progressing, well-formed and flat TeSSLa specification. The corresponding operator network consists of

- the channel operators corresponding to the TeSSLa operators used in φ and
- channels connecting the channel operators according to the flow graph of φ. ⌟

One operand of a TeSSLa operator can only depend on at most one other stream, but multiple operands can depend on the same stream. This relation correlates to a channel having exactly one source but multiple sinks.

We now consider the correctness of this translation. Lemma 7.19 at the end of the last section already states the correctness for trivial operator networks consisting only of a single channel operator. We want to extend this result to operator networks consisting of multiple channel operators connected with channels. For trivial operator networks, there is only one scheduling. For non-trivial operator networks, we consider all fair schedulings. We will show that all operator network functions for a given TeSSLa specification and a fair scheduling are equivalent. A similar consideration for an earlier acyclic version of TeSSLa was given in [LSS+18]. In addition to that result, this section considers recursive specifications, i. e. those with cycles in their flow graph, showing that these cycles cannot lead to deadlocks.

Before we state the main correctness theorem, we observe some properties of the channel operators for the TeSSLa operators, which will be used to prove the main theorem.

Definition 7.21 (Synchronising Operator Network)**.** We call an operator network *synchronising* if all its operator network functions with fair schedulings are equivalent. ⌟

From the definitions of channel operators for the TeSSLa operators given above, we can observe:

Lemma 7.22 (TeSSLa's Operator Networks are Synchronising)**.** *Operator networks which consist only of channel operators for the TeSSLa operators* $\mathbf{unit}^{\#}$*,* $\mathbf{time}^{\#}$*,* $\mathbf{lift}^{\#}$*,* $\mathbf{last}^{\ddagger}$ *or* $\mathbf{delayR}^{\ddagger}$ *are synchronising.*

This lemma implies that the operators are never reading from channels that are not ready because that would be undefined behaviour.

For the next properties of channel operators, we need to extend our notion of progress to channels:

Definition 7.23 (Progress of a Channel)**.** The *progress of a channel* is determined by the progress of the abstract monitoring stream induced by all timestamps and data values passed through the channel (including those currently held by the channel) as defined in Definition 7.12 from Section 7.2 for outputs of an operator network. ⌟

Now we can observe two important properties of the channel operators for the TeSSLa operators: They pass on progress, and they consume progress:

Definition 7.24 (Progress Passing Channel Operators)**.** A channel operator is called *progress passing* if its output eventually has at least the minimal progress of its inputs. ⌟

Channel operators without inputs are considered progress passing because we assume the minimal progress of their inputs to be 0 (exclusive), i. e. the progress of the empty abstract monitoring stream. The term *eventually* in the above definition allows situations where the output's progress is less than the minimal progress of the inputs. However, if the minimal progress of the inputs is not increased, the output's progress must catch up after a finite number of executions.

Channel operators being progress passing is intuitively related to TeSSLa operators being future independent because both properties state that outputs must not depend on future inputs. With a more imperative perspective, they state: Operators must always output at least the minimal progress of the input.

The following property goes one step further: The channel operators not only pass the progress, but they also consume the input with minimal progress afterwards.

Definition 7.25 (Progress Consuming Channel Operators)**.** A channel operator is called *progress consuming* if it eventually consumes inputs whose progress is lower or equal to the output's progress. ⌟

From the definitions of channel operators for the TeSSLa operators given above we can observe:

Lemma 7.26 (TeSSLa's Channel Operators are Progress Passing and Consuming)**.** *The channel operators for the TeSSLa operators* $\mathbf{unit}^{\#}$*,* $\mathbf{time}^{\#}$*,* $\mathbf{lift}^{\#}$*,* $\mathbf{last}^{\ddagger}$ *and* $\mathbf{delayR}^{\ddagger}$ *are progress passing and progress consuming.*

From the combination of progress passing and progress consuming, we get that if the minimal input progress of a channel operator increases, the operator will eventually consume that input progress and submit at least as much progress.

Finally, we observe that the channel operators for $\mathbf{delayR}^{\ddagger}$ and $\mathbf{last}^{\ddagger}$ even fulfil a slightly stronger property than progress passing. If provided with enough progress on the input related to the operator's second argument such that the input related to the operator's first argument is the one with the minimal progress, they not only pass the input's progress but they even increase the progress:

Definition 7.27 (Progress Increasing Channel Operators)**.** A channel operator is called *progress increasing* on an input i if

a) the operator is progress passing and
b) if i is the input with the minimal progress then its output eventually has more progress than i. ⌟

From Definitions 7.17 and 7.18 of the channel operators for $\mathbf{last}^{\ddagger}$ and $\mathbf{delayR}^{\ddagger}$ given above we can observe:

Lemma 7.28 (Channel Operators for $\mathbf{last}^{\ddagger}$ and $\mathbf{delayR}^{\ddagger}$ are Progress Increasing)**.** *The channel operators for* $\mathbf{last}^{\ddagger}$ *and* $\mathbf{delayR}^{\ddagger}$ *are progress increasing on the input related to the TeSSLa operator's first input.*

The channel operators for $\mathbf{last}^{\ddagger}$ and $\mathbf{delayR}^{\ddagger}$ either convert an exclusive progress to an inclusive one, or in case of an inclusive progress, they increase it. Both operators are defined in a way such that their output is not directly dependent on their input.

With all these lemmas in place we can now state the main correctness theorem:

Theorem 7.29 (TeSSLa Operator Networks are Correct)**.** *Let φ be a progressing, well-formed and flat TeSSLa specification, $f^{\ddagger}_{\varphi}$ its adjusted abstract monitoring semantics and f the operator network function of the operator network corresponding to φ. We then have for any fair scheduling*

$$f^{\ddagger}_{\varphi} \equiv f.$$

Proof. Let φ be a progressing, well-formed and flat TeSSLa specification and f the operator network function of the operator network corresponding to φ.

From Lemma 7.19 we know that the individual channel operators are correct. So now we have to prove that the operator network composed of the individual channel operators is correct, too.

From Lemma 7.22 we know that the individual channel operators are synchronising. This result directly carries over to the entire operator network for any fair scheduling because the queues only pass on the output from channel operators to the input of other channel operators. Every such input can be seen as an abstract monitoring stream using the encoding defined in Definition 7.12 from Section 7.2.

While the above considerations are already enough to see the correctness of cycle-free operator networks, we need some more considerations for cycles. From Lemma 7.26 (TeSSLa's Channel Operators are Progress Passing and Consuming) we know that every operator is progress passing; thus, a progress present at any point in the cycle will eventually be passed on through the entire cycle under the assumption that there is sufficient progress available at all external inputs to the cycle. Further from Lemma 7.26 we know that every operator is consuming: The operators are passing on the progress and eventually consume their input, which always allows new inputs to be read. For every cycle in the operator network, we know that there is at least one **last** or **delayR** present in the cycle because it was derived from a well-formed TeSSLa specification. Again under the assumption that their external inputs – which are not part of the cycle – are provided with sufficient progress, **last** and **delayR** guarantee to increase the progress by Lemma 7.28 (Channel Operators for $\mathbf{last}^{\ddagger}$ and $\mathbf{delayR}^{\ddagger}$ are Progress Increasing). Based on these considerations, we can conclude that the operator network is deadlock-free, i. e. if there is input available at all inputs of the network, then eventually one of the inputs will be consumed, and an output will be submitted.

Now we know that a composition of channel operators in an operator network computes precisely the composition of the corresponding abstract operators. Finally, the semantics $f^{\ddagger}_{\varphi}$ are defined as a fixed point which by Lemma 6.26 (Construction of the Least Fixed Point [CHL+18]) from Section 6.3 can be computed as a Kleene chain of individual abstract operators. □

Note how cycles in the dependency graph and hence cycles in the operator network correlate precisely to the fixed point in the semantics being computed by the Kleene chain: The least fixed point in Lemma 6.26 is computed as one fixed point of the entire specification. The operator network computes fixed points for every cycle individually. Both approaches are equivalent because the TeSSLa semantics is compositional, i. e. TeSSLa specifications can be split up into individual specifications, and these specifications can be chained together, as long as cycles, i. e. recursive definitions, remain in the same specification.

7.3.3. Example

We can now revisit Example 6.23 (Variable Frequency Period With Abstract Monitoring Streams) from Section 6.2.1 and show how this specification is evaluated in an operator network. The dependency graph of the original version of this specification using **delay** and not **delayR** was shown in Example 3.33 in Section 3.2.4.

As in the earlier example, we have the input stream $x \in \mathcal{R}_{\mathbb{R}+}$ and the derived streams $\ell, z \in \mathcal{R}_{\mathbb{R}+}$ and $d \in \mathcal{R}_{\mathbb{U}}$ with z being the final output, i. e. the derived stream in which we are interested. The derived streams are defined by the following recursive equations:

$$
\begin{aligned}
d &= \mathbf{delayR}^{\ddagger}(z, x) \\
\ell &= \mathbf{last}^{\ddagger}(x, d) \\
z &= \mathsf{merge}^{\#}(x, \ell)
\end{aligned}
$$

The corresponding operator network is shown in Figure 7.3. The input of channel x and the output of channel z are not connected. These are the input and output of the network.

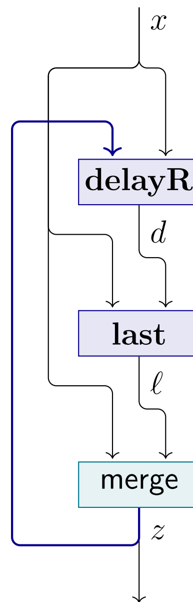


Figure 7.3.: Operator network of the variable frequency period specification. Channels are depicted as arrows. The blue arrow denotes the recursive channel going upwards. Note that there is nothing special about this channel, and which arrow is pointing upwards only depends on the way the graph is drawn.

delayR	in *delay* / z	–	–	⏲1	3	⏲4	3	⏲5	1.5	⏲6.5
	in *reset* / x	–	⏲1	3	⏲5	⏲5	⏲5	1.5	⏲9	⏲9
	out / d	–	⏲1′	–	⏲4	□	⏲5′	–	⏲6.5	□
	delayTime	0	0	1	1	4	4	5	5	6.5
	resetTime	0	1	1′	5	5	5	5′	9	9
	nextTime	∞	∞	1	4	4	7	5	6.5	6.5
	state	DONE	DONE	DONE	VALUE	DONE	TIME	DONE	VALUE	DONE
last	in *value* / x	–	⏲1	3	⏲5	–	–	1.5	⏲9	–
	in *trigger* / d	–	⏲1′	–	⏲4	□	⏲5′	–	⏲6.5	□
	out / l	–	⏲1′	–	⏲4	3	⏲5′	–	⏲6.5	1.5
	triggerTime	0	1′	1′	4	4	5′	5′	6.5	6.5
	valueTime	0	1	1	5	5	5	5	9	9
	store	⊥	⊥	3	3	3	3	1.5	1.5	1.5
merge	in a / x	–	⏲1	3	⏲5	⏲5	⏲5	1.5	⏲9	⏲9
	in b / l	–	⏲1′	⏲1′	⏲4	3	⏲5′	⏲5′	⏲6.5	1.5
	out / z	–	⏲1	3	⏲4	3	⏲5	1.5	⏲6.5	1.5
	progress	0	1	1	4	4	5	5	6.5	6.5

Figure 7.4.: Exemplary execution of the operator network for the variable frequency period specification.

In the following exemplary execution of the operator network we consider the following abstract monitoring stream as input:

$$x = (1, 3)(5, 1.5)(9, _)$$

Figure 7.4 shows the execution of the operator network on the input stream x. The three vertical parts of the table correspond to the three channel operators for **delayR**‡ and **last**‡. The final part labelled merge corresponds to a channel operator for **lift**$^\#$ with the lifted function $f\colon \mathbb{D}^2 \rightarrowtail \mathbb{D}$:

$$f(d_1, d_2) = \begin{cases} d_1 & \text{if } d_1 \neq \bot, \\ d_2 & \text{otherwise.} \end{cases}$$

The inputs and the output of the operators are shown in green. The name of the channel the inputs and outputs are connected to are given after the slash. The symbol – denotes the absence of a timestamp or value on the channel. Timestamps are prefixed with the symbol ⏲. Internal variables of the operators are shown in blue. Variables may contain timestamps, too, but they are not explicitly prefixed with ⏲, since those variables cannot contain anything but timestamps.

Updated values and timestamps are depicted in bold. Repeated values are shown in a lighter colour. Note how the channels do not provide new values or timestamps until all sinks have consumed the old value.

7.3.4. Simplifications for Timestamp-Conservative Specifications

If we consider only timestamp-conservative specifications, we can apply several simplifications: Exclusive progress is sufficient for timestamp-conservative specifications. Remember that in Example 6.16 (Counting With Abstract Monitoring Streams) from Section 6.2 only exclusive progress occurs. Recursive specifications with **delay** need inclusive progress as shown in Example 6.22 (Period With Abstract Monitoring Streams) in Section 6.2.1, but for timestamp-conservative specifications these cases cannot occur because the **delay** operator is not allowed.

The channel operators for $\mathbf{unit}^\#$, $\mathbf{time}^\#$ and even $\mathbf{last}^\ddagger$ do not explicitly consider inclusive progress. They pass on timestamps received on their inputs. The channel operator for $\mathbf{last}^\ddagger$ does not explicitly manipulate timestamps. They are only stored and compared.

The channel operator for $\mathbf{lift}^\#$ however, must consider inclusive progress explicitly because inclusive progress indicates the absence of an event that results in evaluating the function with $\bot$ on the particular input. This evaluation with $\bot$ on the particular input happens in the case of larger progress on the corresponding input, too. However, if the progress on one input is larger, the larger progress is not consumed. If the progress is inclusive but not larger, the inclusive progress must be consumed together with the value on the other input. This is handled by two conditional consumptions of the following form:

if $a = \mathit{incl}(\mathit{progress})$ **then** $\mathit{consume}(a)$

and in a similar form for b. If we know that the progress is never inclusive, then these two conditional consumptions can be removed, and as a result, the variable progress can be removed entirely.

7.4. Implementation Details

This section discusses the actual synthesis of operator networks onto FPGAs. Operator networks consist of channels and channel operators, so we discuss synthesising these two concepts on FPGAs. The implementation discussed below is an extension of an earlier version presented in [Buc20] that neither supports the **delay** operator nor inclusive progress.

In the previous sections, the execution of operator networks was introduced to adhere a given scheduling in Definition 7.12 (Operator Network Function) from Section 7.2. On the actual hardware, every operator is executed in parallel with every clock cycle, which realises a fair scheduling.

7.4.1. Implementation of Channels

The channels as defined in Definition 7.7 from Section 7.2 have several features:

- They can distinguish between data and timestamps.
- Sources can submit, and sinks can check if something is waiting to be consumed.
- They can have multiple sinks and can store data or timestamps until every sink has consumed it.

In the implementation, we split this feature set up into several elements: The channel implementation cannot store anything and can only connect a source with a single sink. Instead, we introduce queues that can store timestamps and values.

A channel indicates to a sink if a value is valid, i. e. if it can be consumed and if it is a data or a timestamp value. It indicates to the source if the channel is ready to receive another value, i. e. if the value was consumed. If we omit the distinction between data and timestamp values, this behaviour is known as ready-valid interfaces from many hardware bus protocols, e. g. the AXI bus provided by Xilinx [XILc]:

- The wire *data* transfers data from the source to the sink. Its bit width corresponds to the bit width of the data values being transferred.
- The wire *ready* is a flag used by the sink to signal the source that the source is ready to consume new data.
- The write *valid* is a flag used by the source to signal the sink that the current data is valid and can be consumed.

Note that the data flow on the ready wire is opposed to the other wires. The ready flag prevents the source from sending new data if the sink is not ready to process data. This mechanism results in messages queued along the pipeline until the source is ready again.

The distinction between timestamps and values can be added into this schema by making the data wire wide enough to contain either a timestamp or a value and equipping it with an additional bit indicating whether it is a timestamp. Different wires are used for timestamps and values in the actual implementation, making the type-safe realisation in Chisel easier (see next section).

AXI provides way more features than the ready-valid interface described above. For example, it contains infrastructure for complex addressing and multi-clock interfaces. AXI4-Stream interfaces are the closest to a simple ready-valid interface because they omit the address phases for faster throughputs. Nevertheless, even AXI4-Stream provides IP blocks that contain unnecessary features.

TeSSLa specifications consist of many channels, and these channels can be considered more an internal feature of the translated specification than an external interface. Hence, the implementation does not rely on any IP blocks but simply uses the following interface together with usage conventions:

- The wire *value* contains event's values.
- The wire *timestamp* contains timestamps.
- The flag *isTimestamp* is true if the current value is a timestamp and should be read from the timestamp wire. Otherwise, the current value is a data value and should be read from the data wire.
- The flags *ready* and *valid* work as before.

In order to avoid combinatorial cycles in the layout, some usage conventions must be established which are very similar to the AXI bus:

- Both, the *ready* and the *valid* flags once being set to true must stay like that until the actual transfer happens.
- The actual transfer happens at the rising edge of the next clock cycle when the flags *ready* and *valid* are both set to true.
- At the source of the channel, the flag *valid* must not be combinatorially derived from the *ready* flag.

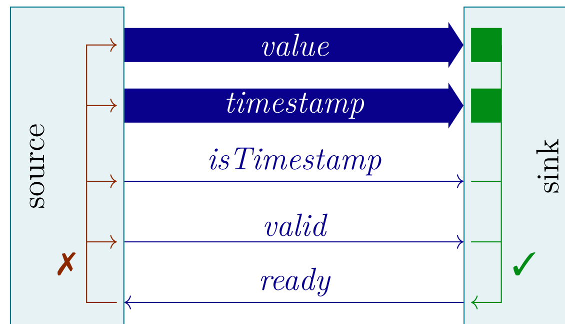


Figure 7.5.: A channel connecting a source with a sink. The wires *timestamp*, *value*, *isTimestamp* and *valid* pass information from the source to the sink and the wire *ready* provides feedback from the sink to the source. To prevent combinatorial cycles the wires from source to sink must not depend on the *ready* wire. The *ready* wire might depend on the other wires.

Very similar conventions apply for using the AXI bus, too. Its documentation [XILc] states: "The `READY` slave output cannot be generated combinatorially from the `VALID` slave input." Note that while AXI forbids a combinatorial relation between *ready* and *valid* at the sink of the channel, the convention above forbids it at the source. Both rules have the same effect of preventing combinatorial cycles, but the adjustment makes the implementations of TeSSLa's channel operators more intuitive because especially the channel operators for the small TeSSLa operators like **lift** do not store data. They only manipulate the received values and immediately pass them on to the next module in the same clock cycle. They are ready to consume a new value with every received valid value that could be passed on to the next channel operator. Intuitively the *ready* flag is used to indicate that a value on the channel was consumed. The *ready* flag depends on the data (or timestamp) anyhow, so an additional dependency on the *valid* flag does not matter much. See Figure 7.5 for a visualisation of the channel and these conventions.

7.4.2. Chisel

On a very abstract level, an FPGA consists of logic cells and a routing network connecting the logic cells. A logic cell consists of lookup tables (LUT), a full adder to implement a combinatorial network, and a flip-flop to store a single bit. In order to program an FPGA, we need a way to specify the routing, the combinatorial network, and the flip-flops. In the two most common hardware description languages (HDL), Verilog and VHDL, this is done with the concept of wires and registers. A wire describes a single logical value and can be connected to other wires using Boolean combinations, which describes the combinatorial networks. A register can store a single bit and can be set and read using wires. The registers and wires are described

in HDLs in a data-flow oriented language which includes a notion of time, e. g. one can specify that a register is read or written with the rising edge of a clock signal.

Neither Verilog nor VHDL was designed exclusively for programming FPGAs. They support a wide range of features, from describing hardware to testing it using testbench-based verification. Hence, the languages support many different programming styles and many different operators and the synthesisable fragment of the languages that can be used to program FPGAs is not precisely specified. It depends on the used tooling. See [Han18] for more discussions on writing synthesisable Verilog.

High-level synthesis (HLS), e. g. SystemC, aims to overcome the difficulties of writing Verilog or VHDL. It automatically transfers programs written in an imperative style into the data-flow-oriented register-transfer level. Chisel [BVR+12] follows a different approach: On the one hand, it utilises Scala as a metaprogramming language used to describe wires and registers. Compared to HLS, one does not lose any precision due to the usage of non-matching programming models because one still describes the exact relation of the wires and registers in a data-flow-oriented way. Schuyler Eldridge, one of Chisel's maintainers, summarises the different approach in [Eld18] as follows: "The differentiating factor here is that Chisel is still, fundamentally, a powerful language for describing *circuits* while HLS is a path for converting *programs* to *circuits*." The generalisation and reusability are, however, highly increased by using Scala features to do so. On the other hand, Chisel is compiled to the Flexible Internal Representation for RTL (Firrtl), which is an HDL specifically for programming FPGAs and hence is entirely synthesisable: Some concepts of Verilog and VHDL are simplified, so for example, there are only two Boolean values, and the global clock is implicitly assumed for registers. Multi-clock designs are still possible, but since this is not needed for the TeSSLa synthesis, we will not discuss this in the context of this thesis.

Chisel – or more precisely the Firrtl [IKL+17] code generated with the Chisel Scala frontend – can be compiled either to Verilog or to VHDL. Chisel allows the definition of modules that are connected by bundles. A module has an interface defined in terms of bundles, and a bundle is a typed set of wires. The following code shows the slightly simplified implementation of the ready-valid interface discussed above in Chisel:

```
class ChannelInterface extends Bundle {
  val ready = Output(Bool())
  val valid = Input(Bool())
  val isTimestamp = Input(Bool())
  val value = Input(DataType)
  val timestamp = Input(TimeType)
}
```

Note that the above code only defines data types but not the functionality. In the same way, modules make it easier to instantiate code multiple times, but they do not add any overhead to the final compilation product. However, modules are preserved in the Verilog or VHDL code which can be helpful to identify portions of the compiled code in the Xilinx tools during debugging.

7.4.3. Implementation of Channel Operators

A channel operator as defined in Definition 7.8 from Section 7.2 is represented as a Chisel module as follows:

The inputs are declared as channel interface and the outputs as flipped channel interface, i. e. a channel interface where inputs and outputs are swapped.

The variables are declared as registers, which are initialised accordingly.

The behaviour is implemented using Chisel's conditional assignment feature with the `when` keyword. This language features provides syntactic sugar for defining values of wires and registers. So for example instead of

```
io.out.timestamp := Mux(io.a.timestamp <= io.b.timestamp,
  io.a.timestamp, io.b.timestamp)
```

one can write

```
when(io.a.timestamp <= io.b.timestamp) {
  io.out.timestamp := io.a.timestamp
}.otherwise {
  io.out.timestamp := io.b.timestamp
}
```

The `Mux` in the first notation takes three arguments:

- The Boolean condition,
- the value to be used if the condition is fulfilled,
- and the value to be used otherwise.

The second notation allows the definition of multiple wires or registers based on the same condition, so this notation can be used to express the behaviour of the channel operators. The functions *val*, *ts* and *consume* of channels are translated using these Chisel definitions:

```
def ts(a: ChannelInterface) = a.valid && a.isTimestamp
def vl(a: ChannelInterface) = a.valid && !a.isTimestamp
def consume(a: ChannelInterface) = {
  a.ready := true.B
}
```

The function *submit* is slightly more complicated because it might block if the channel is not ready to receive a new value. The Chisel semantics does not support blocking calls because this imperative concept cannot be directly translated into its data-flow semantics.

If the channel is ready and we want to submit something, we assign the data wires to the new values and set the valid flag. The actual transfer is then assumed to happen at the rising edge of the next clock cycle. If, however, the ready bit is not set, we simulate the blocking of the call to *submit* by making sure that nothing changes until we reach the same position in the code for the next clock cycle. Following sequential assignments happing after the call to *submit* are postponed until the next clock cycle. The following Chisel definitions provide a non-blocking implementation of *submit*. We distinguish the two cases of submitting a value and a timestamp:

```
def submitValue(value: V): Unit = {
  out.valid := true.B
  out.isTimestamp := false.B
  out.value := value
}

def submitTime(timestamp: T): Unit = {
  out.valid := true.B
  out.isTimestamp := true.B
  out.timestamp := timestamp
}

def submitted = out.ready
```

The predicate `submitted` is mapped to the channels ready flag. Assignments after the call to *submit* are only executed when `submitted` evaluates to true. So for example the following first case from the channel operator for **lift**

if $\mathit{ts}(a) \wedge \mathit{ts}(b)$ **then**
  if $a \leq b$ **then**
    $\mathit{submit}(a); \mathit{consume}(a); \mathit{progress} := a$
    if $a = b$ **then** $\mathit{consume}(b)$
  else
    $\mathit{submit}(b); \mathit{consume}(b); \mathit{progress} := b$

becomes:

```
when (ts(io.a) && ts(io.b)) {
  when(io.a.timestamp <= io.b.timestamp) {
    submitTime(io.a.timestamp)
     when (submitted) { // submit did not block
      consume(io.a)
      progress := io.a.timestamp
      when (io.a.timestamp ==  io.b.timestamp) {
        consume(io.b)
      }
    }
  }.otherwise {
    submitTime(io.b.timestamp)
    when (submitted) { // submit did not block
      consume(io.b)
      progress := io.b.timestamp
    }
  }
}
```

7.4.4. Implementation of Queues

As mentioned earlier, the implementation of channels cannot store values on their own. Instead, queues are included for that purpose. The queues can be seen as an optimisation so that not every channel has to store values. Queues have the same ready-valid interface as the channel operators. A queue always has one input but can have multiple outputs. A queue is a ring buffer storing an alternating sequence of timestamps and values with one write pointer and multiple read pointers, one for each output.

Queues are included into the data-flow graph for the following purposes:

(1) *Dispatch data from a single source to multiple targets.* The ready-valid interface only supports data transfer from a single source to a single target. In order to dispatch data to multiple targets, it is sent to and stored in a queue instead. Multiple targets can now read the data from the queue. Using queues to dispatch data to multiple targets has two benefits:

   a) It keeps the ready-valid interface simple because there is always only one target.

b) The queue is equipped with multiple read pointers such that the targets can asynchronously read the data. Otherwise, without storing the data in a queue, the data transfer would have to happen synchronously from the source to all targets in the same clock cycle.

(2) *Break up combinatorial cycles.* This effect of the queues is implicit because every cycle in the operator network has at least one node whose output is connected to multiple targets. This connection requires a queue anyhow, so every cycle in the network is broken up by at least one queue. A cycle without such a node would not affect the rest of the network and could easily be eliminated.

(3) *Break up too long paths.* As discussed in this chapter's introduction, the length of paths between registers is vital for FPGA design. Since registers store values, their output at a particular clock cycle is independent of their input at that particular clock cycle. The timing limits the number of logical operators and the path length on the actual hardware chip between the involved logic blocks. The slack is the time difference between the required arrival of a signal and the actual arrival of a signal. This slack must be positive for the design to be synthesisable. If sequences of combinatorial operations are becoming too long, the slack gets negative, and additional registers must be introduced into the path. In the translation of operator networks, this is done with additional queues.

Queues are implemented as a custom module and not using the Xilinx FIFO IP core [XILd] because of the following requirements for queues:

We need *multiple reading pointers* which is not immediately supported by Xilinx FIFOs and was easier to implement manually using registers.

Further, queues need the ability to *override the previously inserted timestamp* instead of enqueuing it. If two timestamps are passed along consecutively on a channel, the second timestamp is an update of the first timestamp; both can be combined into the larger timestamp. While preserving both timestamps would be semantically correct, it increases the throughput to combine both timestamps into a single one in the queue.

Finally, the queues need the *first-word fall-through (FWFT)* [XILd] semantics in order to match the channel interface semantics: It must be possible to read a value and decide based on that value if it should be consumed or if the value should stay so that it can be reread in the next clock cycle. Traditional FIFOs do not have FWFT because they are implemented using memory blocks. Memory provides in the current clock cycle the value stored at the address assigned in the last clock cycle.

The Xilinx IP cores for FIFOs do support FWFT, but in combination with the other special requirements, it was easier to manually implement the queues using registers. A possible optimisation would be the usage of memory blocks for larger queues. Memory blocks are much more efficient in resource consumption, but additional registers are still required to implement FWFT. One has to store the data at the position of any reading pointer separately in registers. Hence such an optimisation would only improve the situation for larger queues, and a size of one is already sufficient for many queues. In that case, using memory blocks does not gain much because the entire queue does not store much more information than what needs to be stored in registers anyhow.

Internally two separate queues are used to store timestamps and values instead of one queue storing them alternatingly. This separation has the same practical reason of relying on the available static type checking, which was already discussed for the channel interface. Additionally, it would be a waste of resources to store timestamps and values in the same data structure if their bit width would differ.

More technical details on the actual implementation of the queues can be found in [Buc20, Section 3.2.1. Queues].

Translation of Operator Networks.

All the purposes for introducing queues mentioned above could be already satisfied by adding a queue of size one for every channel. However, this would not be ideal for the resource usage of the hardware layout and the timing because a queue does not pass on a received value in the same timestamp, i. e. every queue introduces latency to the network. In the case of recursive cycles, latency decreases the throughput, too. So we will investigate better heuristics regarding the queue placement in the next section.

Queue Placement

Queues must be placed for every channel with more than one target to satisfy queue purposes (1) and (2). For queue purpose (3), breaking up long paths, however, a better heuristic than placing a queue for every channel can be defined:

In order to estimate the slack between two modules, the following estimation is used: We compute a slack constant for every input of every channel operator based on the amount of combinatorial logic between this input and the operator's output. There is no combinatorial connection to the output from inputs corresponding to delayed-labelled edges in the dependency graph. The same is true for queues: There is no combinatorial connection from their inputs to their inputs.

These slack constants are then aggregated along the operator network and reset to zero for every delay-labelled edge. If the aggregated slack constant exceeds a threshold, a queue is added at that position in the network, and the aggregated slack constant is reset to zero. The threshold's value can only be determined empirically and highly depends on the used FPGA hardware and the synthesis.

Unfortunately, this estimation only considers combinatorial logic but not the physical path length on the hardware. This path length is mainly driven by the physical distance of the elements on the FPGA used to realise the logic. With a higher utilisation of the FPGA, these path lengths typically increase as the utilisation of the available logic elements becomes more difficult.

Queue Depth

For certain operator network graphs, the throughput of the entire graph can be increased with additional buffering capacities in the graph: Assume a data flow graph splitting up the same input and only sending one branch through multiple queues before joining both branches again in one operator at the end. In that situation, every new timestamp must be passed through all the queues until it reaches the operator at the end, where both branches are joined. While the new timestamp travels down the queues, no new timestamp can be inserted into that pipeline of queues because the operator at the end is blocking the input: It cannot consume a timestamp on the input connected to the short branch until the same timestamp has reached its other input connected to the long branch. In this case, both timestamps are always the same because they are inherited from the same origin, and thus there is never sufficient progress at the other input until the same timestamp has arrived.

There is already a queue placed at the input, which dispatches data to both branches. If this queue's depth is increased, it provides already new input data into the pipeline while the operator at the bottom still waits. With this adjustment, the longer branch can process following timestamps, which are not yet accepted by the shorter branch.

In order to compute proper queue depth, all channel operators in the operator network are identified whose inputs depend on a common ancestor. Then the path's latency, i. e. the number of queues on the path, from that ancestor to the operator is computed. The maximal difference of the latencies determines the size of the queue at the common ancestor.

7.5. Tuplification Optimisation

Next to the queues, most FPGA resources are used for the synchronisation mechanisms. Therefore, this section presents some modifications of specifications that do not change their semantics but aim to reduce their resource utilisation by reducing the required amount of synchronisations. The required number of timestamp comparisons for the synchronisation of multiple streams can be reduced by replacing parallel streams with a single stream of tuples.

See Figures 7.6 and 7.7 in Section 7.5.3 for examples of such graph transformations. Note that other than the queue placement discussed in the last section, this optimisation phase is purely performed on the TeSSLa specification itself.

Especially the transformation of parallel **lift** operators into a joined **lift** of tuples can be applied quite often, but it sometimes comes with a cost of increased latency or even throughput: By combining two streams into a stream of pairs, we lose the ability the evaluate both streams asynchronously. This combination reduces the amount of synchronisation needed, but the asynchronous evaluation also allows efficient evaluation of asynchronous streams on hardware. So combining streams to treat them synchronously is a trade-off between resource utilisation and throughput and latency.

Intuitively the following rules are applied: Parallel **lift** operators are combined into a single **lift** operator as shown in Figure 7.6 if both streams have events at the same timestamps. This combination will be formalised in the equivalence relation $\equiv_t$ This restriction is slightly softened by the relation $\subseteq_m$ to include streams with the same timestamps except for an initial phase. Parallel **last** operators are combined into a single **last** operator if their value streams fulfil the same requirements. Their trigger streams must have the same timestamps because otherwise, the tuplification would change the semantics of the specification. Further, in both cases, the streams must be independent for the transformation to work correctly. Especially the situation where both streams are part of different cycles must be avoided because combining different cycles into a common cycle might drastically reduce the throughput of the synthesised specification.

The tuplification optimisations were developed in the context [Buc20].

Before defining what the two graph transformations do and when they are applied, we define auxiliary equivalence and dependency relations used to identify appropriate streams in a TeSSLa specification.

7.5.1. Timestamp Relations

The equivalence of two TeSSLa specifications was defined in Definition 3.29 from Section 3.2.4 as follows: With $\boldsymbol{f}_\varphi, \boldsymbol{f}_\psi \in \mathcal{S}_\mathbb{D}^k \to \mathcal{S}_\mathbb{D}^n$ we denote the semantic functions of the specifications φ and ψ. We call them equivalent and write $\varphi \equiv \psi$ iff

$$\forall\, \boldsymbol{y} \in \mathcal{S}_\mathbb{D}^k \colon \boldsymbol{f}_\varphi(\boldsymbol{y}) = \boldsymbol{f}_\psi(\boldsymbol{y}).$$

We use $T(s) \subseteq \mathbb{T}$ for the set of timestamps used in $s \in \mathcal{S}_\mathbb{D}$ and $\boldsymbol{T}(\boldsymbol{s})$ for the element-wise application of T to all streams in the tuple $\boldsymbol{s} \in \mathcal{S}_\mathbb{D}^k$.

Definition 7.30 (Equivalence of TeSSLa Specifications Regarding Timestamps)**.** Two TeSSLa specifications φ and ψ are called *equivalent regarding their timestamps* denoted with $\varphi \equiv_t \psi$ iff

$$\forall\, \boldsymbol{y} \in \mathcal{S}_\mathbb{D}^k \colon \boldsymbol{T}(\boldsymbol{f}_\varphi(\boldsymbol{y})) = \boldsymbol{T}(\boldsymbol{f}_\psi(\boldsymbol{y})).$$ ⌟

It follows directly that for any two specifications φ and ψ the equivalence $\varphi \equiv \psi$ implies the equivalence regarding timestamps $\varphi \equiv_t \psi$.

In many practical cases, streams are equivalent regarding timestamps except for some initial events. In order to weaken the equivalence regarding timestamps accordingly, a first idea would be to ignore a fixed amount of time at the beginning of the streams, i. e. ignore all timestamps smaller than a threshold. Unfortunately, this would not be a good criterion because one typically cannot derive such a property statically from the specification without knowing the actual input streams. After all, the actual values of the timestamps depend on the actual input streams.

A different approach is to consider two streams related if they contain the same timestamps starting with their first event. Ignoring all events happening on one stream before the first event on the other stream fits quite well to typical specification patterns in which, for example, **last** removes the first event from a stream.

The corresponding relation is no longer an equivalence relation because it depends on the order of the operands whose initial events can be ignored. However, it is possible to define a transitive relation. As explained in the next section, that is sufficient to derive this property on specifications with a rule-based approach.

Let $A \subseteq \mathbb{T}$ be a non-empty set of timestamps $A \neq \emptyset$ and $B \subseteq \mathbb{T}$ an arbitrary set of timestamps. We then say that A *is subsumed under* B and write $A \subseteq_m B$ iff

$$A = \{t \in B \mid t \geq \min(A)\}.$$

Further, for any $B \subseteq \mathbb{T}$ we have

$$\emptyset \subseteq_m B.$$

In the case of tuples of sets of timestamps, the relation is applied element-wise. Let $\boldsymbol{A}, \boldsymbol{B} \in 2^{\mathbb{T}^k}$ be tuples of sets of timestamps. Then $\boldsymbol{A} \subseteq_m \boldsymbol{B}$ holds iff

$$\forall i \in \{1, 2, \ldots, k\} \colon A_i \subseteq_m B_i.$$

Using this new relation we define the corresponding relation on TeSSLa specifications:

Definition 7.31 (Subsumption of TeSSLa Specification)**.** Let φ and ψ be two TeSSLa specifications. We say that φ is subsumed by ψ regarding timestamps and write $\varphi \subseteq_m \psi$ iff

$$\forall \boldsymbol{y} \in \mathcal{S}_{\mathbb{D}}^k \colon \boldsymbol{T}(\boldsymbol{f}_\varphi(\boldsymbol{y})) =_m \boldsymbol{T}(\boldsymbol{f}_\psi(\boldsymbol{y})).$$ ⌟

It follows directly that for any two specifications φ and ψ the equivalence regarding timestamps $\varphi \equiv_t \psi$ implies subsumption in both directions $\varphi \subseteq_m \psi$ and $\varphi \supseteq_m \psi$.

Similar to the regular equivalence relation, we use the notational convention that two streams are equivalent if their specifications projected to these streams are equivalent.

Deriving Timestamp Relations

Neither the equivalence of TeSSLa specifications nor the equivalence relation $\equiv_t$ of TeSSLa specifications regarding timestamps, nor the subsumption relation $\subseteq_m$ of TeSSLa specification can be computed for arbitrary specifications. These relations are semantic properties that might depend on the semantics of a lifted function which is not restricted and thus cannot be analysed statically in the general case.

In the following, we establish some simple rules used to derive the three relations in some cases. These rules can be used to derive an under-approximation of the relations because the following rules are universally true, but they are not sufficient to derive the relations in all cases.

Let f and g be arbitrary ideal $\bot$-functions, $x, y \in \mathcal{S}_{\mathbb{D}}$ streams and $\boldsymbol{z} \in \mathcal{S}_{\mathbb{D}}^n$ a tuple of streams. We then have the following general equivalences:

$$\mathsf{nil} \equiv \mathbf{lift}(f)(\mathbf{last}(x, y), \mathbf{unit})$$
$$\mathsf{nil} \equiv \mathbf{last}(x, \mathsf{nil})$$

$$\begin{aligned}
\mathsf{nil} &\equiv \mathbf{last}(\mathsf{nil}, x)\\
\mathbf{last}(x, y) &\equiv \mathbf{last}(x, \mathbf{last}(x, y))\\
\mathbf{last}(x, y) &\equiv \mathbf{last}(x, \mathbf{last}(\mathbf{unit}, y))
\end{aligned}$$

The following equivalences regarding timestamps hold:

$$\begin{aligned}
x &\equiv_t \mathbf{time}(x)\\
x &\equiv_t \mathbf{lift}(f)(x)\\
\mathbf{last}(x, \mathbf{lift}(f)(\boldsymbol{z})) &\equiv_t \mathbf{lift}(g)(\mathbf{last}(x, z_1), \mathbf{last}(x, z_2), \ldots, \mathbf{last}(x, z_n))
\end{aligned}$$

The following additional equivalences regarding timestamps hold when the initial timestamps are ignored:

$$\begin{aligned}
&y \supseteq_m \mathbf{last}(x, y)\\
&y \supseteq_m \mathbf{lift}(f)(\mathbf{last}(z_1, y), \mathbf{last}(z_2, y), \ldots, \mathbf{last}(z_n, y))
\end{aligned}$$

More rules, especially for additional classes of lifted functions, can be found in [Buc20].

7.5.2. Dependencies

In order to give formal rules when to apply the optimisations, we need two more auxiliary definitions regarding the dependence of streams in a TeSSLa specification:

Definition 7.32 (Dependent Streams)**.** Let $x, y \in \mathcal{S}_{\mathbb{D}}$ be two derived streams defined in a common well-formed TeSSLa specification φ. We than say that x *depends on* y and write $x \rightsquigarrow y$ iff there exists a path in the dependency graph of φ from x to y which does not use any delayed-labelled edge. ⌟

Note that this dependency relation is defined on an acyclic graph because the definition considers only well-formed TeSSLa specifications and ignores the delayed-labelled edges.

Definition 7.33 (Connected Cycles)**.** Let $x, y \in \mathcal{S}_{\mathbb{D}}$ be two derived streams defined in a common well-formed TeSSLa specification φ. We say that x and y *connect cycles* iff

a) x and y are part of two different non-trivial SCCs in the dependency graph of φ and
b) there is a path in the dependency graph of φ which connects the SCCs of x and y. ⌟

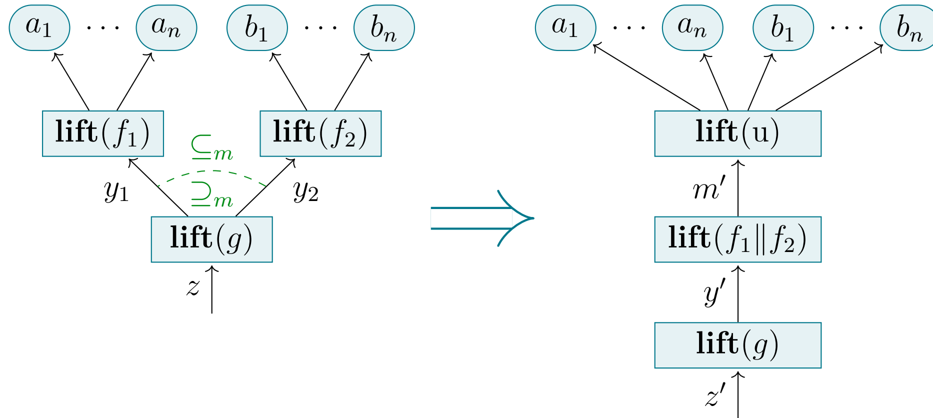


Figure 7.6.: Tuplification for parallel **lift** operators. The conditional relation is shown by the green dashed line: The derived streams y_1 and y_2 should have most events at the same timestamps for this optimisation to be effective.

Note that for this definition, the delayed-labelled edges are considered, too. Otherwise, there would be no non-trivial SCC in the dependency graph of a well-formed TeSSLa specification. Note further that the connection between the SCCs of x and y might be a path in either direction, but not both because then x and y would be part of the same SCC.

7.5.3. Graph Transformations

Using the definitions regarding equivalence and dependency of streams in a TeSSLa specification, we can now precisely define the two graph transformations used to optimise the resource utilisation of the translated specification.

Merge Parallel lift

Let φ be a TeSSLa specification with the following streams: $\boldsymbol{a} \in \mathcal{S}_{\mathbb{D}}^k$ and $\boldsymbol{b} \in \mathcal{S}_{\mathbb{D}}^n$ are tuples of streams and $y_1, y_2, z \in \mathcal{S}_{\mathbb{D}}$ are derived streams with the following definitions:

$$\begin{aligned} y_1 &= \textbf{lift}(f_1)(\boldsymbol{a}) \\ y_2 &= \textbf{lift}(f_2)(\boldsymbol{b}) \\ z &= \textbf{lift}(g)(y_1, y_2) \end{aligned}$$

with $f_1 \colon \mathbb{D}^k \rightarrowtail \mathbb{D}$, $f_2 \colon \mathbb{D}^n \rightarrowtail \mathbb{D}$ and $g \colon \mathbb{D}^2 \rightarrowtail \mathbb{D}$ being arbitrary ideal $\perp$-functions.

We can rewrite this as follows introducing the new derived stream $m' \in \mathcal{S}_{\mathbb{D}^{k+n}}, y' \in \mathcal{S}_{\mathbb{D}^2}$ and $z' \in \mathcal{S}_{\mathbb{D}}$:

$$\begin{aligned} m' &= \textbf{lift}(u)(\boldsymbol{a}, \boldsymbol{b}) \\ y' &= \textbf{lift}(f_1 \| f_2)(m') \\ z' &= \textbf{lift}(g)(y') \end{aligned}$$

Note that m' is a stream of tuples, while $\boldsymbol{a}$ and $\boldsymbol{b}$ are tuples of streams. The function u is the tuplification function introduced earlier. By $(f_1 \| f_2)\colon \mathbb{D}^k \times \mathbb{D}^n \rightarrowtail \mathbb{D}$ we denote the function applying f_1 and f_2 in parallel: $(f_1 \| f_2)(\boldsymbol{c}, \boldsymbol{d}) = (f_1(\boldsymbol{c}), f_2(\boldsymbol{d}))$. We again abuse notation by applying the same function g to two arguments of type $\mathbb{D}_\perp$ in the first case and one argument of type $(\mathbb{D}_\perp \times \mathbb{D}_\perp)_\perp$ in the second case.

By Lemma 3.48 (Associativity of **lift**) from Section 3.4.1 we have $z \equiv z'$ on monitoring streams, but as shown in Example 6.33 from Section 6.3.2 this equivalence does not always hold on abstract monitoring streams. Two independent $\textbf{lift}^\#$ operators can be more asynchronous than one $\textbf{lift}^\#$ operator applied to streams of tuples. From Lemma 6.31 (Relation Between the Abstract TeSSLa Monitoring Semantics and the TeSSLa Semantics) in Section 6.3.1 we know that they only differ regarding the generated progress. This difference in the produced progress is an important trade-off for this optimisation and the reason why using this transformation is not always efficient: It is only efficient if the tuples created by the tuplification are not sparse, i. e. most of their values are not $\perp$. Otherwise, the tuplification introduces additional synchronisation between streams, preventing the FPGA from utilising its main benefit, the asynchronous processing of independent streams. Formally we only apply this optimisation if $y_1 \subseteq_m y_2$ or $y_1 \supseteq_m y_2$.

See Figure 7.6 for a visualisation of the graph transformation and this requirement.

We further require that y_1 and y_2 do not connect cycles because otherwise, the tuplification could combine both cycles into one larger cycle, which could drastically decrease the throughput of the specification by increasing the latency of a cycle.

We can safely assume that $y_1 \not\rightsquigarrow y_2$ because otherwise we could insert an additional $\textbf{lift}(id)$ operator in the graph between y_1 and z and apply the transformation to the resulting specification, i. e. combining y_2 with the inserted id node instead of y_1. Due to the symmetry of the graph, the same applies to $y_2 \rightsquigarrow y_1$.

As shown in Figure 7.6 the depths of the dependency graph might be increased by this graph transformation. An increased depth neither increases the latency nor the resource utilisation because the combinatorial logic in the channel operators for unary **lift** is much simpler than in the general case of the binary or even n-ary **lift** which must synchronise the incoming timestamps and values across all input channels in order to apply the lifted function only on events with the same timestamps.

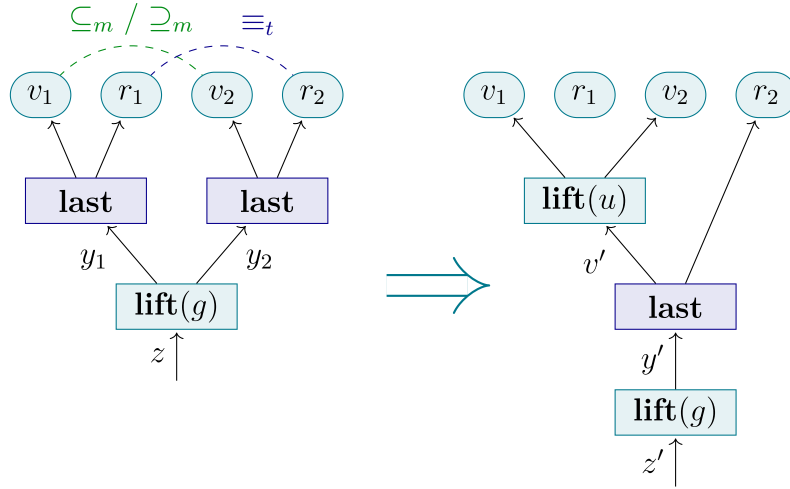


Figure 7.7.: Tuplification for parallel **last** operators. The condition relations are shown by the green and purple dashed line: The value streams v_1 and v_2 should have most events at the same timestamps for this optimisation to be effective. The trigger streams r_1 and r_2 must have all events at the same timestamps for this optimisation to be correct.

Merge Parallel last

Let φ be a TeSSLa specification with the following streams: $v_1, r_1, v_2, r_2 \in \mathcal{S}_\mathbb{D}$ are streams and $y_1, y_2, z \in \mathcal{S}_\mathbb{D}$ are derived streams with the following definitions:

$$\begin{aligned} y_1 &= \textbf{last}(v_1, r_1) \\ y_2 &= \textbf{last}(v_2, r_2) \\ z &= \textbf{lift}(g)(y_1, y_2) \end{aligned}$$

with $g\colon \mathbb{D}^2 \rightarrowtail \mathbb{D}$ being an arbitrary ideal $\bot$-function.

We can rewrite this as follows introducing the new derived streams $v', y' \in \mathcal{S}_{\mathbb{D}^2}$ and $z' \in \mathcal{S}_\mathbb{D}$:

$$\begin{aligned} v' &= \textbf{lift}(u)(v_1, v_2) \\ y' &= \textbf{last}(v', r_1) \\ z' &= \textbf{lift}(g)(y') \end{aligned}$$

Very similar to the previous case of parallel **lift** the streams v' and y' are streams of pairs.

With the condition $r_1 \equiv_t r_2$ we have $z \equiv z'$. For the tuplification to be efficient we further require $v_1 \subseteq_m v_2$ or $v_1 \supseteq_m v_2$. See Figure 7.7 for a visualisation of the graph transformation and these requirements.

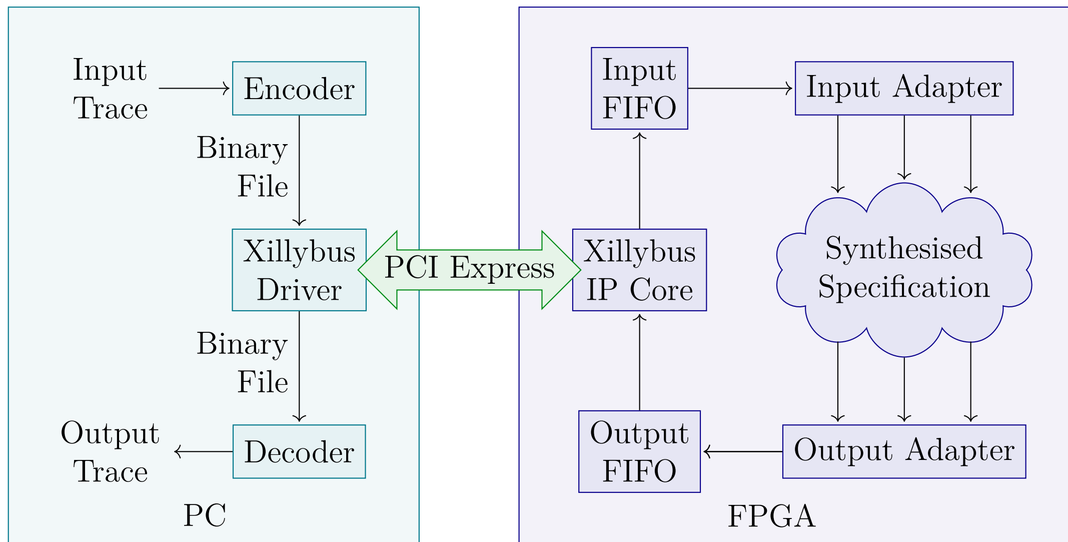


Figure 7.8.: Integration Test Setup using PCI Express via Xillybus.

As before we further require that y_1 and y_2 do not connect cycles and assume that $y_1 \not\rightsquigarrow y_2$ and $y_2 \not\rightsquigarrow y_1$.

7.6. Integration and Test Setup

A TeSSLa specification can be translated into an operator network which can then be compiled via Chisel into a Verilog or VHDL hardware module. This hardware module has inputs and outputs matching the input and output streams of the specification. The usage of such a module highly depends on the concrete use case. If parallel trace sources are available on the same hardware, these trace sources can be attached directly to the inputs. The following approach was used to evaluate synthesised specifications in the context of this thesis: Input events are fed via PIC Express from a host PC to the FPGA. The synthesised hardware module then processes these events on the FPGA, and the output event stream is fed back to the host PC via the same PCI express interface. Figure 7.8 shows an overview of this setup. Xillybus[1] was chosen as a ready to use PCI Express data streaming solution [PS14]. On the host PC, it provides a driver that provides memory-mapped virtual files used to write and read binary data into and from the interface. On the FPGA, it provides an IP core connected to two FIFOs used as input and output buffer. Events can then be read from the input buffer and fed into the synthesised

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

specification. On the other end, output events are written into the output buffer and transferred back through the PCI Express bus.

The compiled operator network is clocked with the same clock as the Xillybus IP core such that the PCI Express interface does not become a bottleneck of this setup because it provides new data for every clock cycle. However, the PCI Express connection only provides a single input and output channel. Hence a synchronous encoding of multiple streams into that single channel is needed. The following ad hoc encoding was mainly chosen because it is simple to decode and encode on the hardware: The first bit of every data frame encodes if the data frame is a timestamp or a value. The timestamps are the same for all encoded streams. In the case of a value, the next few bits encode an address followed by the actual value. If multiple streams carry an event at the same timestamp, then multiple data frames with values might follow after a single timestamp frame. If the operator network has multiple inputs, the PCI Express interface can, after all, become a bottleneck because it can not provide new data to every input of the compiled operator network with every clock cycle. The effect of this limitation is discussed in Section 8.3.3 (Number of Inputs).

The input adapter is a special hardware module reading such a sequence of data frames and dispatching it into multiple input channels of a synthesised operator network as follows: A timestamp is passed on to all input channels, and a value is passed to the channel corresponding to the address given in the data frame. In order to avoid deadlocks, a queue is added directly after the input adapter for every channel. A queue size of 1 is sufficient because the input events are encoded synchronous in the data frame sequence.

The output adapter works in a very similar fashion. Like a **lift** it synchronises its input channels based on the event's timestamps and then passes the timestamp to the output. Addresses are added to values corresponding to their output channel.

The input and output adapters were developed in the context of [Buc20].

On the software side, an encoder takes an input trace in a human-readable text format and encodes it to the corresponding binary format. The encoder must be configured with the specification's meta-data to generate the proper addresses for the named streams. The decoder works similarly on the output stream: It takes the binary data and decodes it into a human-readable text format.

Chisel's builtin simulation engine Treadle[2] is used for fully automated integration tests. Treadle only simulates the semantics of the Firrtl code without simulating the actual FPGA hardware, i. e. the timing is only accurate in terms of clock cycles. Actual hardware paths on the FPGA are not simulated. A given TeSSLa specification

[2] `https://www.chisel-lang.org/treadle/`

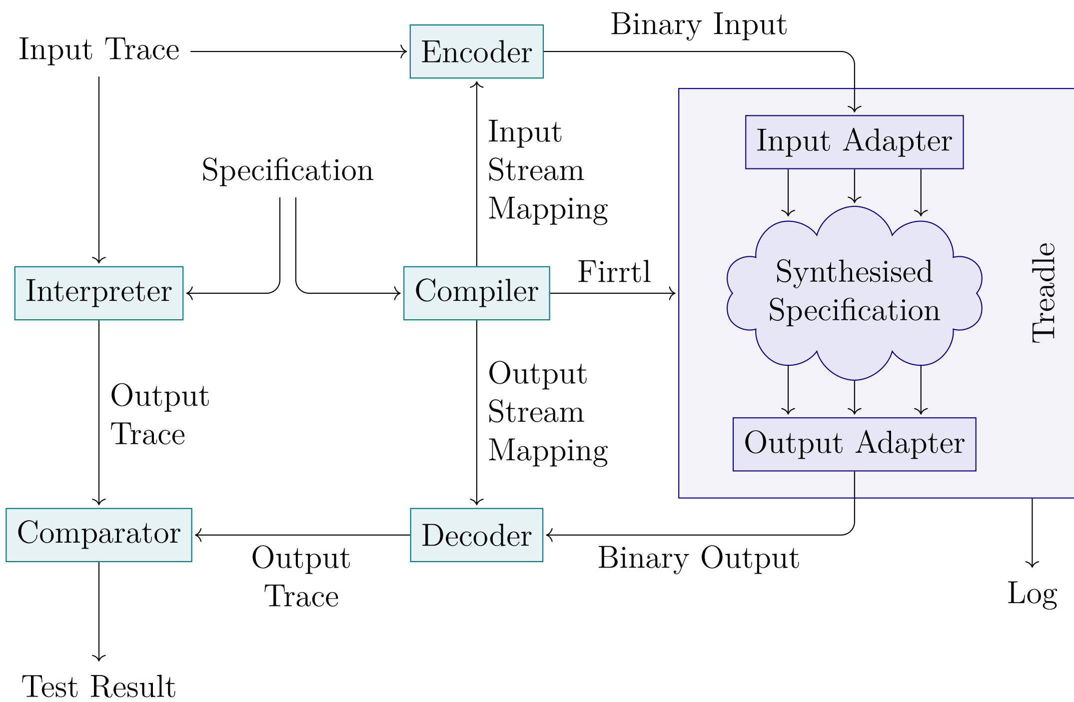


Figure 7.9.: Integration test setup for the simulation in software using Chisel's simulation engine Treadle.

is compiled into Firrtl code for a synthesised hardware module. A given input trace is encoded, fed through the simulator and decoded. The compiler provides mapping information for the encoding and decoding about the addressing of the input and output streams used by the input and output adapters. The same input trace is evaluated for the same specification using the interpreter discussed in Section 4.3 which is used as a reference implementation of the TeSSLa semantics. The output traces are then compared. This entire workflow for these integration tests shown in Figure 7.9 is automated as Scala unit tests. The software encoder and decoder and the input and output adapter provide an integration test setup. Together with the output trace and the comparison result, a detailed log of the inputs, outputs and actions of all the involved channel operators in the operator network synthesised from the specification can be retrieved from the Treadle simulation engine. Every action is timestamped with the current clock cycle.

7.7. Conclusion

The synthesis introduced in this chapter translates a specification's flow graph into an operator network consisting of operators connected by channels. Theorem 7.29 (TeSSLa Operator Networks are Correct) in Section 7.3.2 shows that the operator networks realise the abstract monitoring semantics with the adjusted operators $\mathbf{delayR}^{\ddagger}$ and $\mathbf{last}^{\ddagger}$. The operator networks are compositional: Every channel can be interpreted as an abstract monitoring stream such that a network's outputs can be used as inputs for another network. The central idea of the operator networks is the asynchronous evaluation and the corresponding local synchronisation based on the logical timestamps: The operators send timestamps as indicators of exclusive progress into the channels, potentially followed by a value turning the progress indication into an event with a value at that timestamp. The explicit decentralised progress per channel supports the maximal parallelisation on the hardware.

The tuplification is a specific optimisation for the hardware synthesis to mitigate the additional synchronisation overhead introduced by the asynchronous evaluation. It explicitly synchronises streams that are implicitly synchronous in that events on them occur at identical timestamps. The tuplification at compile-time reduces the number of timestamp comparisons at runtime.

The operator networks are implemented in Chisel and compiled to Verilog or VHDL. A test setup was used to check the correctness of the implementation by comparing its output to the reference implementation provided by the interpreter. The test setup is further used in Chapter 8 to compare the performance of the hardware synthesis with the EPUs and the software compiler.

8 Evaluation

In this chapter, the different approaches to evaluate TeSSLa specifications presented in this thesis are benchmarked. Namely, the following backends are compared:

- The synchronous interpreter from Section 4.3 which builds and evaluates an object graph from the dependency graph at runtime,
- the synchronous compiled software monitor from Section 4.4 which compiles a specification into a single big loop iterating through all timestamps,
- the synchronous EPUs from Chapter 5 which extend the synchronous approach to a pipelining configurable hardware design, and
- the asynchronous synthesised FPGA monitor from Chapter 7.

Although a variety of TeSSLa compilers is used to generate the different evaluation engines, this evaluation focuses on the efficiency of the different approaches in general: The specifications are specifically optimised for every backend, either by the optimisation phases of the corresponding compilers or manually. When comparing different backends, the same input sequences have been used, and the output traces have been checked for correctness to ensure that only equivalent monitors have been compared. This evaluation only considers timestamp-conservative specifications to execute the specifications on all backends.

We are in the setting of online monitoring with finite and relatively small memory. So the general idea is that the monitors analyse traces that are much longer than what the monitors can store. In this setting, the relevant metric used to compare the performance of the different backends is the throughput, i. e. the number of input events that can be processed per time. This evaluation concentrates on the throughput of the actual monitor ignoring any I/O operations: It is ensured that the input events are always available and that the output can always be written.

Possible other metrics of interest like the memory consumption of the monitor or the latency of the monitor, i. e. the time passing between an input event and the corresponding response to that particular input, are not considered: We only examine monitors without complex data structures where all values are either fixed-width integers (64 or 48 bit signed), including timestamps or Boolean flags. All TeSSLa backends only store a fixed amount of data values per operator of the specification,

such that one can compute the memory consumption statically from the specification. Similarly, the latency can be determined statically because it only depends on the size of the specification.

The throughput of the backends is compared in two different ways: First, several specifications derived from real-world use cases of several research projects with different partners are used. Second, synthetic benchmark specifications are used. They allow the variation of just one characteristic property of interest while keeping the other properties fixed. While the real-world specifications provide an overview, measuring the throughput as a function of different specification properties allows us to understand how the different backends perform in general. The considered specification properties are:

- The depth of the specification, i. e. the longest path from an input to an output,
- the recursion depth of the specification, i. e. the longest recursive path inside the specification and
- the parallelism of the specification, i. e. the number of independent paths from inputs to outputs.

Next, we will define precisely how and what was measured in the next section, compare the performance of several benchmarks on the four different backends and finally discuss what we can learn from synthetic specifications with variable properties.

8.1. Measurement Methods

In this thesis, only the performance of the actual monitoring is considered. The aspects of how to get the trace into the monitor and what to do with the output trace are neglected in this discussion in order to focus entirely on the efficiency of the different monitoring implementations.

The metric of interest is the number of processed input events over time. We call this the *throughput* of the monitor. This throughput is measured as the time passed between the first and the last action performed by the monitor during the processing of a known number of events.

In the case of the synchronous software monitors, we know that the monitor is done with all computations after performing the computations for the last timestamp. In the case of the EPU pipeline and asynchronous FPGA network, we have to wait until the output progress reaches the final timestamp. For these monitors, a final maximal timestamp is injected, and we consider the time until this final timestamp was passed

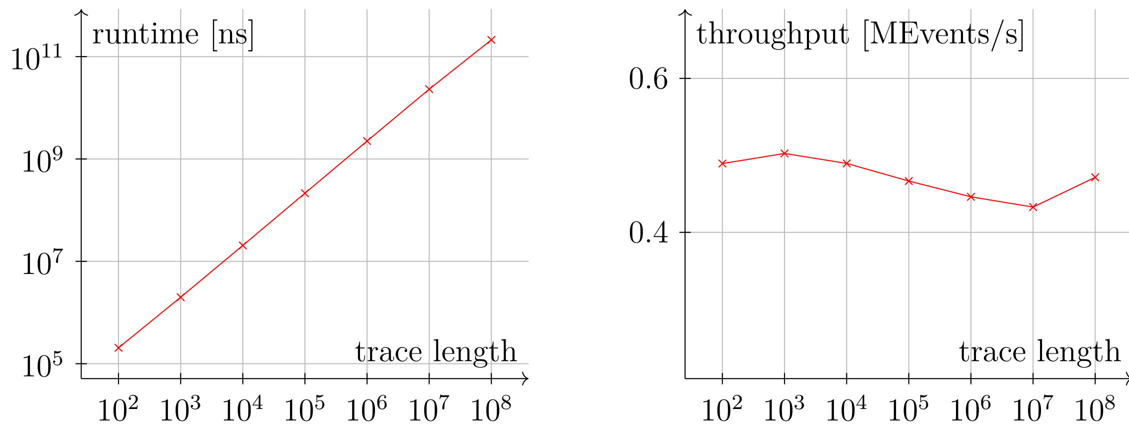


Figure 8.1.: Runtime and throughput of BURST in the interpreter in dependence of the trace length. See Table A.1 in Appendix A.3 for the data table.

through the entire monitor. In order to minimise the influence of the latency onto the measured throughput and get close to the throughput on infinite traces, we consider sufficiently large traces. Depending on the backend, traces of a few 1000 events can be considered sufficient to reduce the influence of the latency enough compared to the runtime of the entire trace being processed by the monitor.

We assume the throughput to be independent of the trace length. Although it is theoretically possible to construct specifications that require certain computations only for certain parts of the trace, it is generally impossible to aggregate growing data because all backends use fixed statically assigned memory. Figure 8.1 shows the runtime of the specification BURST for growing traces. The exact measurements are listed in Appendix A.3 in the appendix. See Section 8.2.1 for a detailed discussion of the specifications used in this evaluation. One can see how the runtime grows linear with the trace length. The constant slope of this line is the throughput in which we are interested. The diagram on the right in Figure 8.1 shows the throughput for different trace lengths. The plot does not show an exact constant value due to several effects affecting the precision of the measurements: The interpreter was executed on a JVM running on a multi-threading computer whose scheduling might interrupt the monitor. See Section 8.1.2 for a detailed discussion on how measurements have been optimised in this setting.

In the case of the EPU pipeline and FPGA synthesis, the execution is not influenced by garbage collection, scheduling or other interferences by operating systems. For each execution of the hardware with the same input trace, the timing is identical to each clock cycle.

8.1.1. Event Generators

The input traces used in the benchmarks are synthetically generated. Generating synthetic events makes the benchmarks independent of real trace sources: It ensures the availability of events without any blocking I/O and thus allows the evaluation to focus entirely on the performance of the different monitor engines. However, this approach has the drawback of less realistic scenarios and repeated input patterns. The input generators simulate realistic inputs to mitigate these problems: They create complex and varying patterns with random number generators.

A pseudorandom number generator (PRNG) with a fixed seed generates entirely deterministic and reproducible but still realistic traces. The xorshift PRNGs [Mar03] can be implemented in software and hardware.

Starting with an initial fixed seed in the variable m with every need for a new random number the following computation is performed:

$$\begin{aligned} x_1 &= m \veebar (m \gg 13) \\ x_2 &= x_1 \veebar (x_1 \ll 7) \\ m &= x_2 \veebar (x_2 \gg 17) \end{aligned}$$

The variables are unsigned 64-bit integers. The operator $\veebar$ denotes the exclusive disjunction, i. e. the XOR operation. The operators $\ll$ and $\gg$ denote the unsigned left or right bit shift, respectively.

The generators were derived from the actual inputs used in the use cases from which the specifications have been derived. The specifications are discussed in detail later in this chapter, and the corresponding generators are given in Appendix A.2.

The output of the monitors was written to a text-based trace format used to check the monitors for correctness. However, for the performance measurement, the output was entirely neglected, again to eliminate any I/O dependencies. Instead of storing the actual output trace, the generated events' values and timestamps are aggregated into a single value. This value serves two purposes:

1. It can be used to ensure that the monitor is still operating correctly and
2. prevents any included optimisation steps from removing parts or even the entire monitor.

8.1.2. Interpreter and Compiler

The interpreter and the compiler are software running on a multi-threading operating system. The benchmarks were executed on a MacBook Pro (15-inch, 2018) with

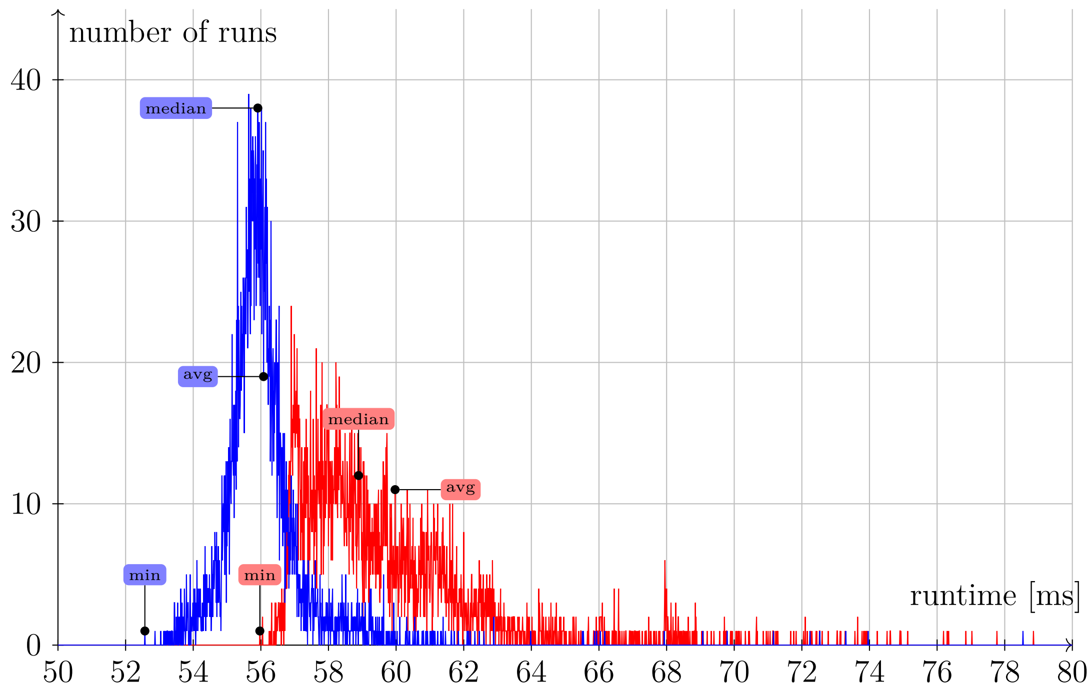


Figure 8.2.: Distribution of the runtime of LONG (with a fixed specification depth of 1) compiled for ■ Java (min: 55.97, median: 58.89, avg: 59.97) and ■ Rust (min: 52.57, median: 55.91, avg: 56.08), aggregated in groups of 10 µs.

2.6 GHz Intel Core i7 and 32 GB 2400 MHz DDR4 RAM using Java 1.9 and Rust 1.44.

The time was measured using `System.nanoTime()`. On Mac OS X, this method uses `mach_absolute_time()` which relies on the processors' time stamp counter and thus provides accuracy on the level of CPU cycles [Lar14].

We take the minimal time of multiple runs because we perform the same computation in multiple runs. Under the assumption that the implementation is correct, the fastest run is as close as possible to the actual time of an uninterrupted execution. See [Bak20] on a detailed discussion of this approach.

The approach is illustrated by Figure 8.2. The plot shows the distribution of all individual execution times of the specification LONG (see Section 8.3.1 below) aggregated in groups of 10 µs. The plot shows the typical right-skewed distribution of multiple executions of the same deterministic computation, which cannot be faster than the optimum but gets randomly slowed down by interruptions of the scheduler and the operating system. As one can see in the plot, using the average runtime of multiple runs would be biased by this right-skew. The median is less affected by this issue, but one can see in the plot that it is still a significant amount away from

the minimum. On the other hand, the minimum can be less stable because it is only based on a single data point. For this evaluation, the minimum was chosen because it provides the best estimate of the optimal, uninterrupted runtime.

Interpreter

We use the simple Scala interpreter presented in Section 4.3 which differs from the official TeSSLa interpreter available online[1] in the absence of a compiler. The official TeSSLa interpreter compiles a textual representation of a TeSSLa specification into an object graph which is then evaluated. The interpreter from Section 4.3 uses a straightforward internal Scala DSL to build the object graph directly, which is then evaluated. The main difference regarding the performance of the two approaches is the evaluation of lifted functions. While the official TeSSLa interpreter compiles functions on primitive data values into an object graph which is then evaluated with every call of the function, the TeSSLa interpreter used here can rely on Scala functions as basic functions on data values, which is much faster.

This evaluation focuses on comparing different approaches to implement the TeSSLa operators. Hence, the simple internal Scala DSL presented in Section 4.3.4 renders the more relevant results as it shows the limits of the approach using message passing directly on the object graph derived from the dependency graph.

Compiler

For the evaluation of the compiler, a slightly more advanced version of the approach presented in Section 4.4 was used, which was developed in [Kal19]. It extends the presented approach with proper runtime error handling and, in some cases, relies on comparing timestamps instead of splitting the operations into the computation of the stream's value and the updated memory value.

For the software compiler, the specifications were slightly rewritten such that they do not lift complex functions but only basic operators. As Theorem 3.51 (Signal Lift and Default) in Section 3.4.2 shows, this does not limit the expressiveness but would limit the performance of the other backends because fewer optimisations directly on the TeSSLa dependency graph are possible. However, in the case of the software compiler, this avoids the need to translate functions on data values. As discussed in Section 4.4 the translation generates a single big loop with all variables being local variables inside a function, which allows many optimisations to be performed by the LLVM backend and the Java JVM JIT.

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

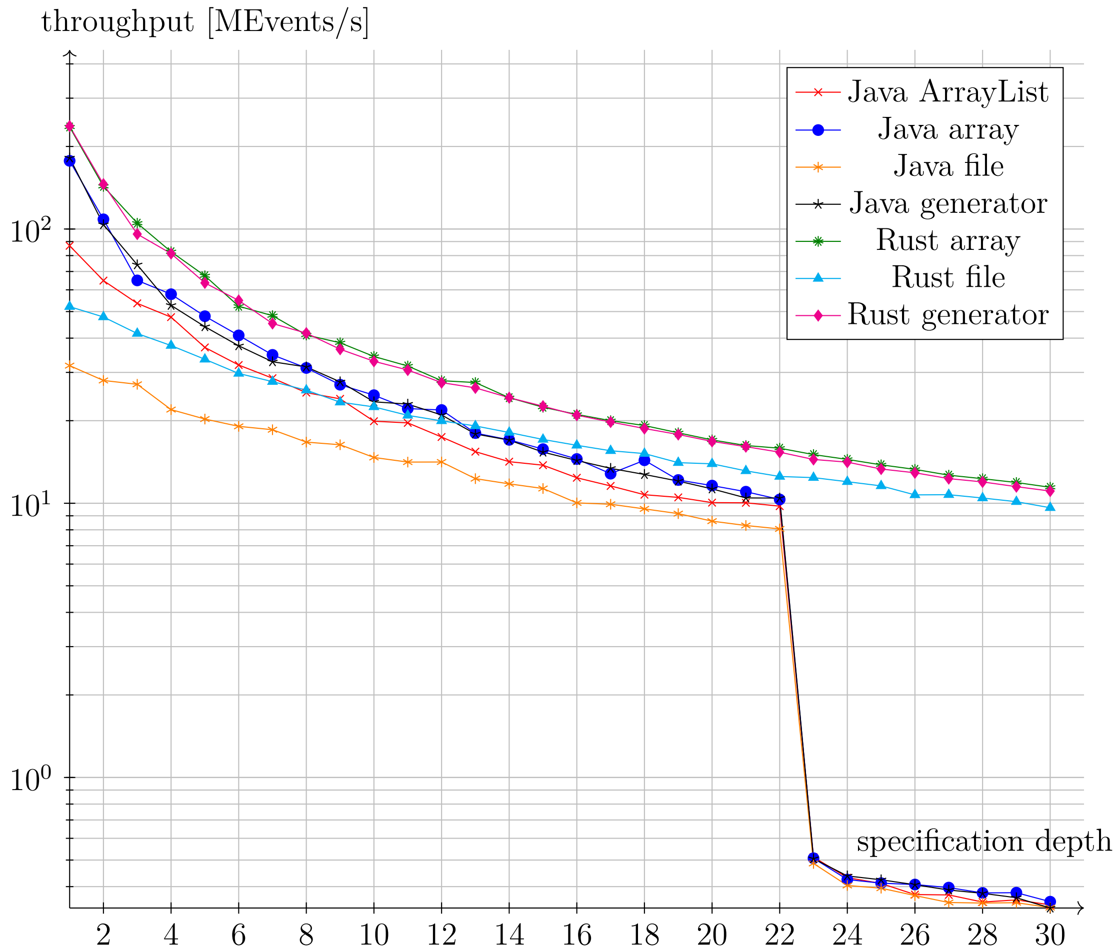


Figure 8.3.: Throughput of LONG in dependence of the specification depth with different compilers and I/O options. See Tables A.8 to A.14 in Appendix A.3 for the data tables.

The compiler can generate Java and Rust code, and different forms of event generators have been evaluated:

File The input events are read from a binary file written by the generator before starting the measurement.

ArrayList The input events are read from an ArrayList, i. e. Java's resizable-array implementation of a List, created by the generator before starting the measurement.

Array The input events are read from an integer array created by the generator before starting the measurement.

Generator The input events are generated during the measurement without any storage.

The last option – generating the events on the fly – is the fastest and less intrusive way. This approach eliminates all I/O operations from the measurement and, at first glance, seems to be the fairest comparison with the hardware backends because they use the same approach. However, while it is a reasonable assumption that events on the hardware can be fed into the system with the cycle speed of the dedicated hardware, this is not true with a CPU: If the events are generated by the same CPU which monitors them, then the events are only stored in the local cache of the CPU, which is unrealistic for actual use cases where events must be read from an external trace source. While these concerns seem reasonable, they cannot be rectified by the measurements:

Figure 8.3 shows the specification LONG (see Section 8.3.1 below) with varying specification sizes and the Java and Rust software monitors using the different event sources discussed above. See Section 8.2.1 for a detailed discussion of the specifications used in this evaluation. The raw data is included in Tables A.8 to A.14 in Appendix A.3. The throughputs for generators and arrays can be considered equivalent for both Java and Rust.

Note that distinguishing between an actual array and an ArrayList only happens for the Java case. After all, an ArrayList in Java consists of an array on the heap storing pointers to the actual integer values because Java data structures can only store objects. As a result, one can see a drastically worse performance of the ArrayList. Rust data structures can directly allocate memory on the heap for the stored values, making them effectively equivalent to an array.

Using files as input sources, on the other hand, is significantly slower than reading the events from memory structures due to the additional I/O operations and corresponding interactions with the operating system. Thus, the array event source was chosen for the compiler and the interpreter to compare the different backends. One

could argue that systems with direct memory access could be used to gain a similar effect in actual implementations.

Comparing Rust with Java, one can see that the static compile-time optimisation performed by Rust and especially the LLVM backend of the Rust compiler gains better results than the JIT compiler and optimiser of the JVM. This result is not surprising considering that the JIT compilation is performed at runtime and thus designed to be very fast to avoid any slowdowns that the user would recognise. The Rust compilation, on the other hand, is performed at compile-time and takes much longer depending on the size of the specification.

Note how the Java throughput drastically drops down to uselessly low throughputs with a specification size of 23 and more. This considerable performance drop is due to the `HugeMethodLimit` of 8000 bytes. The JIT optimisation is not executed for very long methods whose byte code is above this limit. [EGN18, Chapter 10: Understanding JIT Compilation] All in all, the throughput of Rust is higher, and the graphs are more stable. Rust was chosen as the target language for the compiler for the comparison with the different backends.

8.1.3. EPUs

Accemic provided the following setup on a Xilinx XC7V585T of the Virtex 7 series with a clock speed of 100 MHz: Events can be written via USB into a large input buffer before the actual measurement. The events are read from the input buffer during the measurement, and the final EPU writes the output events into a similar output buffer.

The output buffer is continuously read by software. The output buffer might overflow, but that does not affect the throughput measurement. The throughput is measured on the hardware at the input buffer and provided in the form of an event rate. As discussed earlier, we use long traces to mitigate any latency effects. We use traces of length 20 000 because the measured throughput is sufficiently independent of changes in trace length at this length.

Accemic has provided this setup, and the author has no access to the source code of the EPUs. All information about the inner workings of the EPUs are taken from publications [DDG+18, CHS+18, DGH+17] and patents [Weia, Weib] as well as private communication with Albert Schulz and Alexander Weiss from Accemic.

Simulation

Additionally, the hardware benchmarks were performed on a hardware simulation. The simulation is a timing-accurate simulation built with Vivado Simulator and was provided as a closed source binary by Accemic.

The EPU simulation is a perfect simulation of the EPUs regarding the timing. Since tests have shown that the measured throughput on the hardware could be reproduced in the simulation, further benchmarks were only performed in the simulation. The simulation runs with 200 MHz, but the results were converted to a clock speed of 100 MHz for a better comparison with actual hardware measurements.

In the simulation, one can see every message passing through the system. This information allows even better estimations of the throughput on shorter traces since the final dummy message could be ignored in the measurement.

8.1.4. FPGA Synthesis

The test setup described in Section 7.6 was used as follows: Instead of an input adapter, an event generator was implemented in hardware and used as the event source. As mentioned above, the Xorshift PRNG can be implemented with slight adjustments in hardware, too. Instead of the output adapter, a special aggregating module was used, which performs a similar task as the event aggregators in the software benchmarks described above: They ensured that every output was used, and the optimisers could not remove it. A statistics module was added, counting the number of cycles passed between the first and the last event. The event generators and the aggregating and statistics modules were developed in the context of [Buc20].

The benchmark was run on a Xilinx XC7A200T-FBG676 of the Artix-7 series on the Artix-7 AC701 evaluation board with a clock speed of 100 MHz.

Simulation

In addition to the actual hardware, a simulation was used: Chisel's built-in simulation engine Treadle (see Section 7.6) simulates the timing accurately regarding the clock cycles, but actual pathways on the hardware are not simulated. So assuming that the generated design can be synthesised with the given clock speed, the simulation is timing-accurate.

Instead of the hardware input generator and event aggregator, proper input and output adapters were used in the simulation. Since the simulation only simulates the

clock cycles, waiting for I/O does not affect the measurement because the simulation does not continue with the next clock cycle until all the required input data is loaded and encoded. The simulation reads events from a text-based trace file and feeds them into an input adapter. The output is passed through the output adapter and converted into a text-based trace file which allows easy checking for correctness as described in Section 7.6. Since the measurements performed on the actual hardware could be reproduced precisely in the simulator, further measurements were only performed using the simulator. During the execution of the simulator, one can see every message passing through the system, which allows very accurate measurement of the throughput on rather short traces.

8.2. Real-World Specifications

This section discusses evaluations based on specifications derived from real-world use cases of several research projects with different partners.

8.2.1. Specifications

The following specifications were used in the evaluation:

EVENTCHAIN. This specification is based on timing measurements of event chains performed in the ARAMiS II project [BB18]. An event chain is a causal sequence of events. The events are gathered from a system that consists of three independent processing cores communicating via queues implemented in shared memory. Figure 8.4 shows the architecture on the right: Every time the hub gets input data from the network, it writes the new data into the first queue. The following two apps read data from their queue and write their results to the next queue. Other than the hub that only writes something into the queue when new external events arrive, the apps write their input periodically to their output queues with every time slot. The specification tracks an external input event along this causal effect chain and measures the time it takes such an external input event to cause an external output event, i. e. the time between the hub writing something to the first queue and the hub reading the corresponding causal event from the last queue.

RUNTIME. Like the previous one, this specification is based on experiments run in the context of the ARAMiS II project. It computes the time passed between two events and outputs this time for every occurrence of the second event. The canonical example for such a scenario is measuring a function's runtime, i. e. the time passed between the call of the function and its return. Figure 8.5 shows the dependency graph of this specification on the left. Note that the dependency

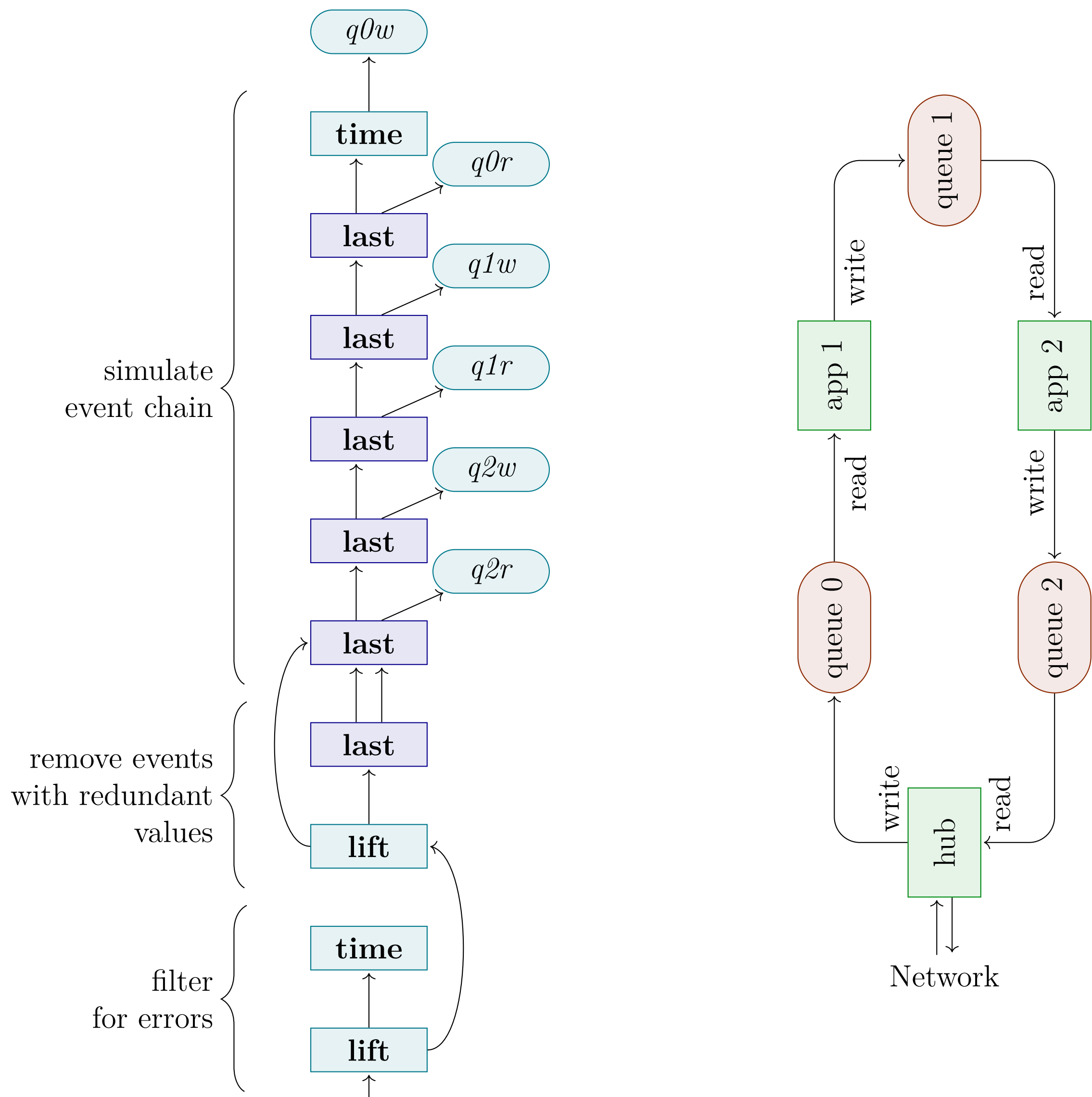


Figure 8.4.: Dependency graph of the specification EVENTCHAIN on the left and simplified architecture of the system under test on the right. Abbreviated stream names of the form *q1r* encode reading of queue 1.

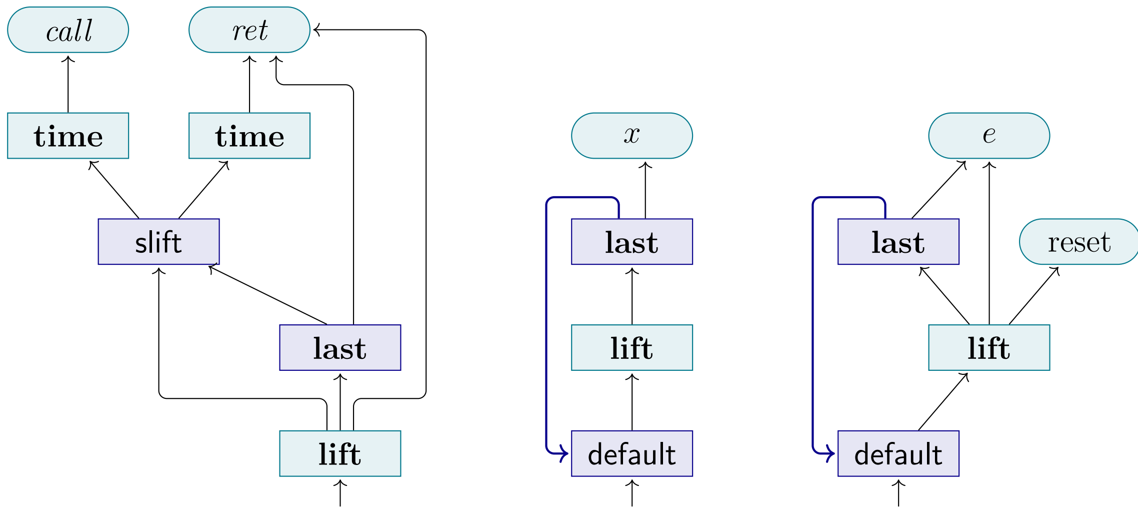


Figure 8.5.: Dependency graphs of the specifications RUNTIME, TOGGLE and RESETCOUNT.

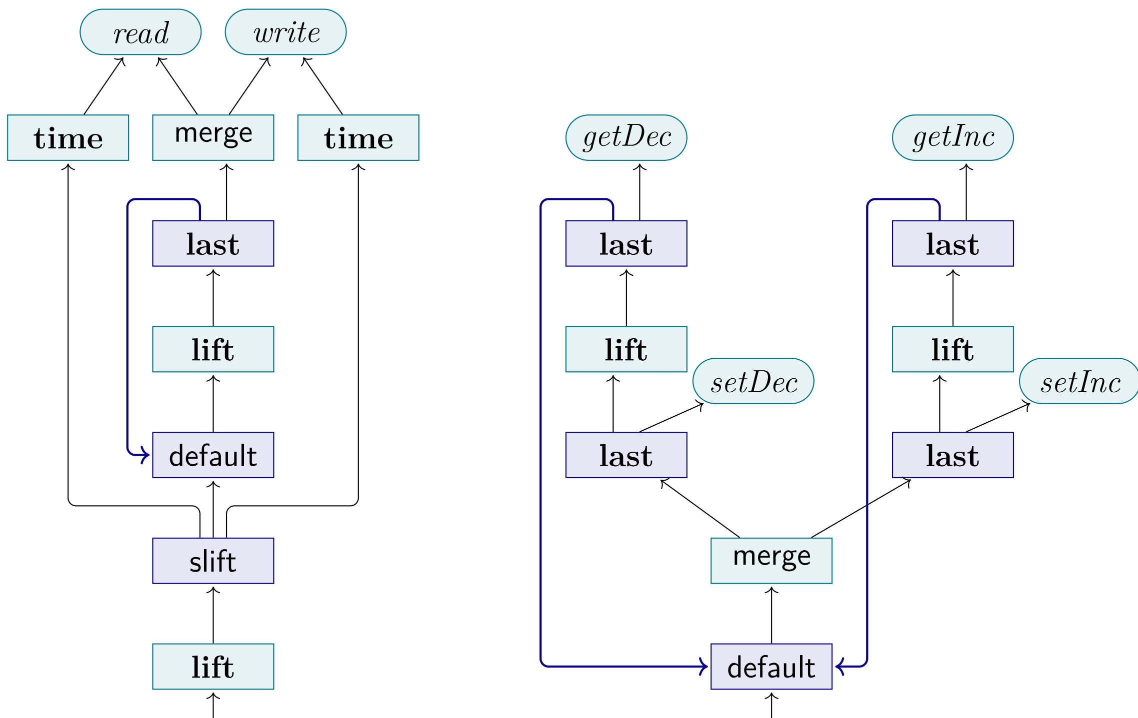


Figure 8.6.: Dependency graph of the specifications ROSACE and INCDEC.

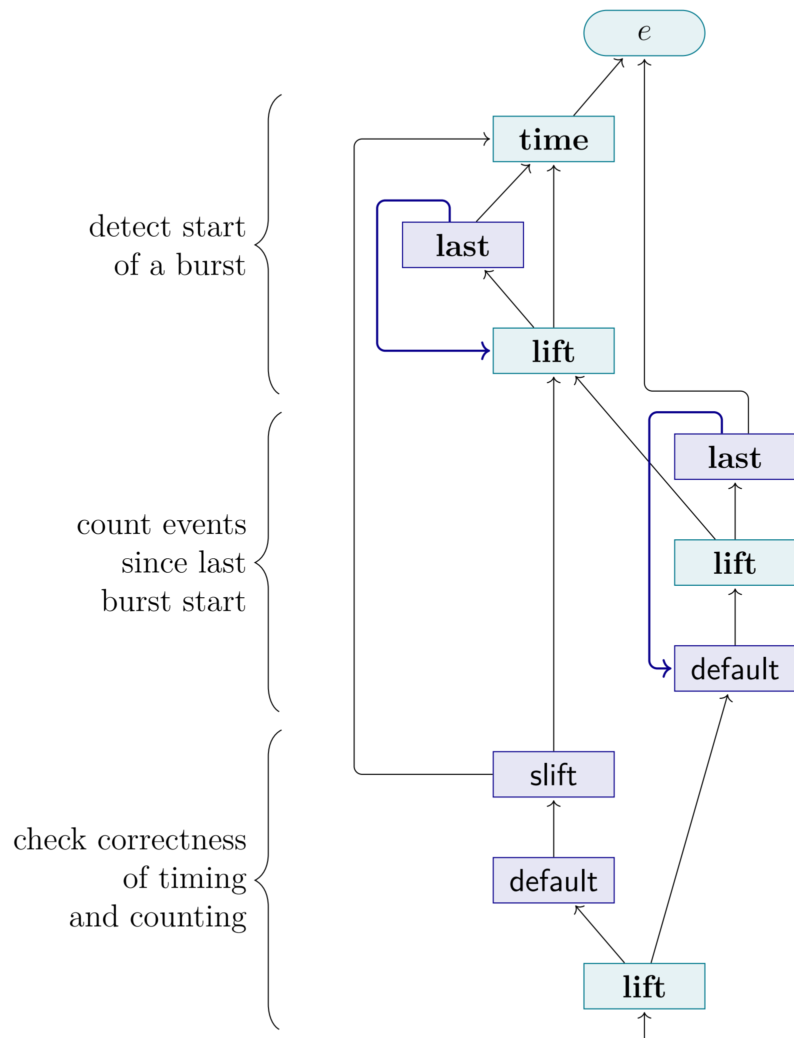


Figure 8.7.: Dependency graph of the specification BURST.

graphs of EVENTCHAIN and RUNTIME do not contain any cycles because no data is aggregated along the trace.

TOGGLE. The simplest possible recursive specification is a counter modulo two, i. e. a Boolean flag being toggled with every event. This specification is included in the benchmark set because it is the core recursive component of the ROSACE specification discussed next. Figure 8.5 shows the dependency graph of this specification in the middle. The recursive dependency is highlighted in blue. The blue arrow is the only arrow going downwards.

ROSACE. The ROSACE case study [PSG+14] was used as an avionic example in the European research project COEMS. The case study defines a longitudinal flight controller designed in Matlab Simulink. Five filters read input values, and their output values are combined into the controller's output. In the implementation analysed in the CONIRAS project, these filters were implemented in different threads running on different cores. The specification checks that the interlocking of the different filters is correct, i. e. that the primary process waits for every filter to finish before combining the outputs into the controller's output. The specification included in this benchmark is the core of a more complex one and checks that read and write events are occurring in the correct pattern using the TOGGLE specification to keep track of whether an even or odd number of input events has been seen. Figure 8.6 shows the dependency graph of this specification on the left. The TOGGLE network in the middle is the only recursive part of this specification.

INCDEC. This specification keeps track of two counters. Like the previous specification, this one is taken from experiments performed in the context of the COEMS project. The application under test performs increment and decrement operations on two separate threads using a shared variable. The specification simulates the state of the shared variable and simulates the increment and decrement operations. This experiment's goal in the COEMS project was to test the performance of the event source used in the project and compare the output of the specification with the output of the running application. It was included in this benchmark because it is a real-world example of two non-synchronous interlocked recursions. Figure 8.6 shows the dependency graph of this specification on the right. Note how the two recursive loops are governed by two individually triggered **last** operators and delayed by two additional individually triggered **last** operators, which are part of the recursive loops, too.

RESETCOUNT. The most basic recursive building blocks are those counting events or summing up events values along the trace. The TOGGLE specification is the most basic form using a modulo two counter. The RESETCOUNT counts the number of events and has an additional input resetting the counter. This specification is the basic building block of the BURST specification discussed next with the addition

of simultaneous input events. Simultaneous events do not occur in the specification BURST. The resetting event always occurs with a simultaneous counting event, setting the counter to one. In this more general specification, those cases are distinguished: A resetting event resets the counter to zero, but if it occurs with a simultaneous counting event, the counter goes directly to one. Figure 8.9 shows the dependency graph on the right. Note how the central lift depends on the last value and both input values in order to distinguish the different cases discussed above.

BURST. The AUTOSAR Timing Extension [AUT17] and the EAST-ADL2 timing extension TADL2 [GDPM13] were examined in the context of the ARAMiS II project as a specification language for timing behaviour of automotive software. This specification is based on the burst pattern introduced by these timing extensions: A certain amount of events is allowed during a burst until the end of the burst period. This period is followed by a silence period which must not contain any events. Afterwards, the next burst period starts with the next event. There are two ways to violate this pattern:

– Too many events can occur during the burst, or
– the burst period can be too long, resulting in events occurring in the silence period.

Figure 8.7 shows the dependency graph of this specification. It consists of three basic blocks:

a) The start of a new burst is identified by comparing the current event's timestamp with the previous burst start.
b) The number of events until the burst start is counted.
c) The duration of the burst and the number of events seen in this burst is compared with the allowed values.

The specifications are formally defined in Appendix A.1 in the appendix.

8.2.2. EPU Optimisation for Simple Recursions

Section 5.9 introduced the recursion optimisation with the foldLift operator. The section evaluates the effect of this optimisation. The dependency graph of the specifications EVENTCHAIN and RUNTIME contains no cycles, such that the optimisation does not apply here. However, they are included for reference to compare the performance of cyclic and acyclic specifications. The specification INCDEC is mutually recursive and hence not rewritten with the foldLift operator. The specifications TOGGLE, ROSACE and RESETCOUNT each contain one recursive specification, i. e. cycle in the dependency graph. We compare the performance of these specifications with and without the usage of the foldLift operator. The specification BURST contains

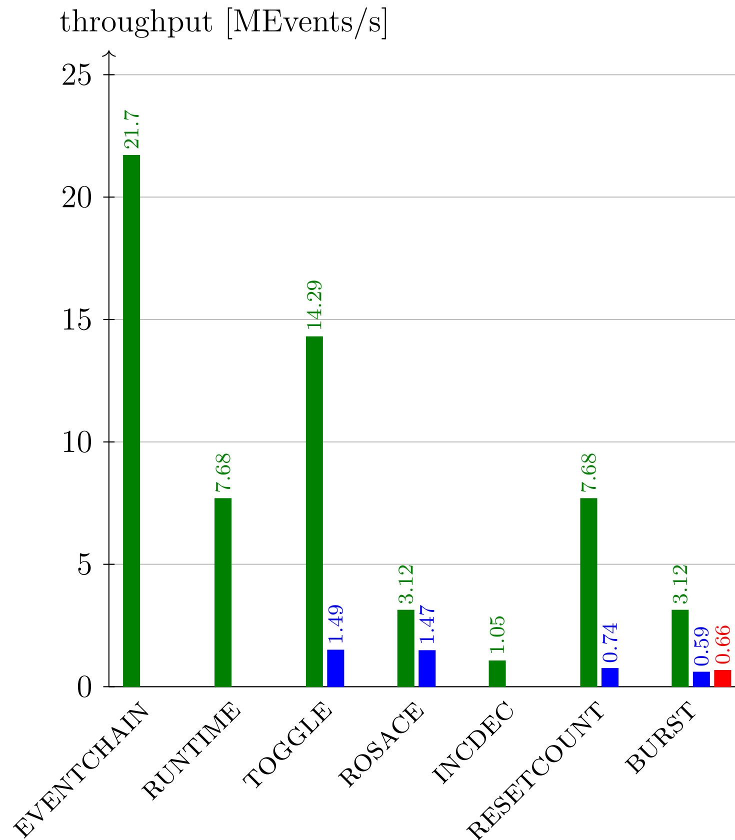


Figure 8.8.: Throughput of different specifications on the EPU backend ■ with foldLift, ■ without foldLift, and ■ with foldLift used only for the counting. See Tables A.3 and A.4 in Appendix A.3 for the data table.

two cycles in the dependency graph, which can both be expressed using foldLift. We compare three cases:

– The entirely unoptimised specification,
– the variant with the unoptimised detection of the burst start and the optimised event count, and
– the variant where both recursions are optimised.

The second case is relevant because counting events is a fairly common operation available as a macro in TeSSLa's standard library and thus available in optimised versions. The detection of the burst start, however, is specific for this use case and thus rewritten using foldLift manually.

Figure 8.8 shows the results of the comparison: The foldLift optimisation improves the throughput by up to an order of magnitude. The specification ROSACE is not only slowed down by the recursion but also by the complex structure of the dependency

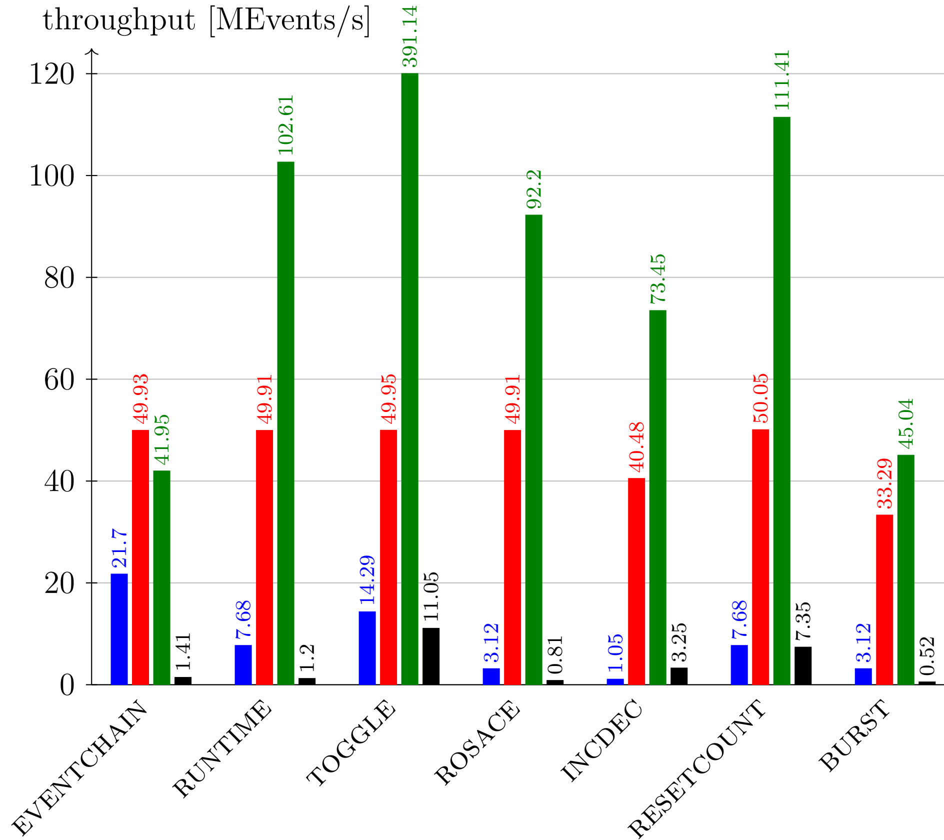


Figure 8.9.: Throughput of different specifications on different backends: ■ EPU, ■ FPGA, ■ compiler and ■ interpreter. The vertical axis is cropped at 120 MEvents/s. See Table A.2 in Appendix A.3 for the data table.

graph, making the optimisation less effective in this case. Similar reasoning applies to the BURST. We can see that the optimisation works best for that specification if applied to all recursive definitions. The partial application only to the counting part of the specification – shown in red in the plot – has nearly no effect because the remaining recursion still significantly reduces the throughput.

As a consequence of the above experiments, the optimisation with foldLift was used in all following experiments whenever possible, i. e. for all recursive specifications except INDEC and RECURSION (see below).

8.2.3. Backend Comparison

The results of measuring the specifications on the four different backends are shown in Figure 8.9. The raw data is included in Table A.2 in Appendix A.3.

First of all, we can see that the throughput on the FPGA is always in the range of 50 MEvents/s. This is the theoretical maximum with a clock speed of 100 MHz and an event encoding alternating between the event's timestamps and values with every other cycle. The specifications INCDEC and BURST drop significantly below this maximum. In both cases, queues are used to distribute the events to multiple targets because some operations depend on several streams. Introducing additional queues, in general, does not affect the throughput, but introducing additional queues into recursive cycles affects how many clock cycles are needed to evaluate the recursion.

Next, we can observe that the software compiler performs rather well. However, we can already see that the size of the specification has a considerable influence on the throughput: While the throughput on the FPGA stays close to 50 MEvents/s, the throughput of the compiled software backend varies from about 40 to nearly 400 MEvents/s. We will investigate this relationship further in the next section.

The EPUs perform rather well and in the same order of magnitude as the FPGA in the cases of the specifications EVENTCHAIN, RUNTIME, TOGGLE and RESETCOUNT. We can observe the maximal throughput for an aggregating specification which has to relate every generated event to the previous event: The specification TOGGLE has a throughput of about 100/7 MEvents/s, which indicates that the internal pipeline of an individual EPU needs seven steps to decode an instruction, fetch the old value from memory and write the new value to the memory again. In the case of the specification EVENTCHAIN, the EPUs can utilise the fact that not every input event starts a new event chain. In case of the specifications ROSACE and BURST the EPUs perform not that well. It can be assumed that this is mainly because of the many dependencies between the operators. It stands out that these two specifications are also those with the smallest throughput on the interpreter. Finally, the specification INCDEC is performing poorly on the EPUs. It is even significantly slower than on the interpreter. This specification entirely consists of two mutual recursions which are not optimised using foldLift. Instead the two recursions are both realised using Definition 5.15 (Commands for Blocking last) from Section 5.6. While expressing arbitrary specifications on the EPUs might be desirable, they do not perform well on mutual recursive specifications. Setups combining preprocessing on the EPUs with further processing in software seem more reasonable.

For direct comparison of the EPUs and the software compiler, one must keep in mind that the software compiler drastically benefits from running specifically one specification while the EPUs can utilise their pipeline structure much better on parallel specifications. See Section 8.3.3 for a detailed investigation on this.

As expected, the interpreter is the slowest of all backends. However, in some simple cases like the specifications TOGGLE or RESETCOUNT, the results are not entirely out of the picture. However, those are precisely those cases where the compiled

backends excel, too. The architecture of the interpreter prevents most compile-time or runtime optimisation from taking place because no static code analysis can identify recurring patterns. The executed code entirely depends on the object graph representing the dependency graph built in the memory at runtime.

8.3. Synthetic Specifications

Synthetic benchmark specifications allow the investigation of the throughput as a function of different specification properties, i. e. the depth of the specification, the recursion depth of the specification, and the number of inputs.

8.3.1. Specification Depth

The specification LONG consists of a variable number of counting blocks chained after another. Figure 8.10 shows the dependency graph of this specification on the left. It is a synthetic specification that does not compute an actual result. Every counting block except the first one produces the same results again. It was taken care of that none of the backends utilises this fact in its optimisations. This specification serves as a simple specification with an adjustable size that works on all backends with a reasonable range of sizes.

Figure 8.11 shows the throughput on the different backends as a function of the depth of the specification. The raw data is included in Tables A.5 to A.8 in Appendix A.3. We can see how the EPUs' and the FPGA's throughput are utterly unaffected by the specification's size due to their pipelining. As discussed in the previous section, they run with their maximal throughput of about 100/7 MEvents/s or 50 MEvents/s, respectively. The interpreter and the compiled backend show a similar behaviour on different scales: The runtime roughly depends linearly on the size of the specification resulting in the depicted decay of the throughput.

Figure 8.12 shows the hardware consumption of the synthesised specification on the FPGA. The space consumption of the actual specification can be almost neglected compared to the size of the Xillybus I/O control logic. (See Section 7.6 for more information on Xillybus.) The relatively large input and output buffers are not synthesised into lookup tables but using dedicated memory modules on the FPGA, making them appear small but distributed all over the hardware layout. The space consumption of the I/O logic and the input and output FIFOs are independent of the actual specification. Figure 8.13 shows how the space consumption of the synthesised specification changes with the size of the specification. One can see that synthesising the TeSSLa operators onto hardware does not consume much space. Even on this

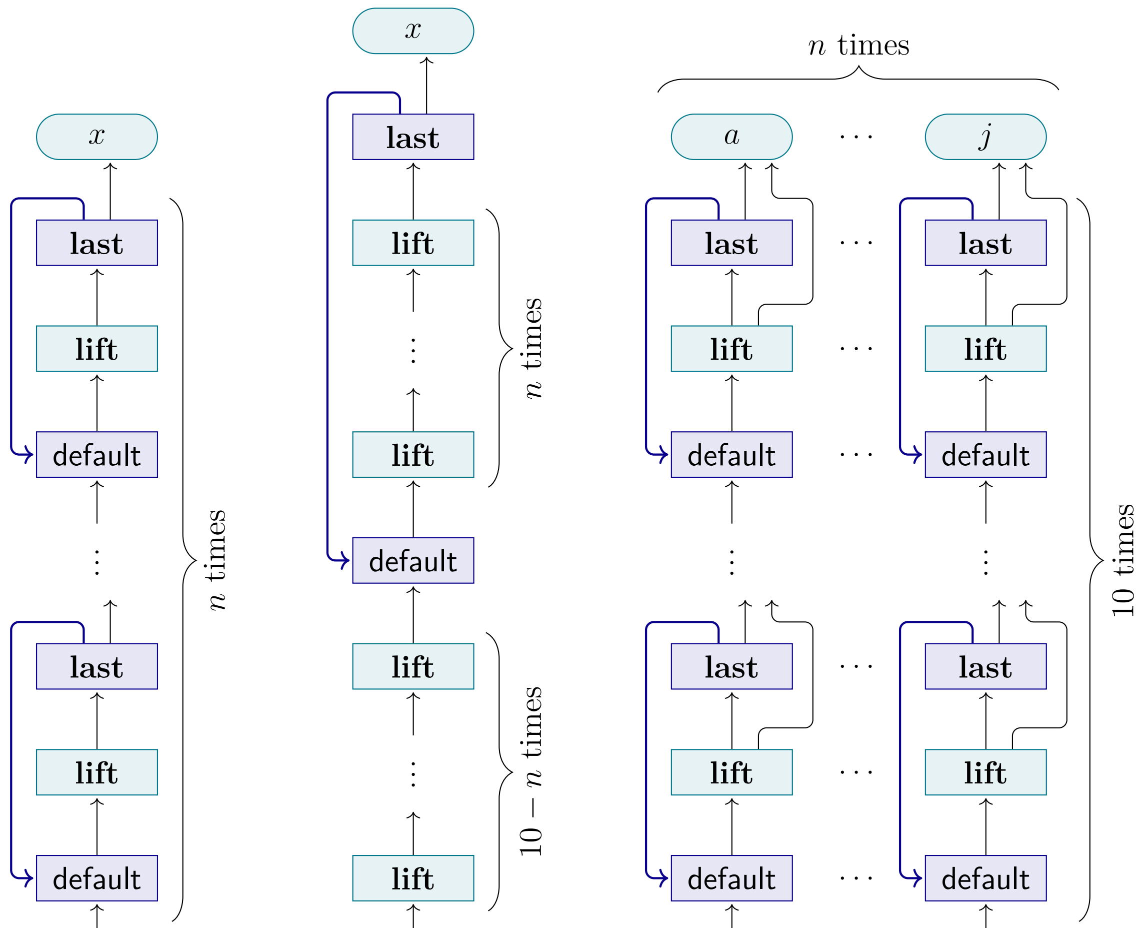


Figure 8.10.: Dependency graphs of synthetic specifications LONG, RECURSION and INPUTS with the parameter n indicating the specification depth, the specification's recursion depth and the specification's parallelism, respectively.

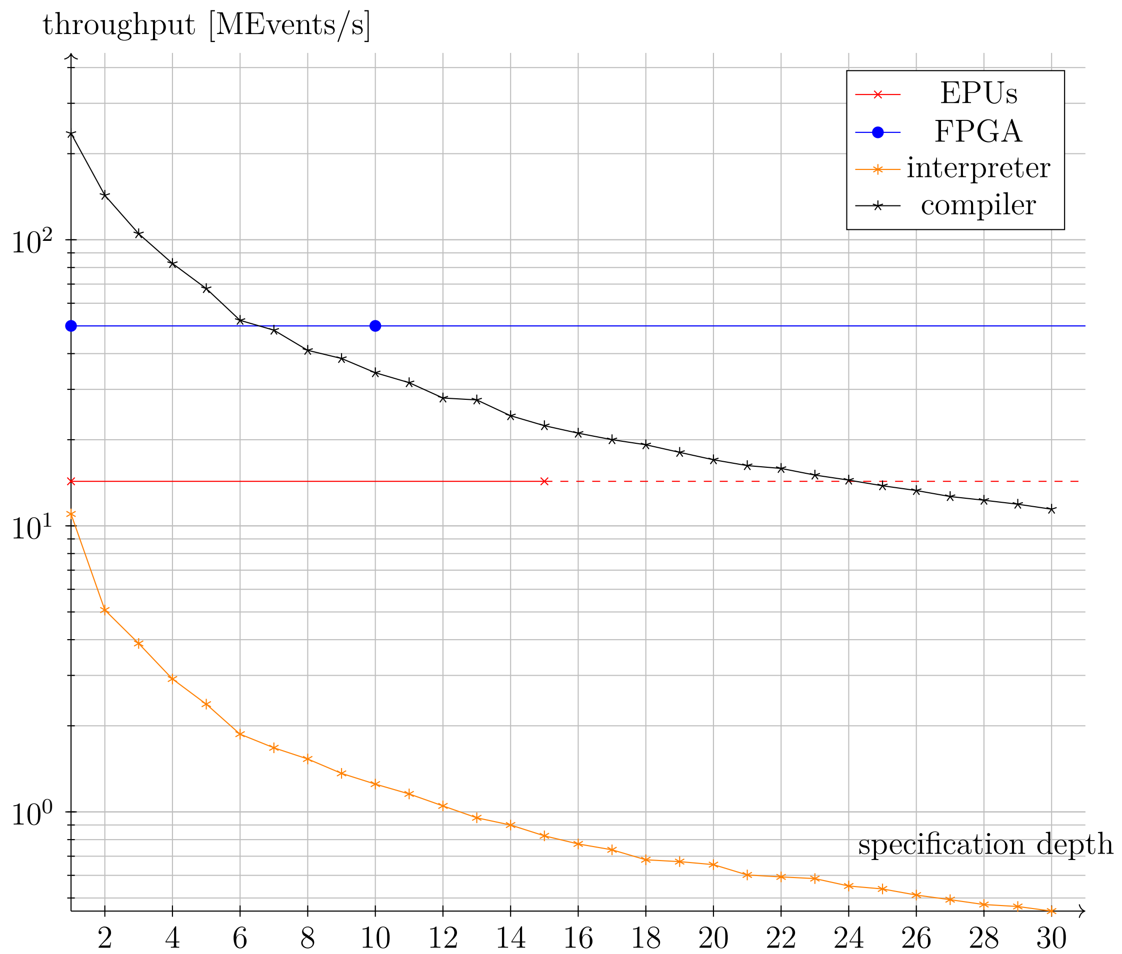


Figure 8.11.: Throughput of LONG in dependence of the specification depth with different backends. See Tables A.5 to A.8 in Appendix A.3 for the data tables.

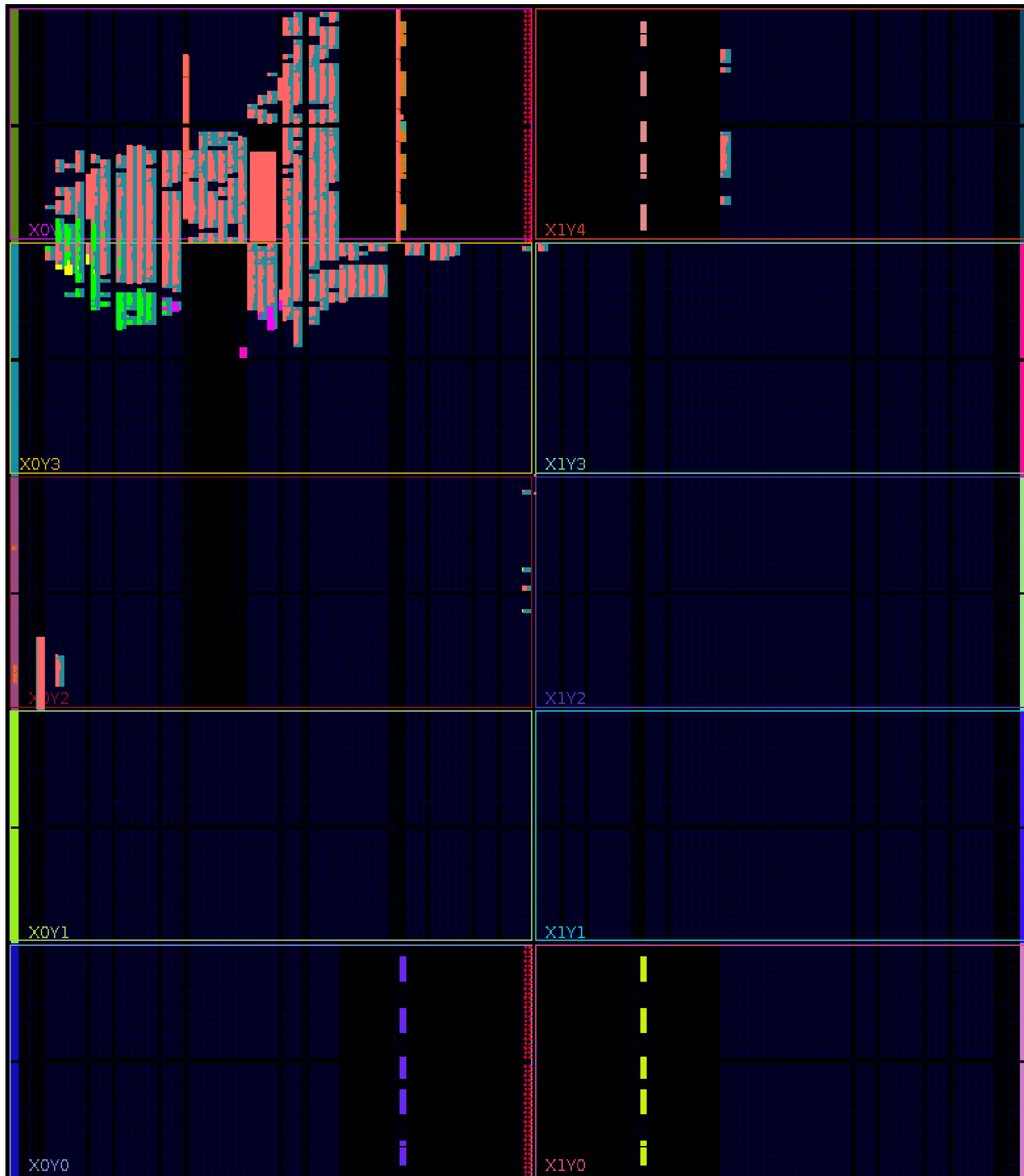


Figure 8.12.: Hardware utilisation of the specification LONG with size 1 visualised in Xilinx Vivado. The following parts of the hardware design are coloured: ■ the input FIFO before the actual monitor, 32 bit × 512 entries, ■ the output FIFO after the actual monitor, same size, ■ the Xillybus I/O logic and ■ the actual specification.

Figure 8.13.: Hardware utilisation of the specification LONG with sizes 1, 10, 100 and 1000 from left to right visualised in Xilinx Vivado manually edited to show only the actual specification without Xillybus and the input and output FIFOs. See Figures A.1 to A.4 in Appendix A.4 in the appendix for the entire visualisations as generated by Xilinx Vivado.

middle-class FPGA hardware, there is plenty of space available. The paragraph on queue depth in Section 7.4.4 describes how the throughput can be optimised by additional queues, which are larger than the trivial queue size 1. However, this might lead to massive consumption of either lookup tables or dedicated memory modules depending on the realisation of the internal FIFOs.

8.3.2. Recursion Depth

The dependency graph of the RECURSION is shown in the middle of Figure 8.10. It is a synthetic benchmark with an adjustable recursion depth. The idea of this benchmark is to keep the size of the specification as constant as possible while adjusting the recursion depth. The additional lift operations could be easily combined into a single lift, and it was again taken care of that none of the compilers does this in an optimisation phase. Since this specification is intended to measure the influence of recursion depth on the throughput, foldLift was not used.

Figure 8.14 shows the throughput of the different backends as a function of the recursion depth. The raw data is included in Tables A.15 to A.18 in Appendix A.3. On the left, we can see that the throughput of the compiled backend is more or less constant, with much noise. Although this was not investigated further in this thesis, this noise likely comes from different LLVM optimisation phases applying different translations for the different specifications.

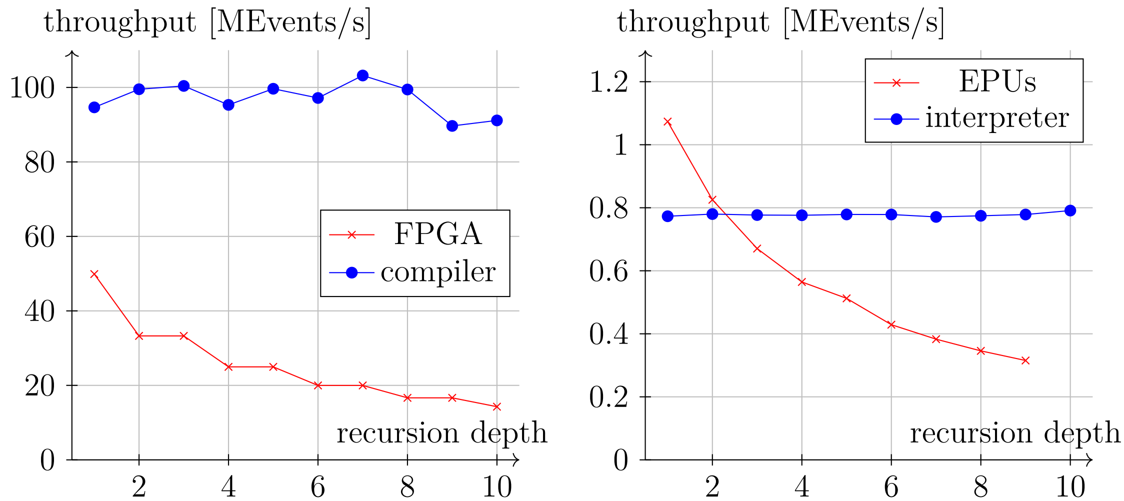


Figure 8.14.: Throughput of RECURSION in dependence of the recursion depth with different backends. See Tables A.15 to A.18 in Appendix A.3 for the data tables.

The FPGA's throughput decays in steps of two because another queue is added into the recursive loop with every other extension of the recursion depth. Every queue adds two additional cycles needed to evaluate the loop because of the alternating encoding of timestamps and values. Increasing the size from 2 to 4 halves the throughput from the maximum of about 50 MEvents/s to about 25 MEvents/s.

On the right of Figure 8.14 we can see the throughput of the EPUs and the interpreter. The interpreter shows the same behaviour as the compiled backend but on an entirely different scale. As discussed in the previous sections, the EPUs are inefficient for unoptimised recursions. The throughput dropping below the interpreter's performance makes clear that the EPUs do not evaluate recursions efficiently.

8.3.3. Number of Inputs

The specification INPUTS consists of n long chains of ten blocks summing up their input event's values. Figure 8.10 shows the dependency graph on the right. The specifications are entirely independent with separate inputs. The considerations on synthetic specifications from the previous two sections also apply here.

Using these long chains allows the hardware backends to utilise their (outer) pipelining more so that we can compare the influence of parallelism on the compiled backend and the hardware backends on a similar scale: Figure 8.15 shows the throughput of

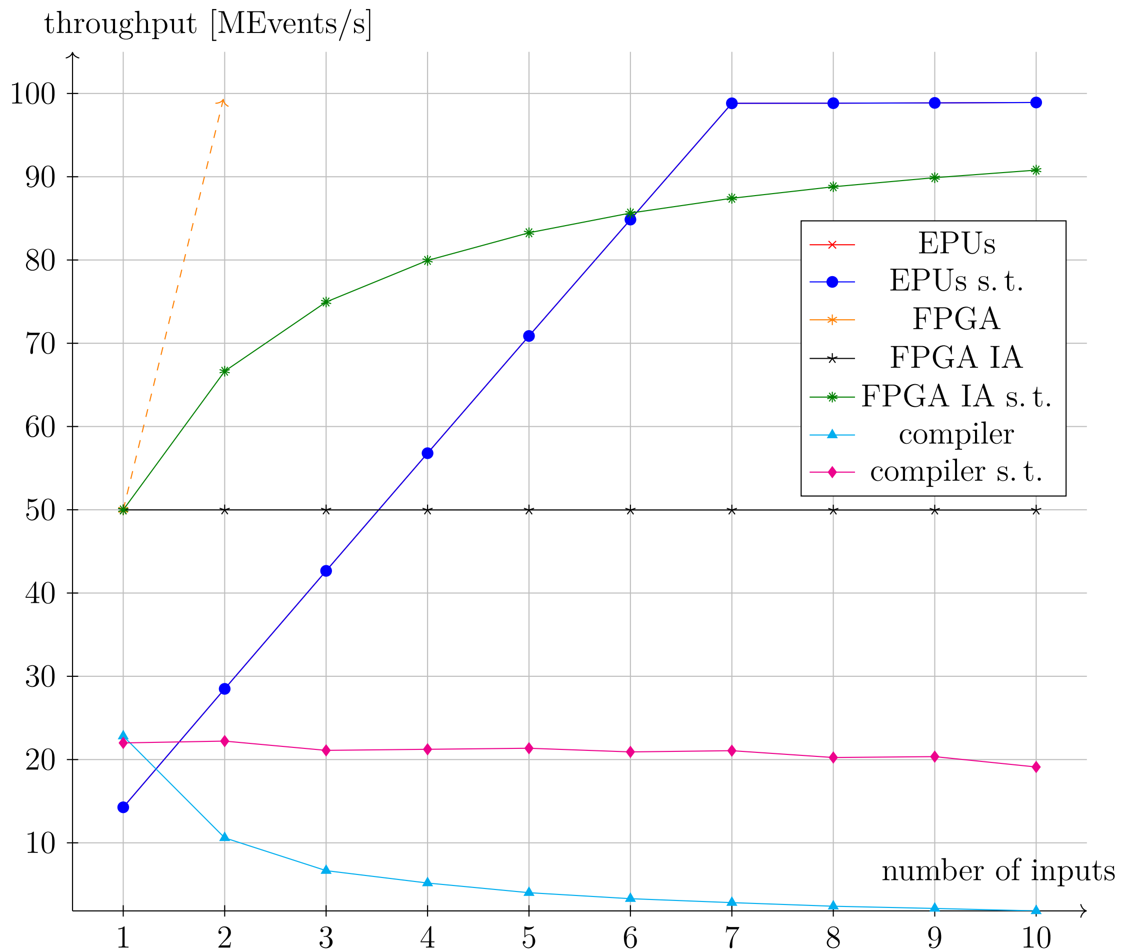


Figure 8.15.: Throughput of INPUT in dependence of the number of inputs with different backends. The abbreviation s. t. in the legend stands for measurements with the same timestamp across all input, and IA indicates the usage of an Input Adapter. See Tables A.19 to A.24 in Appendix A.3 for the data tables.

these three backends as a function of the number of parallel independent inputs of the specification. The raw data is included in Tables A.19 to A.24 in Appendix A.3.

We distinguish two different measurements:

- Using the same timestamps for all parallel inputs, i. e. having an event for every input with every timestamp, and
- using different timestamps for every event, i. e. having a new timestamp for every individual event independent from the input.

As already established in the introduction to this chapter, we measure the throughput, i. e. the number of input events processed per time. Unlike the previous experiments, we now vary the number of input streams. So in the case of entirely independent specification with independent inputs and outputs synthesised on the FPGA, the throughput is directly proportional to the number of inputs. This relation is indicated by the orange plot in Figure 8.15. Without further restrictions of the sources and targets, there is no other upper limit for the throughput than the size of the FPGA. However, this is a theoretical consideration as most applications impose limits on the availability of input events and the processing of output events. Hence, in this case, we do consider at least one aspect of I/O in the measurement: An input adapter (see Section 8.1.4) was used, feeding all of the specifications inputs from a single input data stream. We can see no increase in the throughput for different timestamps and an increase up to about 90 MEvents/s in the case of the same timestamps. This difference can be explained entirely by the event encoding of the shared input stream fed in the input adapter.

For the EPUs, we can see the throughput increases up to nearly 100 MEvents/s with seven parallel inputs. In this case, the internal pipeline of the individual EPUs is used optimally. The EPUs perform with the maximal throughput of 100 MEvents/s, the maximal theoretical speed of processing one event per cycle. Note that there is no difference between the same timestamps and different timestamps because timestamps and values are passed on in parallel between the EPUs.

In the case of the software compiler, we can observe the following effect: We see nearly no influence of the number of inputs on the throughput using the same timestamp. It does not make a big difference if more events are fed through a smaller specification or if fewer events are fed through a larger specification. However, in the case of different timestamps, we see a problematic effect on the compiled backend: Since the compiled backend is synchronous in that all operators are evaluated for every timestamp, a considerable overhead is created because large parts of the specification are irrelevant for a specific timestamp. While this makes the hardware backends faster, it slows down the software backend drastically.

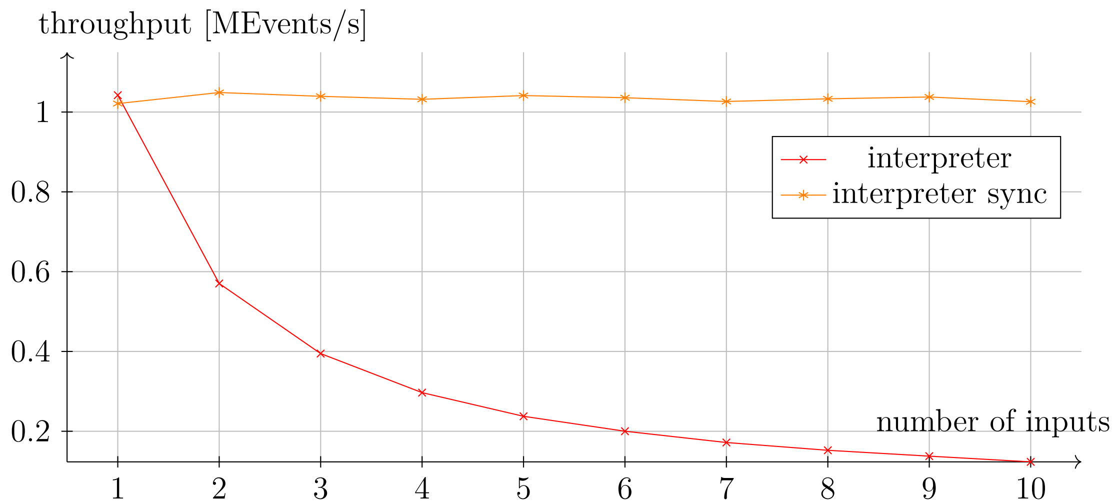


Figure 8.16.: Throughput of INPUT in dependence of the number of inputs with the interpreter. The abbreviation s. t. in the legend stands for measurements with the same timestamp across all input. See Tables A.25 and A.26 in Appendix A.3 for the data tables.

The interpreter shows the same behaviour on a different scale as shown in Figure 8.16. The raw data is included in Tables A.25 and A.26 in Appendix A.3.

For this synthetic specification, an obvious solution would be to execute multiple software backends on the independent input data. However, the same behaviour also occurs for specifications that are not entirely independent but have the same characteristics that large parts of the specification are not executed for certain timestamps. In those cases, one could think of different possible optimisations: Either one could still cut those specifications into separate specifications executed in parallel on different backends that are communicating, or one could add optimised conditions to the generated code, avoiding unnecessary code execution. Both solutions are future work and are not evaluated in the context of this thesis.

8.3.4. Summary

Figure 8.17 shows a qualitative summary of throughputs behaviour on the different backends when properties of the specifications are adjusted.

Interpreter and compiler show the same behaviour on a different scale. The interpreter is significantly slower than the other backends because building and executing an object graph from the dependency graph at runtime prevents most compile-time and runtime optimisations. On the other hand, the compiled backend is much faster because the entire code sits inside one big outer loop using only local variables. This

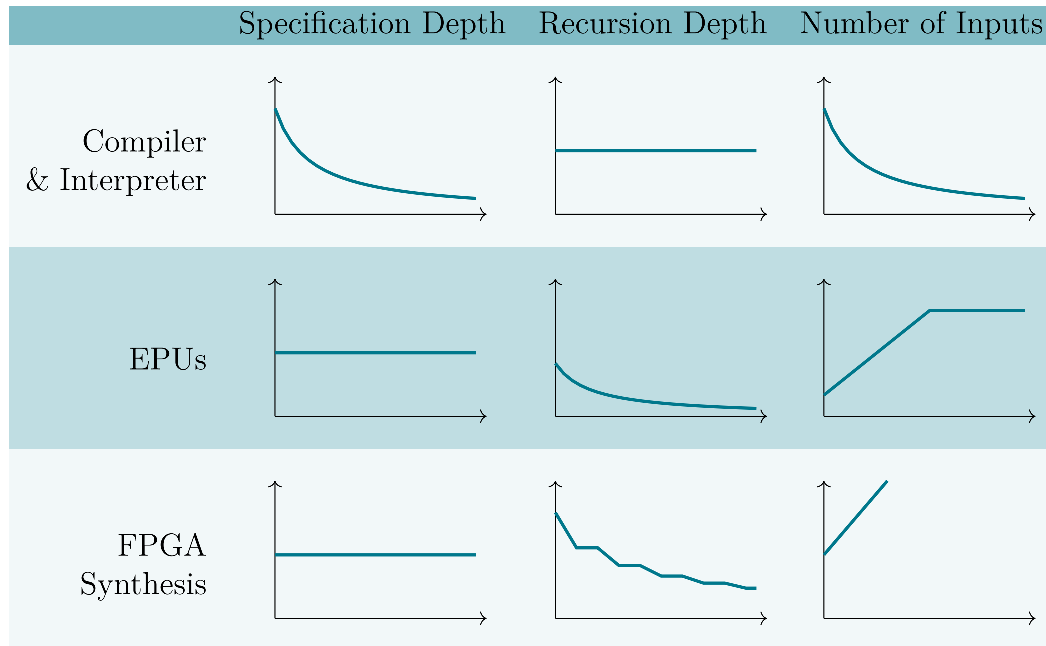


Figure 8.17.: Qualitative overview of the throughput as a function of the depth of the specification, the recursion depth of the specification, and the parallelism of the specification on the different backends.

loop greatly benefits from compile-time optimisation of the Rust compiler and the LLVM backend.

Increasing the size of the specification makes it slower on the software backends. It does not matter if parts of the specification are used or not. The pure size of the specification determines how long one iteration of the loop takes. The synchronous approach is simple and well suited for further optimisation but not well suited for independent events with different timestamps. As discussed in the previous sections, one might use multiple parallel monitors to overcome this limitation.

For the EPUs, we have shown that the general approach of a configurable hardware pipeline works. The throughput is independent of the specification depth. However, recursive specifications are not efficient on synchronous pipelines. Additionally, the general concept of synthesising a configurable network onto an FPGA, which is already a configurable network, adds multiple layers of abstraction to the system, which slows everything down and consumes much space. The EPUs can use their inner pipeline per EPU, resulting in a speedup for parallel specifications up to the theoretical maximum of reading one event per clock cycle. In less ideal situations of not entirely independent specifications conflicting memory access can cause pipeline stalls on EPUs, resulting in smaller throughputs.

We can conclude that the performance of the EPUs highly depends on the structure of the input data and the specification: The EPUs can use their inner pipeline best if the specification consists of many mostly independent parts and the input data is structured so that not every input event is relevant for every part of the specification. Otherwise, the same memory address is used too quickly in succession, leading to the inner pipeline's stalls. As discussed in [WGJ+21], the EPUs were designed for precisely this form of data dependency between the different parts of the specification in combination with such a structure of the input data: The EPUs are supposed to provide a fast engine for evaluating many specifications in parallel on data traces with a high event frequency.

The synthesised FPGA backend is much better suited for complex recursive specifications. Local asynchronous recursions on the FPGA do not slow down the entire computation since the recursion depth only influences the throughput on that particular branch of the specification. Figure 8.17 depicts the worst case of a specification consisting only of one growing recursive loop. The FPGA can facilitate parallelism of the specification. Under the assumption of independent event sources, independent specifications can run entirely independently on the FPGA without influencing each other.

8.4. Comparison of Workflows

This final section of the evaluation considers the different setups and especially the steps involved in compiling a specification for the different backends. The integration setup for the software compiler was presented in Section 4.5, the one for the EPUs in Section 5.10, and the setup for the hardware synthesis in Section 7.6. The following paragraphs summarize the compilation process for each backend:

Interpreter Just like the other backends, the interpreter can be used together with the compiler frontend discussed in Section 4.4.4: The compiler frontend performs parsing, type checking and constant folding. The interpreter interprets the TeSSLa specification processed in this way immediately and without further compilation steps.

Software Compiler The software compiler compiles a TeSSLa specification into source code of the target programming language, in the case of this evaluation Rust. This source code is then compiled using the target language's compiler, i. e. the Rust compiler.

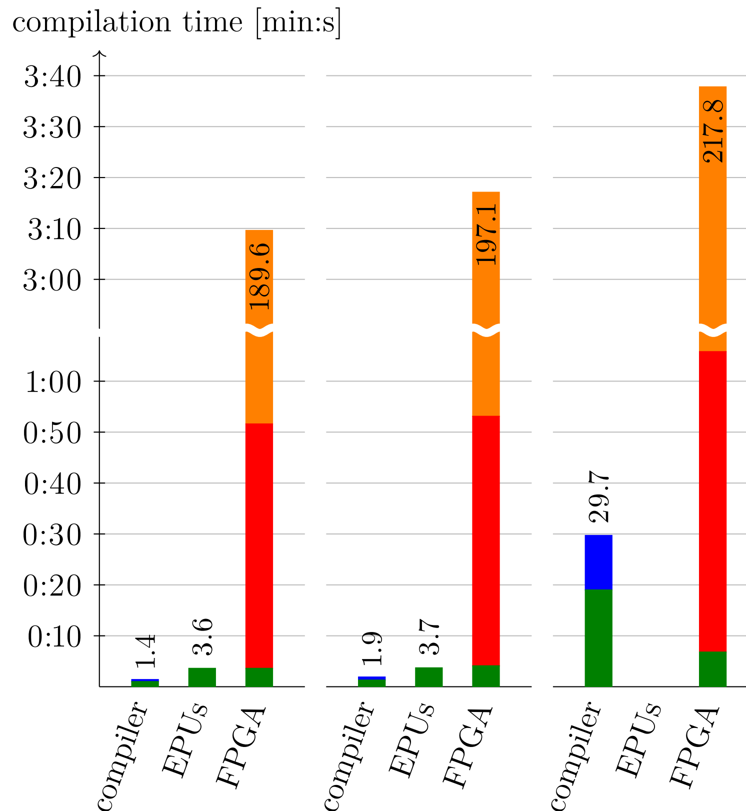


Figure 8.18.: Compilation time of LONG on different backends with different specification depths: 2 on the left, 20 in the center and 200 on the right. The compilation time results from the following involved parties: ■ TeSSLa compiler, ■ Rust compiler, ■ hardware synthesis, and ■ hardware implementation. See Table A.27 in Appendix A.3 for the data table.

EPUs The EPU compiler generates a configuration file which is then used as input to a configuration software provided by Accemic that configures the EPU hardware via USB. It further provides input and output mappings used to generate the input and interpret the output of the EPUs.

FPGA Synthesis The FPGA synthesis integrates the compiler frontend with the Chisel compiler and generates a Verilog module. This module can then be imported into a Verilog project handling the I/O, i. e. providing input events and processing output events. This project is synthesized and uploaded on the FPGA with Xilinx Vivado.

Figure 8.18 shows exemplary compilation times for the different workflows for three scenarios: A small specification, a large specification, and a huge specification. In

the case of the EPUs, the huge specification is missing because it does not fit on the EPUs. The measured compilation includes all compilation steps but does not include uploading the compiled specification onto the target hardware. In the case of the FPGA synthesis, the compilation does not include the out-of-context synthesis for the input FIFO (110 s), the Xillybus logic (71 s), and the output FIFO (98 s) because those can be reused when the specification is changed. The compilation times were measured on a MacBook Pro (15-inch, 2018) with 2.6 GHz Intel Core i7 and 32 GB 2400 MHz DDR4 RAM using Java 1.9 and Rust 1.44. The FPGA synthesis was executed in Xilinx Vivado 2018.2 on a virtual machine on VMware Fusion 12 with four processor cores and 16 GB RAM.

One can clearly see that the FPGA synthesis takes significantly longer than the compilations for the other backends. However, the central problem of the FPGA synthesis is the lack of integration of the compiler with the Xilinx synthesis: The error handling during the compilation can become very tedious: Complicated errors issued by Xilinx Vivado must be manually traced back to the TeSSLa specification. In order to fix these problems, one either has to adjust the TeSSLa specification in non-intuitive ways or adjust the parameters of the heuristic used for the queue placement. Such adjustments require detailed knowledge of the inner workings of the compile process and the synthesis, which the average TeSSLa user does not have.

8.5. Conclusion

The evaluation has shown that the compiled software backend efficiently works for the considered real-work specifications, as long as they are executed individually. Comparing the absolute values for the throughput on the different backends is biased because the throughput depends on the used hardware. However, we can see a clear trend: The FPGA synthesis works well on the examined real-world specifications, too. The EPUs are less efficient on these individual specifications.

However, the comparison of the real-world specifications considers the execution of individual specifications that are relatively small. The synthetic specifications provide insights into the general behaviour of the different backends for more complex specifications. The performance of the software backend is independent of the recursion depth but drops significantly for an increased specification depth or number of inputs. The EPUs and FPGA synthesis can utilise the inherent parallelisation of the hardware: Their performance is independent of the specification depth, and multiple independent inputs do not influence each other. The EPUs' throughout can increase up to 100 MEvents/s, and the FPGA's throughput is only limited by its I/O.

Recursive specifications do not work well on hardware, especially not on EPUs. The FPGA synthesis is affected by this issue, too, but handles it much better due to its asynchronous processing. From the evaluated selection of real-world specifications, we can deduce that many practical specifications do not need complex recursions. All but one specification requires either no recursion or simple recursions that can be expressed with foldLift. Only the INCDEC contains mutual recursive definitions, which are very inefficient on the EPUs.

Regarding the resource utilisations, the EPUs can only handle specifications up to a certain specification depth. The concrete limit depends on the used hardware, but this limit can be reached as the experiment with adjustable specification depth shows. The size of the specification limits the FPGA synthesis, but the synthetic specifications show that this does not pose a serious limitation in practice. The same reasoning applies even more strongly to the software backend: There is a general limit of the specification size, but this is not reached.

We have seen that the software compiler is simple to use and does not have any additional requirements. The compilation for the EPUs is similarly simple, but highly specialised hardware is required. The compilation for the FPGAs is a complicated manual process, but less specialised hardware is required.

All in all, the software backend is the easiest to use and works reasonably fast in many cases without any form of parallelisation. The EPUs perform well on simple specifications and can provide fast processing up to 100 MHz in case of multiple inputs/parallel processing. The FPGA synthesis is not suited for rapid prototyping but seems promising for synthesising a carefully crafted specification onto hardware for long-running tests.

The best approach highly depends on the event sources and the workflow: For interactive debugging sessions with continuously adjusted specifications, the EPUs are the best solution if input streams with high event rates are available on the hardware. If the input streams are available in a software setting, the compiled software backend is sufficient in many cases, especially if only individual specifications are evaluated. The software backends and the EPUs are well suited for long term evaluation without interactive adjustments, too. However, the FPGAs can be faster for large specifications in this setting. The manual compilation process of the FPGA synthesis is not suited for interactive debugging, but the extra effort may be justified for this setting.

9 Conclusion and Future Work

This chapter concludes the thesis with a summary of the main contents and results. The second section presents an overview of open issues and possible extensions.

9.1. Conclusion

This thesis defined monitor constructions for TeSSLa for four different backends: Interpreter, software compiler, EPUs and FPGA synthesis. The introduction motivated two main research questions. The first question Q1 deals with the formal relation between the TeSSLa semantics on streams and the different implementations. Their relation to the TeSSLa semantics and correctness was shown using different abstractions of the monitoring semantics. An empirical evaluation of their performance tackled the second research question Q2 regarding the efficiency of the implementations. It has shown that while the software compiler is fast enough for most specifications, the FPGA synthesis can utilise hardware parallelism to provide constant throughput for long specifications. The EPUs are most efficient for mostly independent inputs with high event frequency.

The TeSSLa semantics on streams is based on straightforward definitions of the basic operators **last**, **lift** and **delay**. Its simplicity comes from the absence of any considerations regarding progress or incomplete streams. The extension to the monitoring semantics on monitoring streams naturally adds these considerations using sets of streams representing all possible refinements of incomplete streams. Existing results on TeSSLa were adopted to this novel monitoring semantics: It was proven that the fixed point is unique, i. e. for every combination of incomplete input streams, the output streams are uniquely defined. Further, the expressiveness of TeSSLa could be characterised as follows: For every function on monitoring streams, which is Scott-continuous, preserves full knowledge, and is future independent, there is a behavioural equivalent TeSSLa specification.

This thesis deals with online monitoring, i. e. input events are processed when they occur, and output events are generated based on the input events that occurred so far. The input traces are obtained by observing a system with discrete events at

arbitrary timestamps. The TeSSLa operators are designed such that it is sufficient to store a finite number of events in the monitor.

The interpreter and the software compiler are based on the same formalism. The synchronised monitoring function is an abstraction of the monitoring semantics. It uses a global progress across all streams. The input streams are assumed to be already synchronised, i. e. the synchronisation of the input streams happens before the actual monitoring. The synchronised streams do not require a fixed step width. The synchronised monitoring function can create additional events at arbitrary timestamps, leading to TeSSLa's ability to process and create Zeno streams. The synchronous implementation computes the next timestamp in every step. The next timestamp is either internally generated by a $\textbf{delay}^{\mathsf{s}}$ operator or the next timestamp from the input stream. The interpreter and the compiler are the most general implementation in the sense that they support any TeSSLa specification with arbitrary immutable data structures.

The interpreter and the compiled monitor execute a specification on a CPU in a single thread. Comparing the four different backends showed that the compiled software monitors are the most efficient solution for complex specifications with arbitrary recursions. The size of the recursion does not influence the throughput. The interpreter is by orders of magnitude slower than the compiled monitor because it represents the specification's flow graph as an object graph in the memory. The dynamic evaluation of this object graph limits the effect of most compile-time and JIT optimisations of the Scala compiler and the JVM.

The EPUs are based on the same synchronous semantics. Instead of a single synchronous execution, the flow graph is mapped onto a pipeline of processing units inspired by data flow processors. The formal EPU model is used to show the correctness of the translation of a TeSSLa specification into a configuration for EPUs. The EPUs support primitive data types, i. e. Boolean and integer values.

This thesis only considers timestamp-conservative specifications for the translation toward EPUs. The EPUs are not designed for the synchronisation required to insert additional timestamps into the synchronous event stream. The EPUs still realise the principle of arbitrary timestamps in the form of sparse coding, i. e. there is no explicit message encoding the absence of events for specific timestamps. This approach makes representing and processing timed event streams efficient, but it is also one of the main reasons why recursive specifications require additional explicit progress messages and synchronisation on the EPUs.

Although the EPUs are slower than the compiled software monitors on single specifications, the evaluation has shown that the EPUs perform very well for their intended use case: They are made to monitor simple specifications on high-frequency streams and mostly independent inputs with a few dependencies. In this scenario, the EPUs

can utilise the parallelisation of the hardware with their pipeline. The reconfiguration can be automated and is fast enough such that interactive debugging sessions with continuously adjusted specifications work similarly well as with the compiled software monitors.

It was shown that the EPUs can evaluate all timestamp-conservative specifications with primitive data types. However, more complex and especially recursive specifications are not handled efficiently by the EPUs. The translation of the blocking last performs poorly due to its complexity. It is possible to rewrite simple recursions with the foldLift operator that allows more efficient translations: Recursively defined streams are translated into sequentially executed commands on a single EPU. Even if this runs counter to the pipelining principle of the EPUs, it is more efficient for simple recursions. The execution of such a sequence of commands on the EPUs still follows the idea of data flow processors. Instead of a program stored sequentially in memory processed by a CPU that increments a program counter, each command has a pointer referring to its successor that is processed as the next command by the EPU.

The FPGA synthesis can utilise the parallelism of the hardware even further using an asynchronous evaluation approach. The flow graph is directly mapped onto the hardware, i. e. the operators are translated into their corresponding channel operators connected through channels corresponding to the edges of the flow graph. The abstract monitoring semantics is an abstraction of the monitoring semantics that preserves the individual progress per stream. The translation into the formally defined operator networks implements this asynchrony by explicitly encoding the individual progress of every stream. Instead of a global synchronisation, the input streams of every operator are synchronised locally per operator.

The synthesis supports primitive data types, i. e. Boolean and integer values of arbitrary but fixed size, as well as option data types with an additional flag indicating the extra value $\perp$ and tuples of fixed size. The asynchrony allows efficient evaluation of complex specifications on the hardware. Even in the case of recursive specifications, only the recursive branch of the flow graph is slowed down.

The synthesis, however, comes with several engineering problems: The heuristics of the queue placement are based on empirical parameters. The compiler generates a Verilog module which is then processed further by Xilinx Vivado. If any errors occur in this final compilation stage, it is complicated for the user to relate the issues with specific parts of the specification.

The empirical findings of this thesis can be summarised as follows: First, the evaluation with the real-world examples shows that all three realisations are feasible for actual usage. The real-world specifications have shown that all three solutions are sufficient in many cases.

The evaluation did not show that one of the three implementations is fundamentally superior to the others. Instead, the best solution depends on the trace source in terms of structure, number and frequency of events, as well as on the depth, size and recursion depth of the specification.

The evaluation carried out cannot provide precise limits as to when a specification can be evaluated most efficiently with which approach, as this depends heavily on the hardware platforms available. Especially, the FPGA hardware used for the EPUs and the synthesis is not directly comparable. Nevertheless, a clear tendency can be seen from the experiments, showing that the three backends are especially promising in different application areas and – depending on the concrete setting – every solution can outperform the others in some cases: The software is most flexible for complex specifications with long recursions aggregating information along the trace; the FPGA synthesis can compile large specifications, and the throughput stays independent of the specification size, i. e. the specification's depth and its number of inputs; and the EPUs can provide high throughput up to 100 MEvents/s for the setting that they are optimised for: Evaluating many specifications in parallel that are mostly independent.

The software monitor offers flexibility and efficiency. It allows interactive debugging and rapid adjustment of specifications. The software backend is the most efficient for complex specifications with long cycles in the dependency graph. However, the experiments with the real-world specifications have shown that the software monitors are sufficient for executing individual specifications and can even outperform the hardware monitors in some situations.

The EPUs can be almost equally flexible and even more efficient for their special use case of mostly independent inputs with a few dependencies. The synthetic specifications were able to show that the EPUs can reach their theoretical maximum of 100 MEvents/s in that setting. For interactive debugging sessions with continuously adjusted specifications, the EPUs are the best solution if input streams with high event rates are available on the hardware.

Finally, the FPGA synthesis is less flexible because adjusting the specification requires an entirely new synthesis, but it can be the most efficient solution for large specifications as its performance is independent of the size of the specification. The throughput of the hardware synthesis is independent of the specification depth and can thus outperform the software for larger specifications. Further, the measured throughputs are often close to the theoretical maximum of 50 MEvents/s on the available hardware. Both the software monitors as well as the EPUs are suited for long-term evaluations without interactive adjustments, but the FPGAs can be more efficient in this setting for large specifications. For example, in the case of the artificial specification LONG they outperformed the EPUs and the software monitor for specifications with a depth of about ten counting blocks chained after another.

Manual compilation of the FPGA synthesis is not suitable for interactive debugging, but for long-term observations, the extra effort may be justified.

9.2. Outlook and Future Work

The final section of the thesis discusses several possible improvements and extensions based on the foundations of this work.

As already mentioned, more engineering is needed for the FPGA synthesis. Using the block RAM of the FPGA for the queues instead of its registers is a promising improvement. This adjustment would drastically decrease the resource utilisation of the queues and therefore allow the synthesis of even larger specifications. A more sophisticated extension is the improvement of error handling during synthesis. Can errors, especially negative slack reported by the Xilinx Vivado synthesis, be mapped back to the TeSSLa specification? Ideally, a negative slack would lead to an adjustment of the parameters used in the heuristic for the queue placement. Additional queues might be able to increase the slack.

The evaluation has clearly shown that the EPUs cannot efficiently process complex recursive specifications. However, the foldLift can express typically occurring simple recursions much better. This raises the question of how much can be gained from simplifying the EPUs if one removes their ability to express blocking **last**. For example, the additional inputs and outputs, the configurable switching network, the blocking counter and the ATSC command handling could be removed.

In the case of the software compilation, this thesis focuses on translating the TeSSLa specification into a generic programming language using only features of structured programming, in particular, no jumps. However, many optimisations are based on automata and jump tables to avoid executions of unused code. By abandoning the idea of a generic translation, one could use specific features of the target language.

Further, the software compilation discussed in this thesis is entirely sequential. In the setting of online monitoring, one could try to utilise the compositionality of TeSSLa specifications to introduce parallelisation into the software compilation: A TeSSLa specification can be split into several specifications such that one monitor's output is used as the following monitor's input. The independent monitors can be scheduled on different CPU cores or even different processors. For offline monitoring and trace analysis, one could also consider splitting the trace and independently processing the parts. The results can then be joined together in a later processing step, following the established map-reduce pattern [DG10].

A natural approach to overcoming the limitations of the individual backends is their combination. Combining the software and hardware backends is a promising approach. The software is most efficient for complex specifications with complex data structures, and the hardware is more efficient on specifications without complex recursions and only works with simple data types. A manual combination is mainly an engineering task. An automatic approach to splitting the specification and determining which part to assign to which backend would require some estimation of the event frequency of streams.

The monitoring semantics introduced in this thesis is inspired by the approach to handling streams with partial information in [LSS+19]. The monitoring semantics can handle the absence of information in the middle of the stream. The abstract monitoring semantics is no longer capable of that. However, the formalisations introduced in this thesis are very well suited for an extension towards partial information. This leads to the next question of how partial information can be efficiently represented on the hardware backends.

The software backend can handle complex data structures as long as they are immutable. The efficient implementation of immutable data structures leads to the aggregate update problem [HB85] addressed for TeSSLa software monitors in [KLS+22]. Some complex data structures can be implemented naturally on hardware, too. For example, finite maps can be implemented in the memory of an FPGA. Compared to the software backend, however, treating overflows of the finite data structures on the hardware is of greater importance. The overflow of data structures like maps and sets could be handled using similar approaches as for partial information. An exemplary specification with a queue that can only hold a finite number of entries and thus has to remove older entries with every enqueuing was already discussed in [LSS+19]. This approach could be extended to hardware synthesis.

A Evaluation Appendix

The appendix contains additional details on the performed experiments for the evaluation.

A.1. Specifications

This section defines the specifications used in the evaluation in Section 8.2.1.

eventchain

Input Streams: $q0w, q0r, q1w, q1r, q2w, q2r \in \mathcal{S}_{\mathbb{U}}$

Specification:

$$\begin{aligned}
startTime &= \mathbf{time}(q0w) \\
a &= \mathbf{last}(startTime, q0r) \\
b &= \mathbf{last}(a, q1w) \\
c &= \mathbf{last}(b, q1r) \\
d &= \mathbf{last}(c, q2w) \\
e &= \mathbf{last}(d, q2r) \\
firstResponse &= \mathbf{lift}(new)(e, \mathsf{prev}(e)) \\
responseTime &= \mathbf{lift}(sub)(\mathbf{time}(firstResponse), firstResponse)
\end{aligned}$$

Lifted functions:

$$new(a,b) = \begin{cases} a & \text{if } a \neq b, \\ \bot & \text{otherwise.} \end{cases}$$

$$sub(a,b) = a - b$$

Output Stream: $responseTime \in \mathcal{S}_{\mathbb{T}}$

The dependency graph is shown in Figure 8.4 in Section 8.2.1.

runtime

Input Streams: $call, ret \in \mathcal{S}_{\mathbb{U}}$

Specification:

$$r = \mathsf{on}(ret, \mathbf{time}(ret) - \mathbf{time}(call))$$

Output Stream: $r \in \mathcal{S}_{\mathbb{T}}$

The dependency graph is shown in Figure 8.5 in Section 8.2.1.

toggle

Input Streams: $x \in \mathcal{S}_{\mathbb{U}}$

Specification:

$$second = \mathsf{default}(\mathbf{lift}(not)(\mathbf{last}(second, x)), true)$$

Lifted function:

$$not(x) = \neg x$$

Output Stream: $second \in \mathcal{S}_{\mathbb{B}}$

The dependency graph is shown in Figure 8.5 in Section 8.2.1.

rosace

Input Streams: $read, write \in \mathcal{S}_{\mathbb{U}}$

Specification:

$$\begin{aligned} trigger &= \mathsf{merge}(read, write) \\ second &= \mathbf{default}(\mathbf{lift}(not)(\mathbf{last}(second, trigger)), true) \\ bad &= second \wedge \mathbf{time}(read) - \mathbf{time}(write) < 10 \\ error &= \mathbf{lift}(filterTrue)(bad) \end{aligned}$$

Lifted functions:

$$filterTrue(a) = \begin{cases} \square & \text{if } a = true, \\ \bot & \text{otherwise.} \end{cases}$$

$$not(x) = \neg x$$

Output Stream: $error \in \mathcal{S}_{\mathbb{U}}$

The dependency graph is shown in Figure 8.6 in Section 8.2.1.

incdec

Input Streams: $setInc, getInc, setDec, getDec \in \mathcal{S}_{\mathbb{U}}$

Specification:

$$\begin{aligned} value &= \mathsf{merge}(\mathbf{last}(incValue, setInc), \mathbf{last}(decValue, setDec), 0) \\ incValue &= \mathbf{lift}(inc)(\mathbf{last}(value, getInc)) \\ decValue &= \mathbf{lift}(dec)(\mathbf{last}(value, getDec)) \end{aligned}$$

Lifted functions

$$\begin{aligned} inc(i) &= i + 1 \\ dec(i) &= i - 1 \end{aligned}$$

Output-Stream: $value \in \mathcal{S}_{\mathbb{N}}$

The dependency graph is shown in Figure 8.6 in Section 8.2.1.

resetCount

Input Streams: $e, reset \in \mathcal{S}_{\mathbb{U}}$

Specification:

$$count = \mathsf{default}(\mathbf{lift}(rc)(e, reset, \mathbf{last}(count, e)), 0)$$

Lifted function:

$$rc(e, r, \ell) = \begin{cases} 0 & \text{if } r \neq \bot \wedge e = \bot, \\ 1 & \text{if } r \neq \bot \wedge e \neq \bot, \\ \ell + 1 & \text{if } r = \bot \wedge e \neq \bot, \\ \bot & \text{otherwise.} \end{cases}$$

Output Stream: $count \in \mathcal{S}_{\mathbb{N}}$

The dependency graph is shown in Figure 8.5 in Section 8.2.1.

burst

Input Stream: $e\colon \mathcal{S}_{\mathbb{U}}$

Specification:

$$\begin{aligned}
timeE &= \textbf{time}(e)\\
starts &= \textbf{lift}(detectStart)(timeE, \textbf{last}(starts, timeE))\\
c &= \mathsf{default}(\textbf{lift}(rc)(e, starts, \textbf{last}(c, e))\\
ok &= \textbf{lift}(valid)(c, \mathsf{default}(\textbf{time}(e) < starts + 5000, true))
\end{aligned}$$

Lifted functions:

$$\begin{aligned}
detectStart(t, \ell) &= \begin{cases} t & \text{if } \ell = \bot \vee t - \ell \geq 7000,\\ \bot & \text{otherwise.}\end{cases}\\
rc(e, r, \ell) &= \begin{cases} 0 & \text{if } r \neq \bot \wedge e = \bot,\\ 1 & \text{if } r \neq \bot \wedge e \neq \bot,\\ \ell + 1 & \text{if } r = \bot \wedge e \neq \bot,\\ \bot & \text{otherwise.}\end{cases}\\
valid(count, inTime) &= count \leq 5 \wedge inTime
\end{aligned}$$

Output Stream: $ok \in \mathcal{S}_{\mathbb{B}}$

The dependency graph is shown in Figure 8.7 in Section 8.2.1.

A.2. Generators

In the following pseudocode the notation $a..b$ represents a range from a to b including both a and b and $a...b$ represents a range from a to b including a but excluding b. For every occurrence of such a range instead of a concrete value we assume that a value from that range is drawn with an even distribution using the PRNG described in Section 8.1.1. Statements with a given chance are either executed or not depending on the output of the PRNG.

Input generator for burst

```
advance 23...29.9 ms
loop
  4..6 times
    unit event e
    advance 1...1.3 ms
  advance 1...5.2 ms
```

Input generator for eventchain

```
advance 23...25.3 ms
loop
  unit event q2r
  advance 1...1.1 ms
  unit event q0w with 1/10 chance
  advance 30...33.0 ms

  unit event q1r
  advance 1...1.1 ms
  unit event q2w
  advance 30...33.0 ms

  unit event q0r
  advance 1...1.1 ms
  unit event q1w
  advance 30...33.0 ms
```

Input generator for incdec

```
advance 23..23000 ns
loop
  unit event getInc
  advance 1..1000 ns
  with 4.2% chance
    unit event getDec
    advance 1..1000 ns
    unit event setInc
    advance 1..1000 ns
  otherwise
```

```
    unit event setInc
    advance 1..1000 ns
    unit event getDec
    advance 1..1000 ns
  unit event setDec
  advance 1..1000 ns
```

Input generator for resetCount

```
advance 23...29.9 ms
loop
  with 1/30 chance
    unit event reset
    unit event e with 1/5 chance
  otherwise
    unit event e
  advance 1...1.3 ms
```

Input generator for rosace

```
advance 23..23000 ns
loop
  with 4.2% chance
    unit event read
    advance 1..1000 ns
    unit event write
    advance 1..1000 ns
  otherwise
    unit event write
    advance 1..1000 ns
    unit event read
    advance 1..1000 ns
```

Input generator for runtime

```
advance 23...69 ms
loop
  unit event call
  advance 1...3 ms
```

unit event ret
advance 1...3 **ms**

Input generator for toggle

advance 23...69 **ms**
loop
  unit event x
  advance 1...3 **ms**

Input generator for long

advance 1 **ns**
loop
  event x **with value** $-17..23$
  advance 1 **ns**

Input generator for recursion

loop
  advance 100...200 **ns**
  unit event x

Input generator for inputs

loop
  $\forall i \in \{a, \ldots, j\}$ *// depending on size*
    advance 100...1100 **ns**
    event i **with value** 0...1000

Input generator for inputs with same timestamps

loop
  advance 100...1100 **ns**
  $\forall i \in \{a, \ldots, j\}$ *// depending on size*
    event i **with value** 0...1000

A.3. Measurement Data

number of runs	trace length	runtime [ns]	throughput [MEvents/s]
$1 \cdot 10^4$	$1 \cdot 10^2$	204,345	0.49
$1 \cdot 10^4$	$1 \cdot 10^3$	1,990,076	0.50
$1 \cdot 10^4$	$1 \cdot 10^4$	20,427,270	0.49
$1 \cdot 10^3$	$1 \cdot 10^5$	214,301,660	0.47
$1 \cdot 10^2$	$1 \cdot 10^6$	2,241,338,228	0.45
$1 \cdot 10^1$	$1 \cdot 10^7$	23,103,290,353	0.43
$1 \cdot 10^1$	$1 \cdot 10^8$	212,104,736,183	0.47

Table A.1.: Runtime of BURST in the interpreter with different trace length. See Figure 8.1 in Section 8.1 for the plot.

spec	backend	number of runs	trace length	runtime [ns]	throughput [MEvents/s]
eventchain	EPU	$1 \cdot 10^0$	$2 \cdot 10^3$	92,180	21.70
eventchain	FPGA	$1 \cdot 10^0$	$2 \cdot 10^3$	40,060	49.93
eventchain	compiler	$3 \cdot 10^1$	$1 \cdot 10^8$	2,384,022,033	41.95
eventchain	interpreter	$1 \cdot 10^1$	$1 \cdot 10^7$	7,077,297,560	1.41
runtime	EPU	$1 \cdot 10^0$	$2 \cdot 10^3$	260,310	7.68
runtime	FPGA	$1 \cdot 10^0$	$2 \cdot 10^3$	40,070	49.91
runtime	compiler	$3 \cdot 10^1$	$1 \cdot 10^8$	974,589,742	102.61
runtime	interpreter	$1 \cdot 10^1$	$1 \cdot 10^7$	8,337,885,724	1.20
toggle	EPU	$1 \cdot 10^0$	$2 \cdot 10^3$	139,940	14.29
toggle	FPGA	$1 \cdot 10^0$	$2 \cdot 10^3$	40,040	49.95
toggle	compiler	$3 \cdot 10^1$	$1 \cdot 10^8$	255,661,581	391.14
toggle	interpreter	$1 \cdot 10^1$	$1 \cdot 10^7$	904,942,849	11.05
rosace	EPU	$1 \cdot 10^0$	$2 \cdot 10^3$	640,780	3.12
rosace	FPGA	$1 \cdot 10^0$	$2 \cdot 10^3$	40,070	49.91
rosace	compiler	$3 \cdot 10^1$	$1 \cdot 10^8$	1,084,636,969	92.20
rosace	interpreter	$1 \cdot 10^1$	$1 \cdot 10^7$	12,395,441,747	0.81
incdec	EPU	$1 \cdot 10^0$	$2 \cdot 10^3$	1,900,600	1.05
incdec	FPGA	$1 \cdot 10^0$	$2 \cdot 10^3$	49,410	40.48
incdec	compiler	$3 \cdot 10^1$	$1 \cdot 10^7$	136,144,602	73.45
incdec	interpreter	$1 \cdot 10^1$	$1 \cdot 10^7$	3,076,022,924	3.25
resetcount	EPU	$1 \cdot 10^0$	$2 \cdot 10^3$	260,480	7.68
resetcount	FPGA	$1 \cdot 10^0$	$2 \cdot 10^3$	39,960	50.05
resetcount	compiler	$3 \cdot 10^1$	$1 \cdot 10^7$	89,755,478	111.41
resetcount	interpreter	$1 \cdot 10^1$	$1 \cdot 10^7$	1,360,060,540	7.35
burst	EPU	$1 \cdot 10^0$	$2 \cdot 10^3$	641,520	3.12
burst	FPGA	$1 \cdot 10^0$	$2 \cdot 10^3$	60,080	33.29
burst	compiler	$3 \cdot 10^1$	$1 \cdot 10^7$	222,010,441	45.04
burst	interpreter	$1 \cdot 10^1$	$1 \cdot 10^7$	19,112,866,384	0.52

Table A.2.: Throughput of different specifications on different backends. In case of the EPU backend the foldLift optimisation is applied where possible. See Figure 8.9 in Section 8.2.3 for the plot.

spec	number of runs	trace length	runtime [ns]	throughput [MEvents/s]
toggle	$1 \cdot 10^0$	$2 \cdot 10^3$	1,340,630	1.49
rosace	$1 \cdot 10^0$	$2 \cdot 10^3$	1,361,040	1.47
resetcount	$1 \cdot 10^0$	$2 \cdot 10^3$	2,701,510	0.74
burst	$1 \cdot 10^0$	$2 \cdot 10^3$	3,375,030	0.59

Table A.3.: Throughput of different specifications on the EPU backend without the foldLift optimisation. See Figure 8.8 in Section 8.2.2 for the plot.

spec	number of runs	trace length	runtime [ns]	throughput [MEvents/s]
burst	$1 \cdot 10^0$	$2 \cdot 10^3$	3,040,550	0.66

Table A.4.: Throughput of BURST on the EPU backend with the foldLift optimisation used only for the counting. See Figure 8.8 in Section 8.2.2 for the plot.

size	number of runs	trace length	runtime [ns]	throughput [MEvents/s]
1	$1 \cdot 10^0$	$2 \cdot 10^4$		14.31
15	$1 \cdot 10^0$	$2 \cdot 10^4$		14.31

Table A.5.: Runtime of LONG with EPUs. See Figure 8.11 in Section 8.3.1 for the plot.

size	number of runs	trace length	runtime [ns]	throughput [MEvents/s]
1	$1 \cdot 10^0$	$1 \cdot 10^7$	200,000,020	50.00
10	$1 \cdot 10^0$	$1 \cdot 10^7$	200,000,030	50.00
100	$1 \cdot 10^0$	$1 \cdot 10^7$	200,000,140	50.00

Table A.6.: Runtime of LONG with FPGA. See Figure 8.11 in Section 8.3.1 for the plot.

size	number of runs	trace length	runtime [ns]	throughput [MEvents/s]
1	$5 \cdot 10^1$	$1 \cdot 10^6$	90,886,323	11.00
2	$5 \cdot 10^1$	$1 \cdot 10^6$	196,931,039	5.08
3	$5 \cdot 10^1$	$1 \cdot 10^6$	257,720,207	3.88
4	$5 \cdot 10^1$	$1 \cdot 10^6$	343,080,960	2.91
5	$5 \cdot 10^1$	$1 \cdot 10^6$	420,098,789	2.38
6	$2 \cdot 10^1$	$1 \cdot 10^6$	534,839,615	1.87
7	$2 \cdot 10^1$	$1 \cdot 10^6$	596,911,548	1.68
8	$2 \cdot 10^1$	$1 \cdot 10^6$	653,112,019	1.53
9	$2 \cdot 10^1$	$1 \cdot 10^6$	733,448,338	1.36
10	$2 \cdot 10^1$	$1 \cdot 10^6$	799,609,658	1.25
11	$1 \cdot 10^1$	$1 \cdot 10^6$	865,397,424	1.16
12	$1 \cdot 10^1$	$1 \cdot 10^6$	952,970,782	1.05
13	$1 \cdot 10^1$	$1 \cdot 10^6$	1,048,707,461	0.95
14	$1 \cdot 10^1$	$1 \cdot 10^6$	1,111,509,669	0.90
15	$1 \cdot 10^1$	$1 \cdot 10^6$	1,212,550,195	0.82
16	$1 \cdot 10^1$	$1 \cdot 10^6$	1,294,163,384	0.77
17	$1 \cdot 10^1$	$1 \cdot 10^6$	1,356,669,363	0.74
18	$1 \cdot 10^1$	$1 \cdot 10^6$	1,470,340,867	0.68
19	$1 \cdot 10^1$	$1 \cdot 10^6$	1,492,344,314	0.67
20	$1 \cdot 10^1$	$1 \cdot 10^6$	1,528,674,981	0.65
21	$1 \cdot 10^1$	$1 \cdot 10^6$	1,661,308,387	0.60
22	$1 \cdot 10^1$	$1 \cdot 10^6$	1,687,632,062	0.59
23	$1 \cdot 10^1$	$1 \cdot 10^6$	1,710,547,569	0.58
24	$1 \cdot 10^1$	$1 \cdot 10^6$	1,815,989,087	0.55
25	$1 \cdot 10^1$	$1 \cdot 10^6$	1,859,339,047	0.54
26	$1 \cdot 10^1$	$1 \cdot 10^6$	1,953,162,596	0.51
27	$1 \cdot 10^1$	$1 \cdot 10^6$	2,027,288,836	0.49
28	$1 \cdot 10^1$	$1 \cdot 10^6$	2,107,824,638	0.47
29	$1 \cdot 10^1$	$1 \cdot 10^6$	2,141,703,793	0.47
30	$1 \cdot 10^1$	$1 \cdot 10^6$	2,222,212,378	0.45

Table A.7.: Runtime of LONG in the interpreter. See Figure 8.11 in Section 8.3.1 for the plot.

size	number of runs	trace length	runtime [ns]	throughput [MEvents/s]
1	$1 \cdot 10^2$	$1 \cdot 10^7$	42,455,063	235.54
2	$1 \cdot 10^2$	$1 \cdot 10^7$	69,896,287	143.07
3	$1 \cdot 10^2$	$1 \cdot 10^7$	95,230,580	105.01
4	$1 \cdot 10^2$	$1 \cdot 10^7$	121,009,501	82.64
5	$1 \cdot 10^2$	$1 \cdot 10^7$	148,092,561	67.53
6	$1 \cdot 10^2$	$1 \cdot 10^7$	191,250,729	52.29
7	$1 \cdot 10^2$	$1 \cdot 10^7$	207,005,864	48.31
8	$1 \cdot 10^2$	$1 \cdot 10^7$	243,450,276	41.08
9	$1 \cdot 10^2$	$1 \cdot 10^7$	259,950,389	38.47
10	$1 \cdot 10^2$	$1 \cdot 10^7$	291,571,671	34.30
11	$1 \cdot 10^2$	$1 \cdot 10^7$	315,873,267	31.66
12	$1 \cdot 10^2$	$1 \cdot 10^7$	357,339,146	27.98
13	$1 \cdot 10^2$	$1 \cdot 10^7$	362,939,398	27.55
14	$1 \cdot 10^2$	$1 \cdot 10^7$	412,174,181	24.26
15	$1 \cdot 10^2$	$1 \cdot 10^7$	446,523,945	22.40
16	$1 \cdot 10^2$	$1 \cdot 10^7$	474,324,084	21.08
17	$1 \cdot 10^2$	$1 \cdot 10^7$	499,858,835	20.01
18	$1 \cdot 10^2$	$1 \cdot 10^7$	520,682,614	19.21
19	$1 \cdot 10^2$	$1 \cdot 10^7$	553,483,858	18.07
20	$1 \cdot 10^2$	$1 \cdot 10^7$	587,497,267	17.02
21	$1 \cdot 10^2$	$1 \cdot 10^7$	615,795,883	16.24
22	$1 \cdot 10^2$	$1 \cdot 10^7$	629,904,953	15.88
23	$1 \cdot 10^2$	$1 \cdot 10^7$	664,520,439	15.05
24	$1 \cdot 10^2$	$1 \cdot 10^7$	692,079,049	14.45
25	$1 \cdot 10^2$	$1 \cdot 10^7$	724,266,821	13.81
26	$1 \cdot 10^2$	$1 \cdot 10^7$	752,526,318	13.29
27	$1 \cdot 10^2$	$1 \cdot 10^7$	789,869,221	12.66
28	$1 \cdot 10^2$	$1 \cdot 10^7$	813,756,208	12.29
29	$1 \cdot 10^2$	$1 \cdot 10^7$	840,189,373	11.90
30	$1 \cdot 10^2$	$1 \cdot 10^7$	874,616,947	11.43

Table A.8.: Runtime of LONG with a compiled monitor in Rust reading events from an array. See Figure 8.11 in Section 8.3.1 and Figure 8.3 in Section 8.1.2 for the plots.

size	number of runs	trace length	runtime [ns]	throughput [MEvents/s]
1	$3 \cdot 10^1$	$1 \cdot 10^7$	42,161,575	237.18
2	$3 \cdot 10^1$	$1 \cdot 10^7$	68,638,228	145.69
3	$3 \cdot 10^1$	$1 \cdot 10^7$	104,338,535	95.84
4	$3 \cdot 10^1$	$1 \cdot 10^7$	122,876,031	81.38
5	$2 \cdot 10^1$	$1 \cdot 10^7$	157,195,923	63.61
6	$2 \cdot 10^1$	$1 \cdot 10^7$	182,200,834	54.88
7	$2 \cdot 10^1$	$1 \cdot 10^7$	221,069,518	45.23
8	$2 \cdot 10^1$	$1 \cdot 10^7$	239,754,759	41.71
9	$2 \cdot 10^1$	$1 \cdot 10^7$	274,378,524	36.45
10	$2 \cdot 10^1$	$1 \cdot 10^7$	303,601,965	32.94
11	$2 \cdot 10^1$	$1 \cdot 10^7$	326,702,867	30.61
12	$2 \cdot 10^1$	$1 \cdot 10^7$	363,233,575	27.53
13	$2 \cdot 10^1$	$1 \cdot 10^7$	379,554,820	26.35
14	$2 \cdot 10^1$	$1 \cdot 10^7$	412,145,522	24.26
15	$2 \cdot 10^1$	$1 \cdot 10^7$	442,657,608	22.59
16	$2 \cdot 10^1$	$1 \cdot 10^7$	477,401,822	20.95
17	$2 \cdot 10^1$	$1 \cdot 10^7$	505,337,253	19.79
18	$2 \cdot 10^1$	$1 \cdot 10^7$	533,898,996	18.73
19	$2 \cdot 10^1$	$1 \cdot 10^7$	561,349,963	17.81
20	$1 \cdot 10^1$	$1 \cdot 10^7$	594,868,885	16.81
21	$1 \cdot 10^1$	$1 \cdot 10^7$	621,718,719	16.08
22	$1 \cdot 10^1$	$1 \cdot 10^7$	651,316,829	15.35
23	$1 \cdot 10^1$	$1 \cdot 10^7$	692,291,093	14.44
24	$1 \cdot 10^1$	$1 \cdot 10^7$	708,075,582	14.12
25	$1 \cdot 10^1$	$1 \cdot 10^7$	750,113,810	13.33
26	$1 \cdot 10^1$	$1 \cdot 10^7$	774,332,259	12.91
27	$1 \cdot 10^1$	$1 \cdot 10^7$	812,987,064	12.30
28	$1 \cdot 10^1$	$1 \cdot 10^7$	836,036,458	11.96
29	$1 \cdot 10^1$	$1 \cdot 10^7$	869,971,770	11.49
30	$1 \cdot 10^1$	$1 \cdot 10^7$	902,631,076	11.08

Table A.9.: Runtime of LONG with a compiled monitor in Rust generating the events in the monitor. See Figure 8.3 in Section 8.1.2 for the plot.

size	number of runs	trace length	runtime [ns]	throughput [MEvents/s]
1	$3 \cdot 10^1$	$1 \cdot 10^7$	191,893,012	52.11
2	$3 \cdot 10^1$	$1 \cdot 10^7$	209,288,573	47.78
3	$3 \cdot 10^1$	$1 \cdot 10^7$	240,498,695	41.58
4	$3 \cdot 10^1$	$1 \cdot 10^7$	266,037,323	37.59
5	$2 \cdot 10^1$	$1 \cdot 10^7$	298,568,966	33.49
6	$2 \cdot 10^1$	$1 \cdot 10^7$	336,275,912	29.74
7	$2 \cdot 10^1$	$1 \cdot 10^7$	360,108,437	27.77
8	$2 \cdot 10^1$	$1 \cdot 10^7$	386,973,528	25.84
9	$2 \cdot 10^1$	$1 \cdot 10^7$	427,971,649	23.37
10	$2 \cdot 10^1$	$1 \cdot 10^7$	445,267,423	22.46
11	$2 \cdot 10^1$	$1 \cdot 10^7$	478,131,160	20.91
12	$2 \cdot 10^1$	$1 \cdot 10^7$	500,682,003	19.97
13	$2 \cdot 10^1$	$1 \cdot 10^7$	522,440,147	19.14
14	$2 \cdot 10^1$	$1 \cdot 10^7$	552,051,045	18.11
15	$2 \cdot 10^1$	$1 \cdot 10^7$	585,286,405	17.09
16	$2 \cdot 10^1$	$1 \cdot 10^7$	614,749,028	16.27
17	$2 \cdot 10^1$	$1 \cdot 10^7$	643,426,801	15.54
18	$2 \cdot 10^1$	$1 \cdot 10^7$	658,988,193	15.17
19	$2 \cdot 10^1$	$1 \cdot 10^7$	711,228,446	14.06
20	$1 \cdot 10^1$	$1 \cdot 10^7$	718,046,716	13.93
21	$1 \cdot 10^1$	$1 \cdot 10^7$	761,880,473	13.13
22	$1 \cdot 10^1$	$1 \cdot 10^7$	798,851,023	12.52
23	$1 \cdot 10^1$	$1 \cdot 10^7$	806,356,772	12.40
24	$1 \cdot 10^1$	$1 \cdot 10^7$	834,501,908	11.98
25	$1 \cdot 10^1$	$1 \cdot 10^7$	864,310,910	11.57
26	$1 \cdot 10^1$	$1 \cdot 10^7$	932,197,753	10.73
27	$1 \cdot 10^1$	$1 \cdot 10^7$	930,659,250	10.75
28	$1 \cdot 10^1$	$1 \cdot 10^7$	956,980,042	10.45
29	$1 \cdot 10^1$	$1 \cdot 10^7$	987,008,010	10.13
30	$1 \cdot 10^1$	$1 \cdot 10^7$	1,038,327,175	9.63

Table A.10.: Runtime of LONG with a compiled monitor in Rust reading events from a binary file. See Figure 8.3 in Section 8.1.2 for the plot.

size	number of runs	trace length	runtime [ns]	throughput [MEvents/s]
1	$5 \cdot 10^1$	$1 \cdot 10^7$	115,108,411	86.87
2	$5 \cdot 10^1$	$1 \cdot 10^7$	154,213,936	64.84
3	$5 \cdot 10^1$	$1 \cdot 10^7$	186,693,330	53.56
4	$5 \cdot 10^1$	$1 \cdot 10^7$	209,600,857	47.71
5	$2.5 \cdot 10^1$	$1 \cdot 10^7$	270,277,015	37.00
6	$2.5 \cdot 10^1$	$1 \cdot 10^7$	313,317,769	31.92
7	$2.5 \cdot 10^1$	$1 \cdot 10^7$	349,931,168	28.58
8	$2.5 \cdot 10^1$	$1 \cdot 10^7$	395,040,110	25.31
9	$2.5 \cdot 10^1$	$1 \cdot 10^7$	416,186,503	24.03
10	$2.5 \cdot 10^1$	$1 \cdot 10^7$	502,192,217	19.91
11	$2.5 \cdot 10^1$	$1 \cdot 10^7$	509,126,384	19.64
12	$2.5 \cdot 10^1$	$1 \cdot 10^7$	573,332,915	17.44
13	$2.5 \cdot 10^1$	$1 \cdot 10^7$	648,424,424	15.42
14	$2.5 \cdot 10^1$	$1 \cdot 10^7$	705,324,338	14.18
15	$2.5 \cdot 10^1$	$1 \cdot 10^7$	726,477,490	13.77
16	$2.5 \cdot 10^1$	$1 \cdot 10^7$	807,393,266	12.39
17	$2.5 \cdot 10^1$	$1 \cdot 10^7$	864,756,539	11.56
18	$2.5 \cdot 10^1$	$1 \cdot 10^7$	930,784,136	10.74
19	$2.5 \cdot 10^1$	$1 \cdot 10^7$	951,975,985	10.50
20	$2.5 \cdot 10^1$	$1 \cdot 10^7$	994,735,067	10.05
21	$2.5 \cdot 10^1$	$1 \cdot 10^7$	996,262,936	10.04
22	$2.5 \cdot 10^1$	$1 \cdot 10^7$	1,024,798,746	9.76
23	$5 \cdot 10^0$	$1 \cdot 10^7$	19,511,254,316	0.51
24	$5 \cdot 10^0$	$1 \cdot 10^7$	23,182,482,390	0.43
25	$5 \cdot 10^0$	$1 \cdot 10^7$	24,351,205,401	0.41
26	$5 \cdot 10^0$	$1 \cdot 10^7$	26,720,707,734	0.37
27	$5 \cdot 10^0$	$1 \cdot 10^7$	26,803,498,679	0.37
28	$5 \cdot 10^0$	$1 \cdot 10^7$	28,457,711,393	0.35
29	$5 \cdot 10^0$	$1 \cdot 10^7$	27,950,136,947	0.36
30	$5 \cdot 10^0$	$1 \cdot 10^7$	29,131,451,612	0.34

Table A.11.: Runtime of LONG with a compiled monitor in Java reading events from an ArrayList. See Figure 8.3 in Section 8.1.2 for the plot.

size	number of runs	trace length	runtime [ns]	throughput [MEvents/s]
1	$1 \cdot 10^2$	$1 \cdot 10^7$	56,414,772	177.26
2	$1 \cdot 10^2$	$1 \cdot 10^7$	92,182,443	108.48
3	$1 \cdot 10^2$	$1 \cdot 10^7$	153,947,047	64.96
4	$1 \cdot 10^2$	$1 \cdot 10^7$	172,890,409	57.84
5	$5 \cdot 10^1$	$1 \cdot 10^7$	207,971,713	48.08
6	$5 \cdot 10^1$	$1 \cdot 10^7$	244,282,516	40.94
7	$5 \cdot 10^1$	$1 \cdot 10^7$	287,720,960	34.76
8	$5 \cdot 10^1$	$1 \cdot 10^7$	321,092,695	31.14
9	$5 \cdot 10^1$	$1 \cdot 10^7$	369,685,599	27.05
10	$5 \cdot 10^1$	$1 \cdot 10^7$	403,611,467	24.78
11	$5 \cdot 10^1$	$1 \cdot 10^7$	451,690,090	22.14
12	$5 \cdot 10^1$	$1 \cdot 10^7$	457,071,750	21.88
13	$2.5 \cdot 10^1$	$1 \cdot 10^7$	554,769,174	18.03
14	$2.5 \cdot 10^1$	$1 \cdot 10^7$	587,175,093	17.03
15	$2.5 \cdot 10^1$	$1 \cdot 10^7$	634,930,335	15.75
16	$2.5 \cdot 10^1$	$1 \cdot 10^7$	689,760,685	14.50
17	$2.5 \cdot 10^1$	$1 \cdot 10^7$	780,226,537	12.82
18	$2.5 \cdot 10^1$	$1 \cdot 10^7$	697,012,571	14.35
19	$2.5 \cdot 10^1$	$1 \cdot 10^7$	822,764,168	12.15
20	$2.5 \cdot 10^1$	$1 \cdot 10^7$	862,241,176	11.60
21	$2.5 \cdot 10^1$	$1 \cdot 10^7$	907,608,105	11.02
22	$2.5 \cdot 10^1$	$1 \cdot 10^7$	968,641,897	10.32
23	$5 \cdot 10^0$	$1 \cdot 10^7$	19,697,085,457	0.51
24	$5 \cdot 10^0$	$1 \cdot 10^7$	23,556,932,732	0.42
25	$5 \cdot 10^0$	$1 \cdot 10^7$	24,284,851,519	0.41
26	$5 \cdot 10^0$	$1 \cdot 10^7$	24,565,081,899	0.41
27	$5 \cdot 10^0$	$1 \cdot 10^7$	25,201,218,749	0.40
28	$5 \cdot 10^0$	$1 \cdot 10^7$	26,355,000,608	0.38
29	$5 \cdot 10^0$	$1 \cdot 10^7$	26,302,536,265	0.38
30	$5 \cdot 10^0$	$1 \cdot 10^7$	28,371,086,716	0.35

Table A.12.: Runtime of LONG with a compiled monitor in Java reading events from an array. See Figure 8.3 in Section 8.1.2 for the plot.

size	number of runs	trace length	runtime [ns]	throughput [MEvents/s]
1	$5 \cdot 10^1$	$1 \cdot 10^7$	314,910,361	31.76
2	$5 \cdot 10^1$	$1 \cdot 10^7$	356,673,584	28.04
3	$5 \cdot 10^1$	$1 \cdot 10^7$	368,331,088	27.15
4	$5 \cdot 10^1$	$1 \cdot 10^7$	455,482,677	21.95
5	$5 \cdot 10^1$	$1 \cdot 10^7$	493,139,546	20.28
6	$5 \cdot 10^1$	$1 \cdot 10^7$	524,186,272	19.08
7	$5 \cdot 10^1$	$1 \cdot 10^7$	539,341,968	18.54
8	$5 \cdot 10^1$	$1 \cdot 10^7$	598,544,745	16.71
9	$5 \cdot 10^1$	$1 \cdot 10^7$	611,871,085	16.34
10	$3 \cdot 10^1$	$1 \cdot 10^7$	680,844,087	14.69
11	$3 \cdot 10^1$	$1 \cdot 10^7$	707,735,632	14.13
12	$3 \cdot 10^1$	$1 \cdot 10^7$	707,319,073	14.14
13	$3 \cdot 10^1$	$1 \cdot 10^7$	813,360,494	12.29
14	$3 \cdot 10^1$	$1 \cdot 10^7$	849,765,657	11.77
15	$3 \cdot 10^1$	$1 \cdot 10^7$	883,555,421	11.32
16	$3 \cdot 10^1$	$1 \cdot 10^7$	997,008,603	10.03
17	$3 \cdot 10^1$	$1 \cdot 10^7$	1,008,440,908	9.92
18	$3 \cdot 10^1$	$1 \cdot 10^7$	1,048,516,161	9.54
19	$3 \cdot 10^1$	$1 \cdot 10^7$	1,090,875,171	9.17
20	$2 \cdot 10^1$	$1 \cdot 10^7$	1,162,163,782	8.60
21	$2 \cdot 10^1$	$1 \cdot 10^7$	1,205,447,676	8.30
22	$2 \cdot 10^1$	$1 \cdot 10^7$	1,240,629,188	8.06
23	$2 \cdot 10^1$	$1 \cdot 10^7$	20,581,595,278	0.49
24	$2 \cdot 10^1$	$1 \cdot 10^7$	24,764,218,324	0.40
25	$5 \cdot 10^0$	$1 \cdot 10^7$	25,331,132,132	0.39
26	$5 \cdot 10^0$	$1 \cdot 10^7$	26,943,967,294	0.37
27	$5 \cdot 10^0$	$1 \cdot 10^7$	28,568,458,247	0.35
28	$5 \cdot 10^0$	$1 \cdot 10^7$	28,694,563,788	0.35
29	$5 \cdot 10^0$	$1 \cdot 10^7$	28,611,482,278	0.35
30	$5 \cdot 10^0$	$1 \cdot 10^7$	29,810,854,786	0.34

Table A.13.: Runtime of LONG with a compiled monitor in Java reading events from a binary file. See Figure 8.3 in Section 8.1.2 for the plot.

size	number of runs	trace length	runtime [ns]	throughput [MEvents/s]
1	$5 \cdot 10^1$	$1 \cdot 10^7$	55,031,923	181.71
2	$5 \cdot 10^1$	$1 \cdot 10^7$	96,375,368	103.76
3	$5 \cdot 10^1$	$1 \cdot 10^7$	135,019,584	74.06
4	$5 \cdot 10^1$	$1 \cdot 10^7$	189,825,305	52.68
5	$2.5 \cdot 10^1$	$1 \cdot 10^7$	227,047,434	44.04
6	$2.5 \cdot 10^1$	$1 \cdot 10^7$	266,527,259	37.52
7	$2.5 \cdot 10^1$	$1 \cdot 10^7$	304,807,979	32.81
8	$2.5 \cdot 10^1$	$1 \cdot 10^7$	318,353,101	31.41
9	$2.5 \cdot 10^1$	$1 \cdot 10^7$	360,732,475	27.72
10	$2.5 \cdot 10^1$	$1 \cdot 10^7$	426,692,094	23.44
11	$2.5 \cdot 10^1$	$1 \cdot 10^7$	435,028,712	22.99
12	$2.5 \cdot 10^1$	$1 \cdot 10^7$	476,450,042	20.99
13	$2.5 \cdot 10^1$	$1 \cdot 10^7$	558,846,522	17.89
14	$2.5 \cdot 10^1$	$1 \cdot 10^7$	588,630,770	16.99
15	$2.5 \cdot 10^1$	$1 \cdot 10^7$	650,885,786	15.36
16	$2.5 \cdot 10^1$	$1 \cdot 10^7$	698,459,391	14.32
17	$2.5 \cdot 10^1$	$1 \cdot 10^7$	745,943,303	13.41
18	$2.5 \cdot 10^1$	$1 \cdot 10^7$	785,306,568	12.73
19	$2.5 \cdot 10^1$	$1 \cdot 10^7$	830,273,239	12.04
20	$2.5 \cdot 10^1$	$1 \cdot 10^7$	886,362,434	11.28
21	$2.5 \cdot 10^1$	$1 \cdot 10^7$	956,252,611	10.46
22	$2.5 \cdot 10^1$	$1 \cdot 10^7$	957,569,432	10.44
23	$5 \cdot 10^0$	$1 \cdot 10^7$	19,766,721,742	0.51
24	$5 \cdot 10^0$	$1 \cdot 10^7$	22,855,646,809	0.44
25	$5 \cdot 10^0$	$1 \cdot 10^7$	23,605,091,223	0.42
26	$5 \cdot 10^0$	$1 \cdot 10^7$	24,605,476,669	0.41
27	$5 \cdot 10^0$	$1 \cdot 10^7$	25,723,924,486	0.39
28	$5 \cdot 10^0$	$1 \cdot 10^7$	26,443,267,828	0.38
29	$5 \cdot 10^0$	$1 \cdot 10^7$	27,436,330,765	0.36
30	$5 \cdot 10^0$	$1 \cdot 10^7$	29,920,376,543	0.33

Table A.14.: Runtime of LONG with a compiled monitor in Java generating the events in the monitor. See Figure 8.3 in Section 8.1.2 for the plot.

size	number of runs	trace length	runtime [ns]	throughput [MEvents/s]
1	$1 \cdot 10^0$	$2 \cdot 10^3$	1,863,140	1.07
2	$1 \cdot 10^0$	$2 \cdot 10^3$	2,423,040	0.83
3	$1 \cdot 10^0$	$2 \cdot 10^3$	2,982,880	0.67
4	$1 \cdot 10^0$	$2 \cdot 10^3$	3,542,860	0.56
5	$1 \cdot 10^0$	$2 \cdot 10^3$	3,902,790	0.51
6	$1 \cdot 10^0$	$2 \cdot 10^3$	4,662,800	0.43
7	$1 \cdot 10^0$	$2 \cdot 10^3$	5,222,510	0.38
8	$1 \cdot 10^0$	$2 \cdot 10^3$	5,782,580	0.35
9	$1 \cdot 10^0$	$2 \cdot 10^3$	6,342,770	0.32

Table A.15.: Runtime of RECURSION with EPUs. See Figure 8.14 in Section 8.3.2 for the plot.

size	number of runs	trace length	runtime [ns]	throughput [MEvents/s]
1	$1 \cdot 10^0$	$2 \cdot 10^3$	40,090	49.89
2	$1 \cdot 10^0$	$2 \cdot 10^3$	60,090	33.28
3	$1 \cdot 10^0$	$2 \cdot 10^3$	60,090	33.28
4	$1 \cdot 10^0$	$2 \cdot 10^3$	80,090	24.97
5	$1 \cdot 10^0$	$2 \cdot 10^3$	80,090	24.97
6	$1 \cdot 10^0$	$2 \cdot 10^3$	100,090	19.98
7	$1 \cdot 10^0$	$2 \cdot 10^3$	100,090	19.98
8	$1 \cdot 10^0$	$2 \cdot 10^3$	120,090	16.65
9	$1 \cdot 10^0$	$2 \cdot 10^3$	120,090	16.65
10	$1 \cdot 10^0$	$2 \cdot 10^3$	140,090	14.28

Table A.16.: Runtime of RECURSION with FPGA. See Figure 8.14 in Section 8.3.2 for the plot.

size	number of runs	trace length	runtime [ns]	throughput [MEvents/s]
1	$1 \cdot 10^1$	$1 \cdot 10^6$	1,293,576,656	0.77
2	$1 \cdot 10^1$	$1 \cdot 10^6$	1,282,129,736	0.78
3	$1 \cdot 10^1$	$1 \cdot 10^6$	1,287,247,647	0.78
4	$1 \cdot 10^1$	$1 \cdot 10^6$	1,288,393,030	0.78
5	$1 \cdot 10^1$	$1 \cdot 10^6$	1,284,132,299	0.78
6	$1 \cdot 10^1$	$1 \cdot 10^6$	1,284,469,244	0.78
7	$1 \cdot 10^1$	$1 \cdot 10^6$	1,297,113,069	0.77
8	$1 \cdot 10^1$	$1 \cdot 10^6$	1,291,197,404	0.77
9	$1 \cdot 10^1$	$1 \cdot 10^6$	1,284,472,190	0.78
10	$1 \cdot 10^1$	$1 \cdot 10^6$	1,264,121,985	0.79

Table A.17.: Runtime of RECURSION in the interpreter. See Figure 8.14 in Section 8.3.2 for the plot.

size	number of runs	trace length	runtime [ns]	throughput [MEvents/s]
1	$5 \cdot 10^2$	$1 \cdot 10^6$	10,565,228	94.65
2	$5 \cdot 10^2$	$1 \cdot 10^6$	10,048,449	99.52
3	$5 \cdot 10^2$	$1 \cdot 10^6$	9,959,853	100.40
4	$5 \cdot 10^2$	$1 \cdot 10^6$	10,491,837	95.31
5	$5 \cdot 10^2$	$1 \cdot 10^6$	10,035,661	99.64
6	$5 \cdot 10^2$	$1 \cdot 10^6$	10,291,439	97.17
7	$5 \cdot 10^2$	$1 \cdot 10^6$	9,689,444	103.21
8	$5 \cdot 10^2$	$1 \cdot 10^6$	10,055,936	99.44
9	$5 \cdot 10^2$	$1 \cdot 10^6$	11,152,585	89.67
10	$5 \cdot 10^2$	$1 \cdot 10^6$	10,971,842	91.14

Table A.18.: Runtime of RECURSION with a compiled monitor in Rust reading events from an array. See Figure 8.14 in Section 8.3.2 for the plot.

size	number of runs	trace length	runtime [ns]	throughput [MEvents/s]
1	$1 \cdot 10^0$	$2 \cdot 10^4$	1,401,920	14.27
2	$1 \cdot 10^0$	$2 \cdot 10^4$	701,970	28.49
3	$1 \cdot 10^0$	$2 \cdot 10^4$	468,780	42.66
4	$1 \cdot 10^0$	$2 \cdot 10^4$	352,150	56.79
5	$1 \cdot 10^0$	$2 \cdot 10^4$	282,190	70.87
6	$1 \cdot 10^0$	$2 \cdot 10^4$	235,700	84.85
7	$1 \cdot 10^0$	$2 \cdot 10^4$	202,390	98.82
8	$1 \cdot 10^0$	$2 \cdot 10^4$	202,360	98.83
9	$1 \cdot 10^0$	$2 \cdot 10^4$	202,290	98.87
10	$1 \cdot 10^0$	$2 \cdot 10^4$	202,190	98.92

Table A.19.: Runtime of INPUTS with EPUs. See Figure 8.15 in Section 8.3.3 for the plot.

size	number of runs	trace length	runtime [ns]	throughput [MEvents/s]
1	$1 \cdot 10^0$	$2 \cdot 10^4$	1,401,920	14.27
2	$1 \cdot 10^0$	$2 \cdot 10^4$	701,970	28.49
3	$1 \cdot 10^0$	$2 \cdot 10^4$	468,780	42.66
4	$1 \cdot 10^0$	$2 \cdot 10^4$	352,150	56.79
5	$1 \cdot 10^0$	$2 \cdot 10^4$	282,190	70.87
6	$1 \cdot 10^0$	$2 \cdot 10^4$	235,700	84.85
7	$1 \cdot 10^0$	$2 \cdot 10^4$	202,390	98.82
8	$1 \cdot 10^0$	$2 \cdot 10^4$	202,360	98.83
9	$1 \cdot 10^0$	$2 \cdot 10^4$	202,290	98.87
10	$1 \cdot 10^0$	$2 \cdot 10^4$	202,190	98.92

Table A.20.: Runtime of INPUTS with EPUs and same timestamps. See Figure 8.15 in Section 8.3.3 for the plot.

size	number of runs	trace length	runtime [ns]	throughput [MEvents/s]
1	$1 \cdot 10^0$	$1 \cdot 10^4$	200,050	49.99
2	$1 \cdot 10^0$	$1 \cdot 10^4$	200,060	49.99
3	$1 \cdot 10^0$	$1 \cdot 10^4$	200,070	49.98
4	$1 \cdot 10^0$	$1 \cdot 10^4$	200,080	49.98
5	$1 \cdot 10^0$	$1 \cdot 10^4$	200,090	49.98
6	$1 \cdot 10^0$	$1 \cdot 10^4$	200,100	49.98
7	$1 \cdot 10^0$	$1 \cdot 10^4$	200,110	49.97
8	$1 \cdot 10^0$	$1 \cdot 10^4$	200,120	49.97
9	$1 \cdot 10^0$	$1 \cdot 10^4$	200,130	49.97
10	$1 \cdot 10^0$	$1 \cdot 10^4$	200,140	49.97

Table A.21.: Runtime of INPUTS with FPGA. See Figure 8.15 in Section 8.3.3 for the plot.

size	number of runs	trace length	runtime [ns]	throughput [MEvents/s]
1	$1 \cdot 10^0$	$1 \cdot 10^4$	200,050	49.99
2	$1 \cdot 10^0$	$1 \cdot 10^4$	150,060	66.64
3	$1 \cdot 10^0$	$1 \cdot 10^4$	133,410	74.96
4	$1 \cdot 10^0$	$1 \cdot 10^4$	125,080	79.95
5	$1 \cdot 10^0$	$1 \cdot 10^4$	120,090	83.27
6	$1 \cdot 10^0$	$1 \cdot 10^4$	116,770	85.64
7	$1 \cdot 10^0$	$1 \cdot 10^4$	114,400	87.41
8	$1 \cdot 10^0$	$1 \cdot 10^4$	112,620	88.79
9	$1 \cdot 10^0$	$1 \cdot 10^4$	111,250	89.89
10	$1 \cdot 10^0$	$1 \cdot 10^4$	110,140	90.79

Table A.22.: Runtime of INPUTS with FPGA and same timestamps. See Figure 8.15 in Section 8.3.3 for the plot.

size	number of runs	trace length	runtime [ns]	throughput [MEvents/s]
1	$3 \cdot 10^1$	$1 \cdot 10^7$	438,282,992	22.82
2	$3 \cdot 10^1$	$1 \cdot 10^7$	943,497,022	10.60
3	$3 \cdot 10^1$	$1 \cdot 10^7$	1,501,698,020	6.66
4	$3 \cdot 10^1$	$1 \cdot 10^7$	1,933,665,874	5.17
5	$3 \cdot 10^1$	$1 \cdot 10^7$	2,493,617,556	4.01
6	$3 \cdot 10^1$	$1 \cdot 10^7$	3,038,784,779	3.29
7	$3 \cdot 10^1$	$1 \cdot 10^7$	3,549,832,799	2.82
8	$3 \cdot 10^1$	$1 \cdot 10^7$	4,195,010,789	2.38
9	$3 \cdot 10^1$	$1 \cdot 10^7$	4,721,059,822	2.12
10	$3 \cdot 10^1$	$1 \cdot 10^7$	5,480,912,964	1.82

Table A.23.: Runtime of INPUTS with a compiled monitor in Rust reading events from an array. See Figure 8.14 in Section 8.3.2 for the plot.

size	number of runs	trace length	runtime [ns]	throughput [MEvents/s]
1	$3 \cdot 10^1$	$1 \cdot 10^7$	454,581,374	22.00
2	$3 \cdot 10^1$	$1 \cdot 10^7$	450,094,565	22.22
3	$3 \cdot 10^1$	$1 \cdot 10^7$	473,836,104	21.10
4	$3 \cdot 10^1$	$1 \cdot 10^7$	470,959,807	21.23
5	$3 \cdot 10^1$	$1 \cdot 10^7$	468,193,715	21.36
6	$3 \cdot 10^1$	$1 \cdot 10^7$	478,036,733	20.92
7	$3 \cdot 10^1$	$1 \cdot 10^7$	474,696,460	21.07
8	$3 \cdot 10^1$	$1 \cdot 10^7$	493,909,965	20.25
9	$3 \cdot 10^1$	$1 \cdot 10^7$	491,353,147	20.35
10	$3 \cdot 10^1$	$1 \cdot 10^7$	523,353,844	19.11

Table A.24.: Runtime of INPUTS with a compiled monitor in Rust reading events from an array and same timestamps. See Figure 8.14 in Section 8.3.2 for the plot.

size	number of runs	trace length	runtime [ns]	throughput [MEvents/s]
1	$3 \cdot 10^1$	$1 \cdot 10^5$	95,947,040	1.04
2	$3 \cdot 10^1$	$1 \cdot 10^5$	175,342,749	0.57
3	$3 \cdot 10^1$	$1 \cdot 10^5$	253,340,832	0.39
4	$3 \cdot 10^1$	$1 \cdot 10^5$	336,888,281	0.30
5	$3 \cdot 10^1$	$1 \cdot 10^5$	420,987,502	0.24
6	$3 \cdot 10^1$	$1 \cdot 10^5$	499,716,627	0.20
7	$3 \cdot 10^1$	$1 \cdot 10^5$	581,990,990	0.17
8	$3 \cdot 10^1$	$1 \cdot 10^5$	656,499,892	0.15
9	$3 \cdot 10^1$	$1 \cdot 10^5$	726,384,785	0.14
10	$3 \cdot 10^1$	$1 \cdot 10^5$	810,786,992	0.12

Table A.25.: Runtime of INPUTS in the interpreter. See Figure 8.16 in Section 8.3.3 for the plot.

size	number of runs	trace length	runtime [ns]	throughput [MEvents/s]
1	$3 \cdot 10^1$	$1 \cdot 10^5$	97,923,099	1.02
2	$3 \cdot 10^1$	$1 \cdot 10^5$	95,337,957	1.05
3	$3 \cdot 10^1$	$1 \cdot 10^5$	96,204,118	1.04
4	$3 \cdot 10^1$	$1 \cdot 10^5$	96,906,743	1.03
5	$3 \cdot 10^1$	$1 \cdot 10^5$	96,031,745	1.04
6	$3 \cdot 10^1$	$1 \cdot 10^5$	96,534,117	1.04
7	$3 \cdot 10^1$	$1 \cdot 10^5$	97,423,466	1.03
8	$3 \cdot 10^1$	$1 \cdot 10^5$	96,791,496	1.03
9	$3 \cdot 10^1$	$1 \cdot 10^5$	96,365,410	1.04
10	$3 \cdot 10^1$	$1 \cdot 10^5$	97,469,643	1.03

Table A.26.: Runtime of INPUTS in the interpreter with same timestamps. See Figure 8.16 in Section 8.3.3 for the plot.

backend	spec. depth	TeSSLa compiler [s]	Rust compiler [s]	FPGA synthesis [s]	FPGA impl. [s]	total [s]
compiler	2	1.0	0.4	—	—	1.4
compiler	20	1.3	0.6	—	—	1.9
compiler	200	19.0	10.7	—	—	29.7
EPUs	2	3.6	—	—	—	3.6
EPUs	20	3.7	—	—	—	3.7
FPGA	2	3.6	—	48.0	138.0	189.6
FPGA	20	4.1	—	49.0	144.0	197.1
FPGA	200	6.8	—	59.0	152.0	217.8

Table A.27.: Compilation time of LONG on different backends with different specification depths. See Figure 8.18 in Section 8.4 for the plot.

A.4. Hardware Utilisation

Figures A.1 to A.4 below show the hardware consumption of the synthesised specification LONG on the FPGA. See Section 8.3.1 and Figure 8.13 for a discussion.

Figure A.1.: Hardware utilisation of the specification LONG with size 1 visualised in Xilinx Vivado. The following parts of the hardware design are coloured: ■ the input FIFO before the actual monitor, 32 bit × 512 entries, ■ the output FIFO after the actual monitor, same size, ■ the Xillybus I/O logic and ■ the actual specification.

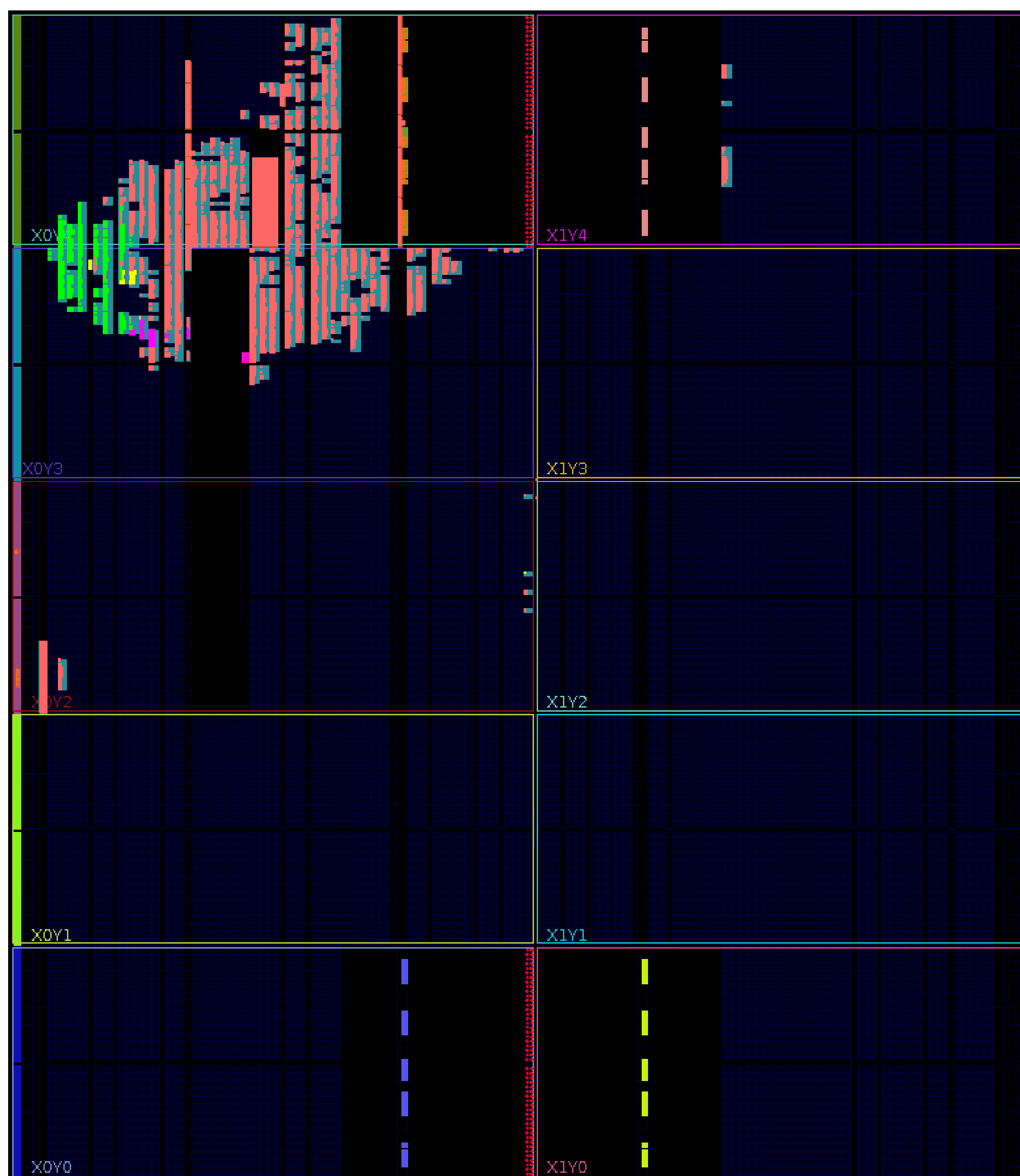


Figure A.2.: Hardware utilisation of the specification LONG with size 10 visualised in Xilinx Vivado. The following parts of the hardware design are coloured: ■ the input FIFO before the actual monitor, 32 bit × 512 entries, ■ the output FIFO after the actual monitor, same size, ■ the Xillybus I/O logic and ■ the actual specification.

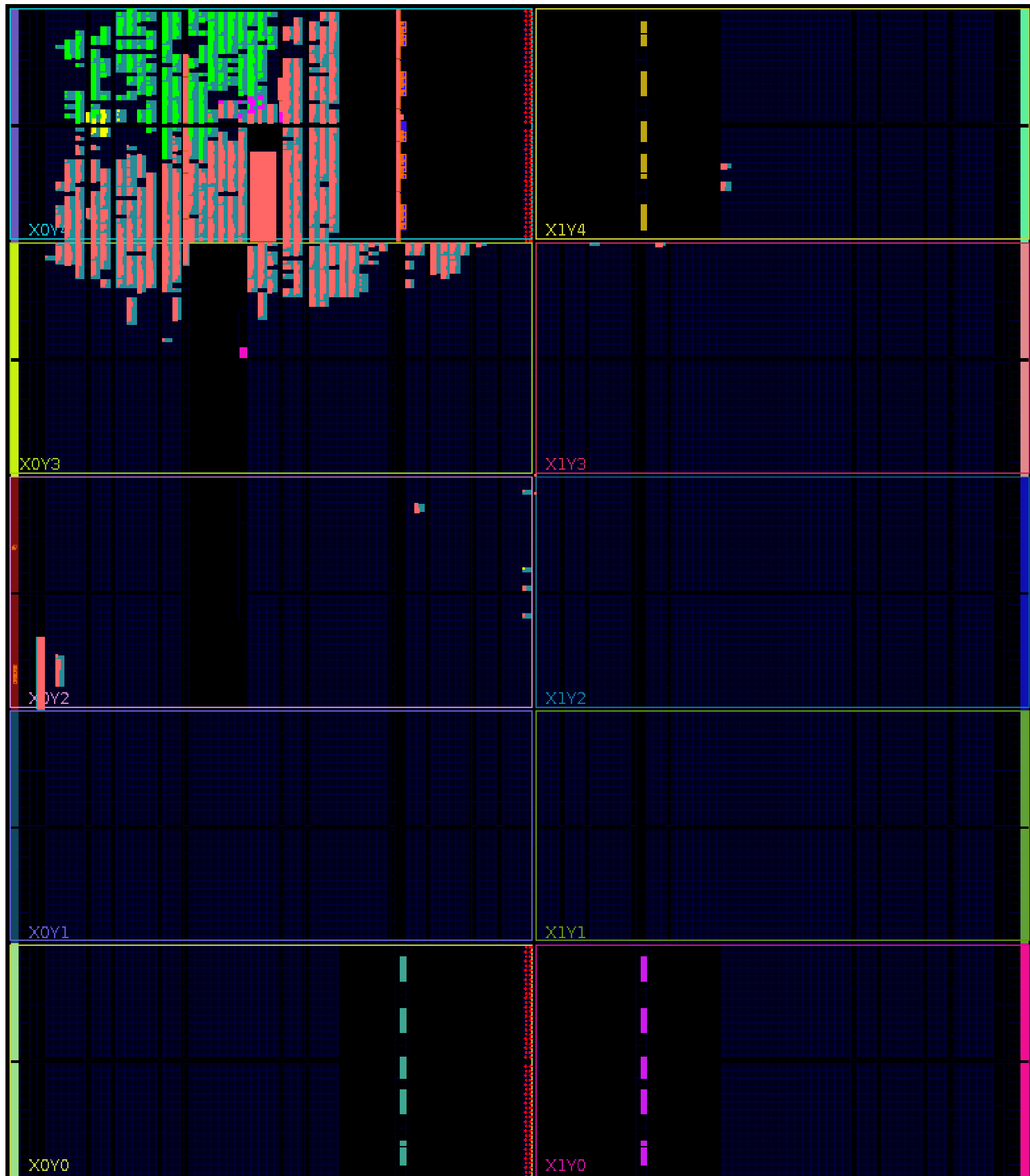


Figure A.3.: Hardware utilisation of the specification LONG with size 100 visualised in Xilinx Vivado. The following parts of the hardware design are coloured: ■ the input FIFO before the actual monitor, 32 bit × 512 entries, ■ the output FIFO after the actual monitor, same size, ■ the Xillybus I/O logic and ■ the actual specification.

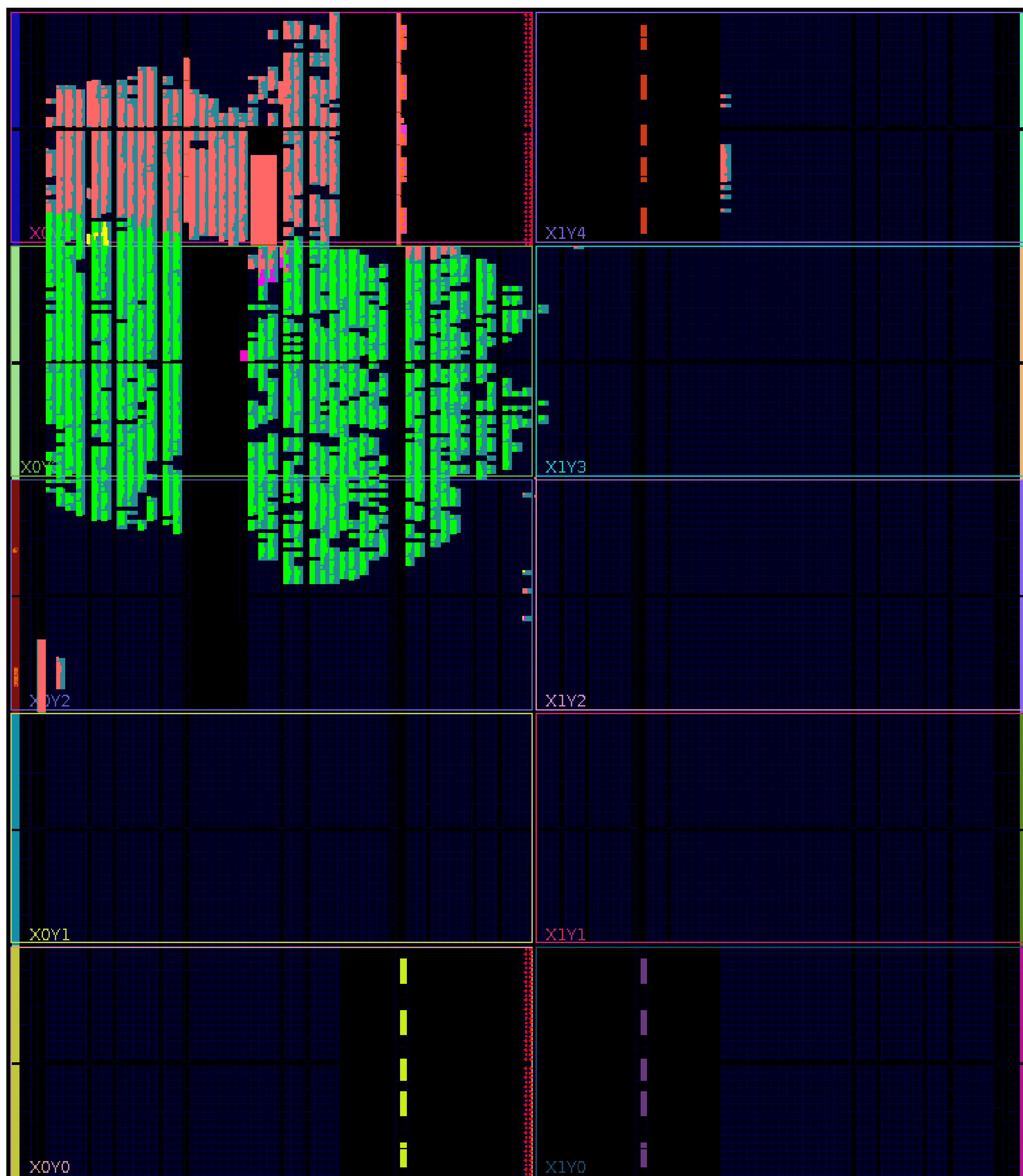


Figure A.4.: Hardware utilisation of the specification LONG with size 1000 visualised in Xilinx Vivado. The following parts of the hardware design are coloured: ■ the input FIFO before the actual monitor, 32 bit × 512 entries, ■ the output FIFO after the actual monitor, same size, ■ the Xillybus I/O logic and ■ the actual specification.

List of Figures

1.1. Overview of the different semantics and implementations introduced in this thesis . . . 14

3.1. Dependency graph of a well-formed specification . . . 51
3.2. Dependency graph of a not-well-formed specification . . . 52
3.3. Comparison of the filter operator and the sfilter operator . . . 68
3.4. Visualisation how the fixed point is computed for the counting example 82
3.5. Visualisation how the fixed point is computed for a recursion without a base case . . . 84
3.6. Visualisation of the evaluation of a not-well-formed simple recursion 85
3.7. Visualisation of the evaluation of a not-well-formed advanced recursion 85
3.8. Visualisation of the evaluation of a not-well-formed advanced recursion over discrete data domain . . . 87
3.9. Visualisation of the evaluation of the period example . . . 87
3.10. Visualisation of the evaluation of the period-recursion without base case . . . 90
3.11. Visualisation of the evaluation of the variable frequency period . . . 90
3.12. Stream visualisation showing the allowed and forbidden causal dependencies for a future-independent function on monitoring streams 97
3.13. Stream visualisation showing exemplary applications of a function that is not future independent . . . 98
3.14. Exemplary application of a future-independent function . . . 99

4.1. The common event driven synchronous execution scheme . . . 106
4.2. Evaluation of the synchronous monitoring function for a variable frequency period example . . . 125
4.3. UML class diagram of the interpreter . . . 136
4.4. UML object diagram for the variable frequency period example . . . 137
4.5. Integration test setup for the software compiler in comparison with the interpreter used as reference implementation . . . 147

5.1. Von Neumann control flow architecture and basic data flow architecture . . . 151
5.2. Exemplary message routing between EPUs . . . 154
5.3. Message frame encoding of the EPUs . . . 155

5.4. Architectural overview of the components of an Event Processing Unit (EPU) . . . 155
5.5. Schematic visualisation of the stages of the inner pipeline over time 158
5.6. EPU network diagram for the EPU network example . . . 162
5.7. EPU network diagrams for the forward command and the commands for **time**, the unary slift(f) and default . . . 171
5.8. EPU network diagrams for the commands for merge and **last** . . . 173
5.9. EPU network diagrams for the commands for the binary slift(f) and filter . . . 175
5.10. EPU network diagram for the command for slift(ite) . . . 176
5.11. EPU network diagram for the absolute value example . . . 178
5.12. Exemplary evaluation of the absolute value example using the synchronous semantics . . . 179
5.13. Exemplary evaluation of the EPU network derived from the absolute value example . . . 179
5.14. EPU network diagrams for the commands for blocking **last** . . . 183
5.15. EPU network diagram for the recursive example . . . 186
5.16. Exemplary evaluation of the recursive example using the synchronous semantics . . . 186
5.17. Exemplary evaluation of the EPU network derived from the recursive example . . . 187
5.18. Schematic representation of an condition logic tree of depth 3 . . . 191
5.19. Condition tree . . . 193
5.20. Tree representations . . . 195
5.21. Adjusted EPU network diagram for the absolute value example . . . 196
5.22. Abstract flow graph of recursive specifications suitable for a simple translation . . . 201
5.23. EPU network diagram for the simplified translation of the recursive example . . . 202
5.24. EPU network diagrams for the commands for foldLift . . . 205
5.25. Architectural overview diagram of the EPU hardware and test setup 208

6.1. Exemplary evaluation of the specification $\ell = \textbf{last}(v, r)$ with monitoring semantics in comparison with two abstractions . . . 213
6.2. Visualisation of abstract monitoring streams . . . 215
6.3. Comparison of the abstract monitoring semantics and the monitoring semantics . . . 221
6.4. Comparison of the abstract monitoring semantics and the monitoring semantics . . . 226
6.5. Variable frequency period with abstract monitoring streams . . . 228

7.1. Top-down view of a simple, generic FPGA architecture . . . 239

7.2. Simplified logic block 239
7.3. Operator network of the variable frequency period specification . . . 257
7.4. Exemplary execution of the operator network for the variable frequency period specification 258
7.5. A channel connecting a source with a sink 262
7.6. Tuplification for parallel **lift** operators 274
7.7. Tuplification for parallel **last** operators 276
7.8. Integration Test Setup using PCI Express via Xillybus 277
7.9. Integration test setup for the simulation in software using Chisel's simulation engine Treadle 279

8.1. Runtime and throughput of BURST in the interpreter in dependence of the trace length 283
8.2. Distribution of the runtime of LONG (with a fixed specification depth of 1) 285
8.3. Throughput of LONG in dependence of the specification depth with different compilers and I/O options 287
8.4. Dependency graph of the specification EVENTCHAIN and simplified architecture of the system under test 292
8.5. Dependency graphs of the specifications RUNTIME, TOGGLE and RESETCOUNT 293
8.6. Dependency graph of the specifications ROSACE and INCDEC 293
8.7. Dependency graph of the specification BURST 294
8.8. Throughput of different specifications on the EPU backend 297
8.9. Throughput of different specifications on different backends 298
8.10. Dependency graphs of synthetic specifications LONG, RECURSION and INPUTS 301
8.11. Throughput of LONG in dependence of the specification depth with different backends 302
8.12. Hardware utilisation of the specification LONG with size 1 visualised in Xilinx Vivado. 303
8.13. Hardware utilisation of the specification LONG with different sizes . 304
8.14. Throughput of RECURSION in dependence of the recursion depth with different backends 305
8.15. Throughput of INPUT in dependence of the number of inputs with different backends 306
8.16. Throughput of INPUT in dependence of the number of inputs with the interpreter 308
8.17. Qualitative overview of the throughput as a function of the depth of the specification, the recursion depth of the specification, and the parallelism of the specification on the different backends 309

8.18. Compilation time of LONG on different backends with different specification depths 311

A.1. Hardware utilisation of the specification LONG with size 1 visualised in Xilinx Vivado. 348
A.2. Hardware utilisation of the specification LONG with size 10 visualised in Xilinx Vivado. 349
A.3. Hardware utilisation of the specification LONG with size 100 visualised in Xilinx Vivado. 350
A.4. Hardware utilisation of the specification LONG with size 1000 visualised in Xilinx Vivado. 351

List of Tables

A.1. Runtime of BURST in the interpreter with different trace length . . . 329
A.2. Throughput of different specifications on different backends 330
A.3. Throughput of different specifications on the EPU backend without the foldLift optimisation 331
A.4. Throughput of BURST on the EPU backend with the foldLift optimisation used only for the counting 331
A.5. Runtime of LONG with EPUs 331
A.6. Runtime of LONG with FPGA 331
A.7. Runtime of LONG in the interpreter 332
A.8. Runtime of LONG with a compiled monitor in Rust reading events from an array 333
A.9. Runtime of LONG with a compiled monitor in Rust generating the events in the monitor 334
A.10. Runtime of LONG with a compiled monitor in Rust reading events from a binary file 335
A.11. Runtime of LONG with a compiled monitor in Java reading events from an ArrayList 336
A.12. Runtime of LONG with a compiled monitor in Java reading events from an array 337
A.13. Runtime of LONG with a compiled monitor in Java reading events from a binary file 338
A.14. Runtime of LONG with a compiled monitor in Java generating the events in the monitor 339
A.15. Runtime of RECURSION with EPUs 340
A.16. Runtime of RECURSION with FPGA 340
A.17. Runtime of RECURSION in the interpreter 341
A.18. Runtime of RECURSION with a compiled monitor in Rust reading events from an array 341
A.19. Runtime of INPUTS with EPUs 342
A.20. Runtime of INPUTS with EPUs and same timestamps 342
A.21. Runtime of INPUTS with EPUs and same timestamps 343
A.22. Runtime of INPUTS with FPGA and same timestamps 343
A.23. Runtime of INPUTS with a compiled monitor in Rust reading events from an array 344

A.24. Runtime of INPUTS with a compiled monitor in Rust reading events from an array and same timestamps 344
A.25. Runtime of INPUTS in the interpreter 345
A.26. Runtime of INPUTS in the interpreter with same timestamps 345
A.27. Compilation time of LONG on different backends with different specification depths 346

List of Definitions and Theorems

Definition 3.12 (Time Domain) 41
Definition 3.13 (Data Domain) 41
Definition 3.14 (Stream) 41
Definition 3.15 (Timestamps of a Stream) 42
Definition 3.17 (Zeno [Lam02, Mos07, ZJLS00]) 42
Definition 3.20 (Limit of a Stream) 44
Definition 3.21 (Functional View of a Stream) 44
Definition 3.22 (TeSSLa Syntax) 45
Definition 3.23 (TeSSLa Semantics [Sch20]) 45
Definition 3.24 (Semantics of the Operator **unit** [Sch20]) 46
Definition 3.25 (Semantics of the Operator **time** [Sch20]) 46
Definition 3.26 (Semantics of the Operator **lift** [Sch20]) 47
Definition 3.27 (Semantics of the Operator **last** [Sch20]) 47
Definition 3.28 (Semantics of the Operator **delay**) 48
Definition 3.29 (Equivalence of TeSSLa Specifications) 49
Definition 3.30 (Flat [CHL+18]) 50
Definition 3.31 (Dependency and Flow Graph [CHL+18]) 50
Definition 3.32 (Well-Formed TeSSLa Specification [CHL+18]) 50
Definition 3.35 (Semantics of the Operator nil) 53
Definition 3.36 (Semantics of the Operator const) 53
Definition 3.37 (Semantics of the Operator merge) 54
Definition 3.38 (Semantics of the Operators prev, sync and on) 54
Definition 3.39 (Semantics of the Operator slift) 56
Definition 3.40 (Semantics of the Operator filter) 56
Definition 3.41 (Semantics of the Operator default) 57
Definition 3.42 (Semantics of the Operator foldn) 58
Definition 3.43 (Semantics of the Operator fold) 59
Definition 3.44 (Semantics of the Operator reduce) 59
Definition 3.45 (Semantics of the Aggregation Operators count, sum, minimum and maximum) 60
Definition 3.46 (Semantics of the Operator resetCount) 60
Definition 3.47 (Semantics of the Operator delayedLast) 61
Lemma 3.48 (Associativity of **lift**) 64
Lemma 3.49 (Oversampling of Signals) 64

Lemma 3.50 (Associativity of slift) 64
Theorem 3.51 (Signal Lift and Default) 66
Definition 3.52 (Semantics of the Operator sfilter) 67
Definition 3.53 (Semantics of the Operator prevn) 70
Definition 3.54 (Semantics of the Operator mdelay) 71
Definition 3.55 (Semantics of the Operator mdelayedLast) 72
Definition 3.56 (Semantics of the Operator shift) 73
Definition 3.57 (Monitoring Stream) 74
Definition 3.58 (Monitoring Stream of Independent Events) 74
Definition 3.59 (Timestamps of a Monitoring Stream) 75
Definition 3.62 (Limit of a Monitoring Stream) 76
Definition 3.63 (Refinement Relation) 77
Lemma 3.64 (Refinement Relation is a Dcpo) 77
Definition 3.65 (TeSSLa Operators on Monitoring Streams) 77
Definition 3.67 (TeSSLa Monitoring Semantics) 78
Definition 3.68 (Monitoring-Equivalence of TeSSLa Specifications) 79
Lemma 3.69 (Scott-Continuity of the TeSSLa Operators) 79
Lemma 3.81 (Construction of the Least Fixed Point) 89
Theorem 3.82 (Uniqueness of the Fixed Point in the Monitoring Semantics) 91
Lemma 3.83 (TeSSLa Monitoring Semantics is Scott-Continuous) 92
Definition 3.84 (Preserving Full Knowledge) 92
Lemma 3.85 (Relation Between TeSSLa Monitoring Semantics and TeSSLa Semantics) 93
Definition 3.86 (Maximal Refinement) 93
Theorem 3.87 (TeSSLa Monitoring Semantics Produces Maximal Refinement) 94
Lemma 3.88 (Relation of Equivalence and Monitoring Equivalence) 95
Corollary 3.89 (Uniqueness of the Fixed Point in the Semantics) 95
Definition 3.90 (Segments of a Monitoring Stream up to a Timestamp) . . 96
Definition 3.91 (Future Independence) 96
Lemma 3.94 (TeSSLa Monitoring Semantics is Future Independent) 99
Definition 3.95 (Timestamp Conservative) 100
Definition 3.96 (Timestamp-Conservative TeSSLa) 100
Lemma 3.97 (Timestamp-Conservative TeSSLa is Timestamp Conservative) 100
Lemma 3.98 (Expressiveness of Timestamp Conservative TeSSLa) 100
Lemma 3.99 (Expressiveness of TeSSLa) 101
Definition 3.100 (Behavioural Equivalence) 102
Lemma 3.101 (Behavioural Equivalence of Scott-Continuous Functions) . . 102
Theorem 3.102 (Expressiveness of TeSSLa) 103

Definition 4.1 (Progress of a Monitoring Stream) 107
Lemma 4.2 (Future Independence Preserves Progress) 108

Definition 4.3 (Synchronised Stream) 109
Definition 4.4 (Progress of a Synchronised Stream) 110
Definition 4.5 (Abstraction Function for Synchronised Streams) 110
Definition 4.6 (Concretisation Function for Synchronised Streams) 111
Lemma 4.7 (Abstraction Preserves Progress) 111
Definition 4.9 (Functional View of Synchronised Streams) 112
Definition 4.10 (Prefix Relation on Synchronised Streams) 113
Lemma 4.11 (Galois Connection for Synchronised Streams) 113
Definition 4.13 (Semantics of the Synchronous Operator **unit**^{s}) 115
Definition 4.14 (Semantics of the Synchronous Operator **time**^{s}) 115
Definition 4.15 (Semantics of the Synchronous Operator **lift**^{s}) 116
Definition 4.16 (Semantics of the Synchronous Operator **last**^{s}) 116
Definition 4.17 (Semantics of the Synchronous Operator **delay**^{s}) 116
Definition 4.18 (Joined Operator Function) 118
Definition 4.19 (Closed Operator Function) 118
Definition 4.20 (Synchronised Monitoring Function) 119
Theorem 4.24 (Correctness of Synchronised Monitoring) 126
Corollary 4.25 (Synchronised Monitoring is an Abstraction of the Monitoring Semantics) 126
Corollary 4.26 (Synchronised Monitoring is Behavioural Equivalent to the Monitoring Semantics) 127
Definition 4.27 (Imperative Algorithm for the Synchronised Monitoring Function) 128
Lemma 4.28 (Correctness of the Imperative Algorithm for the Synchronised Monitoring Function) 129
Definition 4.29 (Delayed and Immediate Inputs) 131
Definition 4.30 (Message-Passing Implementation of the Closed Operator Function) 131
Lemma 4.31 (Correctness of the Message-Passing Implementation of the Closed Operator Function) 132
Definition 4.32 (Linearising Implementation of the Closed Operator Function) 132
Lemma 4.33 (Correctness of the Linearising Implementation of the Closed Operator Function) 132

Definition 5.1 (EPU Command) 160
Definition 5.2 (EPU Network) 160
Definition 5.3 (Well-Formed EPU Network) 161
Definition 5.6 (Forward Command) 172
Definition 5.7 (Command for **time**) 172
Definition 5.8 (Commands for $\mathsf{slift}(f)$ With a Unary Arithmetic Operation f) 172

Definition 5.9 (Commands for default) 173
Definition 5.10 (Commands for merge) 173
Definition 5.11 (Commands for **last**) 174
Definition 5.12 (Commands for slift(f) With a Binary Arithmetic Operation f) 175
Definition 5.13 (Commands for filter) 175
Definition 5.14 (Commands for slift(ite)) 176
Definition 5.15 (Commands for Blocking last) 182
Theorem 5.16 188
Definition 5.20 (Semantics of the Operator foldLift) 204
Definition 5.21 (Commands for foldLift) 204

Definition 6.2 (Abstract Monitoring Streams [CHL+18]) 213
Definition 6.3 (Timestamps of an Abstract Monitoring Stream) 214
Definition 6.4 (Progress of an Abstract Monitoring Stream) 214
Definition 6.6 (Abstraction Function for Abstract Monitoring Streams) . . 216
Definition 6.7 (Concretisation Function for Abstract Monitoring Streams) . 216
Definition 6.8 (Functional View of Synchronised Streams) 217
Definition 6.9 (Prefix Relation on Abstract Monitoring Streams [CHL+18]) 217
Lemma 6.10 (Galois Connection for Abstract Monitoring Streams) 218
Definition 6.12 (Semantics of the Abstract Operator **unit**$^\#$ [CHL+18]) . . . 219
Definition 6.13 (Semantics of the Abstract Operator **time**$^\#$ [CHL+18]) . . 219
Definition 6.14 (Semantics of the Abstract Operator **lift**$^\#$ [CHL+18]) . . . 219
Definition 6.15 (Semantics of the Abstract Operator **last**$^\#$ [CHL+18]) . . . 220
Definition 6.18 (Semantics of the Operator **delayR**) 223
Definition 6.19 (Progressing TeSSLa) 223
Lemma 6.20 (Expressiveness of Progressing TeSSLa) 224
Definition 6.21 (Semantics of the Abstract Operator **delayR**$^\#$ [CHL+18]) . 225
Lemma 6.24 (Perfectness of the Abstract TeSSLa Operators) 229
Definition 6.25 (TeSSLa Abstract Monitoring Semantics [CHL+18]) 229
Lemma 6.26 (Construction of the Least Fixed Point [CHL+18]) 230
Theorem 6.27 (Uniqueness of the Fixed Point [CHL+18]) 230
Theorem 6.28 (TeSSLa Abstract Monitoring Semantics is an Abstraction) 231
Definition 6.30 (Preserving Full Knowledge on Abstract Monitoring Streams) 232
Lemma 6.31 (Relation Between the Abstract TeSSLa Monitoring Semantics and the TeSSLa Semantics) 233
Corollary 6.32 (Equivalence of Abstract TeSSLa Specifications) 234

Definition 7.2 (Semantics of the Abstract Operator **delayR**‡) 241
Definition 7.3 (Semantics of the Abstract Operator **last**‡) 242
Definition 7.5 (Adjusted TeSSLa Abstract Monitoring Semantics) 243
Definition 7.6 (Extended Time Domain) 244

Definition 7.7 (Channel) . . . 244
Definition 7.8 (Channel Operator) . . . 245
Definition 7.9 (Operator Network) . . . 245
Definition 7.10 (Scheduling) . . . 245
Definition 7.12 (Operator Network Function) . . . 246
Definition 7.13 (Channel Operator for $\mathbf{unit}^{\#}$) . . . 247
Definition 7.14 (Channel Operator for $\mathbf{time}^{\#}$) . . . 248
Definition 7.15 (Channel Operator for Unary $\mathbf{lift}^{\#}(f)$) . . . 248
Definition 7.16 (Channel Operator for binary $\mathbf{lift}^{\#}(f)$) . . . 249
Definition 7.17 (Channel Operator for $\mathbf{last}^{\ddagger}$) . . . 250
Definition 7.18 (Channel Operator for $\mathbf{delayR}^{\ddagger}$) . . . 251
Lemma 7.19 (Correctness of the TeSSLa Channel Operators) . . . 252
Definition 7.20 (Translating TeSSLa Specifications to Operator Networks) 253
Definition 7.21 (Synchronising Operator Network) . . . 253
Lemma 7.22 (TeSSLa's Operator Networks are Synchronising) . . . 253
Definition 7.23 (Progress of a Channel) . . . 254
Definition 7.24 (Progress Passing Channel Operators) . . . 254
Definition 7.25 (Progress Consuming Channel Operators) . . . 254
Lemma 7.26 (TeSSLa's Channel Operators are Progress Passing and Consuming) . . . 254
Definition 7.27 (Progress Increasing Channel Operators) . . . 255
Lemma 7.28 (Channel Operators for $\mathbf{last}^{\ddagger}$ and $\mathbf{delayR}^{\ddagger}$ are Progress Increasing) . . . 255
Theorem 7.29 (TeSSLa Operator Networks are Correct) . . . 255
Definition 7.30 (Equivalence of TeSSLa Specifications Regarding Timestamps) . . . 271
Definition 7.31 (Subsumption of TeSSLa Specification) . . . 272
Definition 7.32 (Dependent Streams) . . . 273
Definition 7.33 (Connected Cycles) . . . 273

Bibliography

[ABL95] Pascalin Amagbégnon, Loïc Besnard, and Paul Le Guernic. Implementation of the data-flow synchronous language SIGNAL. In *PLDI*, pages 163–173. ACM, 1995.

[AC03] Arvind and David E. Culler. Dataflow architectures. *Annual Review of Computer Science*, 1:225–253, 11 2003.

[ACM02] Eugene Asarin, Paul Caspi, and Oded Maler. Timed regular expressions. *J. ACM*, 49(2):172–206, 2002.

[AFR16] Rajeev Alur, Dana Fisman, and Mukund Raghothaman. Regular programming for quantitative properties of data streams. In *ESOP*, volume 9632 of *Lecture Notes in Computer Science*, pages 15–40. Springer, 2016.

[AJ94] Samson Abramsky and Achim Jung. Domain theory. In Samson Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of logic in computer science*, volume 3. Clarendon Press, 1994.

[AMHM02] Guido Araujo, Sharad Malik, Zhining Huang, and Nahri Moreano. Datapath merging and interconnection sharing for reconfigurable architectures. In *ISSS*, pages 38–43. ACM / IEEE Computer Society, 2002.

[AUT17] AUTOSAR. Specification of Timing Extensions. Technical report, AUTOSAR, 2017.

[Bak20] Denis Bakhvalov. Benchmarking: compare measurements and check which is faster. `https://easyperf.net/blog/2019/12/30/Comparing-performance-measurements`, January 2020. [Online; accessed 04.12.2020].

[BB18] Jürgen Becker and Falco K. Bapp. The aramis project initiative - multicore systems in safety- and mixed-critical applications. In *ARC*, volume 10824 of *Lecture Notes in Computer Science*, pages 685–699. Springer, 2018.

[BCE+03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[Ber00a] Gérard Berry. *The Esterel v5 language primer.* Centre de mathématiques appliquées, Ecole des mines and INRIA, 2000.

[Ber00b] Gérard Berry. The foundations of esterel. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 425–454. The MIT Press, 2000.

[Ber07] Gérard Berry. Scade: Synchronous design and validation of embedded control software. In S. Ramesh and Prahladavaradan Sampath, editors, *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, pages 19–33. Springer Netherlands, 2007.

[Ber16] Gerard Berry. Formally unifying modeling and design for embedded systems - A personal view. In *ISoLA (2)*, volume 9953 of *Lecture Notes in Computer Science*, pages 134–149, 2016.

[BFKS20] Jan Baumeister, Bernd Finkbeiner, Matthis Kruse, and Maximilian Schwenger. Automatic optimizations for stream-based monitoring languages. In *RV*, volume 12399 of *Lecture Notes in Computer Science*, pages 451–461. Springer, 2020.

[BFS+20] Jan Baumeister, Bernd Finkbeiner, Sebastian Schirmer, Maximilian Schwenger, and Christoph Torens. Rtlola cleared for take-off: Monitoring autonomous aircraft. In *CAV (2)*, volume 12225 of *Lecture Notes in Computer Science*, pages 28–39. Springer, 2020.

[BFST19] Jan Baumeister, Bernd Finkbeiner, Maximilian Schwenger, and Hazem Torfah. FPGA stream-monitoring of real-time properties. *ACM Trans. Embed. Comput. Syst.*, 18(5s):88:1–88:24, 2019.

[BFST20] Jan Baumeister, Bernd Finkbeiner, Maximilian Schwenger, and Hazem Torfah. FPGA stream-monitoring of real-time properties. *CoRR*, abs/2003.12477, 2020.

[BG92] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[BHW+13] Rico Backasch, Christian Hochberger, Alexander Weiss, Martin Leucker, and Richard Lasslop. Runtime verification for multicore soc with high-quality trace data. *ACM Trans. Design Autom. Electr. Syst.*, 18(2):18:1–18:26, 2013.

[BJK+05] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004]*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.

[BKH17] Mohamed Recem Boussaha, Raphaël Khoury, and Sylvain Hallé. Monitoring of security properties using beepbeep. In *FPS*, volume 10723 of *Lecture Notes in Computer Science*, pages 160–169. Springer, 2017.

[BL12] Alexander Brant and Guy G. F. Lemieux. ZUMA: an open FPGA overlay architecture. In *FCCM*, pages 93–96. IEEE Computer Society, 2012.

[BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, 2011.

[BM76] J. Adrian Bondy and Uppaluri S. R. Murty. *Graph Theory with Applications*. Macmillan Education UK, 1976.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerland, and Michael Stal. *Pattern-Oriented Software Architecture, A System of Patterns*. Wiley, 1996.

[BPR10] Jean Berstel, Dominique Perrin, and Christophe Reutenauer. *Codes and Automata*, volume 129 of *Encyclopedia of mathematics and its applications*. Cambridge University Press, 2010.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems - Focus on Streams, Interfaces, and Refinement*. Monographs in Computer Science. Springer, 2001.

[BS14] Laura Bozzelli and César Sánchez. Foundations of boolean stream runtime verification. In *RV*, volume 8734 of *Lecture Notes in Computer Science*, pages 64–79. Springer, 2014.

[BS16] Laura Bozzelli and César Sánchez. Foundations of boolean stream runtime verification. *Theor. Comput. Sci.*, 631:118–138, 2016.

[Buc20] Thiemo Bucciarelli. Synthesis of stream-based monitors on fpgas. Master's thesis, Universität zu Lübeck, 2020.

[BVR+12] Jonathan Bachrach, Huy Vo, Brian C. Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: constructing hardware in a scala embedded language. In *DAC*, pages 1216–1225. ACM, 2012.

[CA13] Davor Capalija and Tarek S. Abdelrahman. A high-performance overlay architecture for pipelined execution of data flow graphs. In *FPL*, pages 1–8. IEEE, 2013.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.

[CFM+97] Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko. Compiling java just in time. *IEEE Micro*, 17(3):36–43, 1997.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. The MIT Press, 1999.

[CGS20] Martín Ceresa, Felipe Gorostiaga, and César Sánchez. Declarative stream runtime verification (hlola). In *APLAS*, volume 12470 of *Lecture Notes in Computer Science*, pages 25–43. Springer, 2020.

[CHL+18] Lukas Convent, Sebastian Hungerecker, Martin Leucker, Torben Scheffel, Malte Schmitz, and Daniel Thoma. Tessla: Temporal stream-based specification language. In *SBMF*, volume 11254 of *Lecture Notes in Computer Science*, pages 144–162. Springer, 2018.

[CHS+18] Lukas Convent, Sebastian Hungerecker, Torben Scheffel, Malte Schmitz, Daniel Thoma, and Alexander Weiss. Hardware-based runtime verification with embedded tracing units and stream processing. In *RV*, volume 11237 of *Lecture Notes in Computer Science*, pages 43–63. Springer, 2018.

[CKNZ11] Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. Model checking and the state explosion problem. In Bertrand Meyer and Martin Nordio, editors, *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, volume 7682 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 2011.

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *POPL*, pages 178–188. ACM Press, 1987.

[CPP05] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A conservative extension of synchronous data-flow with state machines. In *EMSOFT*, pages 173–182. ACM, 2005.

[Dav19] Adam L. Davis. *Reactive streams in Java*. Apress, 2019.

[DDG+18] Normann Decker, Boris Dreyer, Philip Gottschling, Christian Hochberger, Alexander Lange, Martin Leucker, Torben Scheffel, Simon Wegener, and Alexander Weiss. Online analysis of debug trace data for embedded systems. In *DATE*, pages 851–856. IEEE, 2018.

[DG88] Jack B. Dennis and Guang R. Gao. An efficient pipelined dataflow processor architecture. In *SC*, pages 368–373. IEEE Computer Society, 1988.

[DG10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[DGH+17] Normann Decker, Philip Gottschling, Christian Hochberger, Martin Leucker, Torben Scheffel, Malte Schmitz, and Alexander Weiss. Rapidly adjustable non-intrusive online monitoring for multi-core systems. In *SBMF*, volume 10623 of *Lecture Notes in Computer Science*, pages 179–196. Springer, 2017.

[DLT16] Normann Decker, Martin Leucker, and Daniel Thoma. Monitoring modulo theories. *Int. J. Softw. Tools Technol. Transf.*, 18(2):205–225, 2016.

[DM74] Jack B. Dennis and David Misunas. A preliminary architecture for a basic data flow processor. In *ISCA*, pages 126–132. ACM, 1974.

[DMB+12] Alexandre Donzé, Oded Maler, Ezio Bartocci, Dejan Nickovic, Radu Grosu, and Scott A. Smolka. On temporal logic and signal processing. In *ATVA*, volume 7561 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2012.

[DMF12] Luca Deri, Simone Mainardi, and Francesco Fusco. tsdb: A compressed database for time series. In *TMA*, volume 7189 of *Lecture Notes in Computer Science*, pages 143–156. Springer, 2012.

[DSS+05] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. LOLA: runtime monitoring of synchronous systems. In *TIME*, pages 166–174. IEEE Computer Society, 2005.

[EGN18] Benjamin J Evans, James Gough, and Chris Newland. *Optimizing Java: Practical Techniques for Improving JVM Application Performance*. O'Reilly Media, 2018.

[EH97] Conal Elliott and Paul Hudak. Functional reactive animation. In *ICFP*, pages 263–273. ACM, 1997.

[EKMS93] Marcel Erné, Jürgen Koslowski, Austin Melton, and George E Strecker. A primer on galois connections. *Annals of the New York Academy of Sciences*, 704(1):103–125, 1993.

[Eld18] Schuyler Eldridge. What benefits does chisel offer over classic hardware description languages? `https://stackoverflow.com/questions/53007782/what-benefits-does-chisel-offer-over-classic-hardware-description-languages`, October 2018. [Online; accessed 24.06.2021].

[FFS+19] Peter Faymonville, Bernd Finkbeiner, Malte Schledjewski, Maximilian Schwenger, Marvin Stenger, Leander Tentrup, and Hazem Torfah. Streamlab: Stream-based monitoring of cyber-physical systems. In *CAV (1)*, volume 11561 of *Lecture Notes in Computer Science*, pages 421–431. Springer, 2019.

[FFST16] Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah. A stream-based specification language for network monitoring. In *RV*, volume 10012 of *Lecture Notes in Computer Science*, pages 152–168. Springer, 2016.

[FOPS20] Bernd Finkbeiner, Stefan Oswald, Noemi Passing, and Maximilian Schwenger. Verified rust monitors for lola specifications. In *RV*, volume 12399 of *Lecture Notes in Computer Science*, pages 431–450. Springer, 2020.

[GDPM13] Arda Goknil, Julien DeAntoni, Marie-Agnès Peraldi-Frati, and Frédéric Mallet. Tool support for the analysis of TADL2 timing constraints using timesquare. In *2013 18th International Conference on Engineering of Complex Computer Systems, Singapore, July 17-19, 2013*, pages 145–154. IEEE Computer Society, 2013.

[GH01] Dimitra Giannakopoulou and Klaus Havelund. Automata-based verification of temporal properties on running programs. In *ASE*, pages 412–416. IEEE Computer Society, 2001.

[GH17] Philip Gottschling and Christian Hochberger. Reep: A toolset for generation and programming of reconfigurable datapaths for event processing. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2017, Orlando / Buena Vista, FL, USA, May 29 - June 2, 2017*, pages 141–149. IEEE Computer Society, 2017.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[GHS11] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *HPCA*, pages 503–514. IEEE Computer Society, 2011.

[GL87] Thierry Gautier and Paul Le Guernic. SIGNAL: A declarative language for synchronous programming of real-time systems. In *FPCA*, volume 274 of *Lecture Notes in Computer Science*, pages 257–277. Springer, 1987.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[GS21] Felipe Gorostiaga and César Sánchez. Hlola: a very functional tool for extensible stream runtime verification. In *TACAS (2)*, volume 12652 of *Lecture Notes in Computer Science*, pages 349–356. Springer, 2021.

[GSM+99] Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Laufer. Piperench: A coprocessor for streaming multimedia acceleration. In *ISCA*, pages 28–39. IEEE Computer Society, 1999.

[GT05] Michael Grottke and Kishor S Trivedi. A classification of software faults. *Journal of Reliability Engineering Association of Japan*, 27(7):425–438, 2005.

[Hal05] Nicolas Halbwachs. A synchronous language at work: the story of lustre. In *MEMOCODE*, pages 3–11. IEEE Computer Society, 2005.

[Han18] James W. Hanlon. Writing synthesizable verilog. `https://jameswhanlon.com/writing-synthesizable-verilog.html`, May 2018. [Online; accessed 15.01.2021].

[HB85] Paul Hudak and Adrienne G. Bloss. The aggregate update problem in functional programming systems. In *POPL*, pages 300–314. ACM Press, 1985.

[HK08] Ali R. Hurson and Krishna M. Kavi. Dataflow computers: Their history and future. In *Wiley Encyclopedia of Computer Science and Engineering*. John Wiley & Sons, Inc., 2008.

[HK17] Sylvain Hallé and Raphaël Khoury. Event stream processing with beepbeep 3. In *RV-CuBES*, volume 3 of *Kalpa Publications in Computing*, pages 81–88. EasyChair, 2017.

[HK18] Sylvain Hallé and Raphaël Khoury. Writing domain-specific languages for beepbeep. In *RV*, volume 11237 of *Lecture Notes in Computer Science*, pages 447–457. Springer, 2018.

[HMU07] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley, 2007.

[HN03] Jerker Hammarberg and Simin Nadjm-Tehrani. Development of safety-critical reconfigurable hardware with esterel. *Electron. Notes Theor. Comput. Sci.*, 80:219–234, 2003.

[HR01] Klaus Havelund and Grigore Rosu. Monitoring programs using rewriting. In *ASE*, pages 135–143. IEEE Computer Society, 2001.

[HR02] Klaus Havelund and Grigore Rosu. Synthesizing monitors for safety properties. In *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.

[HRR91] Nicolas Halbwachs, Pascal Raymond, and Christophe Ratel. Generating efficient code from data-flow programs. In *PLILP*, volume 528 of *Lecture Notes in Computer Science*, pages 207–218. Springer, 1991.

[Hug89] John Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, 1989.

[Hut99] Graham Hutton. A tutorial on the universality and expressiveness of fold. *J. Funct. Program.*, 9(4):355–372, 1999.

[IKL+17] Adam M. Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *ICCAD*, pages 209–216. IEEE, 2017.

[JBG+15] Stefan Jaksic, Ezio Bartocci, Radu Grosu, Reinhard Kloibhofer, Thang Nguyen, and Dejan Nickovic. From signal temporal logic to FPGA monitors. In *MEMOCODE*, pages 218–227. IEEE, 2015.

[JBGN16] Stefan Jaksic, Ezio Bartocci, Radu Grosu, and Dejan Nickovic. Quantitative monitoring of STL with edit distance. In *RV*, volume 10012 of *LNCS*, pages 201–218. Springer, 2016.

[Kal19] Hannes Kallwies. Efficient code generation for stream-based specifications. Master's thesis, Universität zu Lübeck, 2019.

[KL19] Agus Kurniawan and Wely Lau. *Practical Azure Functions*. Apress, 2019.

[Kle56] Stephen C Kleene. Representation of events in nerve nets and finite automata. *Automata studies*, 34:3–41, 1956.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[KLS+22] Hannes Kallwies, Martin Leucker, Torben Scheffel, Malte Schmitz, and Daniel Thoma. Aggregate update problem for multi-clocked dataflow languages. In *CGO*, pages 79–91. IEEE, 2022.

[KVB+99] Moonjoo Kim, Mahesh Viswanathan, Hanêne Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Formally specified monitoring of temporal properties. In *ECRTS*, pages 114–122. IEEE Computer Society, 1999.

[LA04] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE Computer Society, 2004.

[Lam02] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[Lar14] Staffan Larsen. JDK-8040140: System.nanoTime() is slow and non-monotonic on OS X. `https://bugs.openjdk.java.net/browse/JDK-8040140`, April 2014. [Online; accessed 24.02.2022].

[Lat02] Chris Lattner. Llvm: An infrastructure for multi-stage optimization. Master's thesis, University of Illinois at Urbana-Champaign, 2002.

[Lat12] Chris Lattner. Llvm. In Amy Brown and Greg Wilson, editors, *The Architecture of Open Source Applications*. lulu.com, 2012.

[LS09] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Log. Algebraic Methods Program.*, 78(5):293–303, 2009.

[LSS+18] Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, and Alexander Schramm. Tessla: runtime verification of non-synchronized real-time streams. In *SAC*, pages 1925–1933. ACM, 2018.

[LSS+19] Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, and Daniel Thoma. Runtime verification for timed event streams with partial information. In *RV*, volume 11757 of *Lecture Notes in Computer Science*, pages 273–291. Springer, 2019.

[LSS+20] Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, and Alexander Schramm. Runtime verification of real-time event streams under non-synchronized arrival. *Softw. Qual. J.*, 28(2):745–787, 2020.

[Mar03] George Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8(14):1–6, 2003.

[Max04] Clive Maxfield. *The design warrior's guide to FPGAs: devices, tools and flows*. Elsevier, 2004.

[MN04] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *FORMATS/FTRTFT*, volume 3253 of *Lecture Notes in Computer Science*, pages 152–166. Springer, 2004.

[Mos07] Pieter J. Mosterman. Hybrid dynamic systems. In Paul A. Fishwick, editor, *Handbook of Dynamic System Modeling*. Chapman and Hall/CRC, 2007.

[MRS17] Patrick Moosbrugger, Kristin Y. Rozier, and Johann Schumann. R2U2: monitoring and diagnosis of security threats for unmanned aerial systems. *Formal Methods in System Design*, 51(1):31–61, 2017.

[Mye04] Glenford J. Myers. *The art of software testing (2. ed.)*. Wiley, 2004.

[Ore44] Oystein Ore. Galois connexions. *Transactions of the American Mathematical Society*, 55(3):493–513, 1944.

[Ort05] Jorge L. Ortega-Arjona. The pipes and filters pattern. A functional parallelism architectural pattern for parallel programming. In *EuroPLoP*, pages 637–650. UVK - Universitaetsverlag Konstanz, 2005.

[Pac18] Vinicius Feitosa Pacheco. *Microservice Patterns and Best Practices: Explore patterns like CQRS and event sourcing to create scalable, maintainable, and testable microservices*. Packt Publishing Ltd, 2018.

[PC90] Gregory M. Papadopoulos and David E. Culler. Monsoon: An explicit token-store architecture. In *ISCA*, pages 82–91. ACM, 1990.

[PEB07] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. *Compiling Esterel*. Springer, 2007.

[PF11] Terence Parr and Kathleen Fisher. Ll(*): the foundation of the ANTLR parser generator. In *PLDI*, pages 425–436. ACM, 2011.

[PGMN10] Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot: A hard real-time runtime monitor. In *RV*, volume 6418 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 2010.

[PMCR08] Rodolfo Pellizzoni, Patrick O'Neil Meredith, Marco Caccamo, and Grigore Rosu. Hardware runtime monitoring for dependable cots-based real-time embedded systems. In *Proceedings of the 29th IEEE Real-Time Systems Symposium, RTSS 2008, Barcelona, Spain, 30 November - 3 December 2008*, pages 481–491. IEEE Computer Society, 2008.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.

[PQ95] Terence John Parr and Russell W. Quong. ANTLR: A predicated-*LL(k)* parser generator. *Softw. Pract. Exp.*, 25(7):789–810, 1995.

[PS14] Thomas B. Preußer and Rainer G. Spallek. Ready pcie data streaming solutions for fpgas. In *FPL*, pages 1–4. IEEE, 2014.

[PSG+14] Claire Pagetti, David Saussié, Romain Gratia, Eric Noulard, and Pierre Siron. The ROSACE case study: From simulink specification to multi/many-core execution. In *RTAS*, pages 309–318. IEEE Computer Society, 2014.

[RFB14] Thomas Reinbacher, Matthias Függer, and Jörg Brauer. Runtime verification of embedded real-time systems. *Formal Methods in System Design*, 44(3):203–239, 2014.

[RH91] Frédéric Rocheteau and Nicolas Halbwachs. Implementing reactive programs on circuits: A hardware implementation of LUSTRE. In *REX Workshop*, volume 600 of *Lecture Notes in Computer Science*, pages 195–208. Springer, 1991.

[RRS14] Thomas Reinbacher, Kristin Yvonne Rozier, and Johann Schumann. Temporal-logic based runtime observer pairs for system health management of real-time systems. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 357–372. Springer, 2014.

[Sch20] Torben Scheffel. *Expressiveness and Complexity of Stream-based Specification Languages.* PhD thesis, Universität zu Lübeck, 2020.

[SJN+17] Konstantin Selyunin, Stefan Jaksic, Thang Nguyen, Christian Reidl, Udo Hafner, Ezio Bartocci, Dejan Nickovic, and Radu Grosu. Runtime

monitoring with recovery of the SENT communication protocol. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*, pages 336–355. Springer, 2017.

[SKK+99] Oleg Sokolsky, Sampath Kannan, Moonjoo Kim, Insup Lee, and Mahesh Viswanathan. Steering of real-time systems based on monitoring and checking. In *WORDS (Fall)*, pages 11–18. IEEE Computer Society, 1999.

[SLD12] Aaron Spear, Markus Levy, and Mathieu Desnoyers. Using tracing to solve the multicore system debug problem. *Computer*, 45(12):60–64, 2012.

[SLG94] Viggo Stoltenberg-Hansen, Ingrid Lindström, and Edward R. Griffor. *Mathematical theory of domains*, volume 22 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, 1994.

[SMR15] Johann Schumann, Patrick Moosbrugger, and Kristin Y. Rozier. R2U2: monitoring and diagnosis of security threats for unmanned aerial systems. In Ezio Bartocci and Rupak Majumdar, editors, *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, volume 9333 of *Lecture Notes in Computer Science*, pages 233–249. Springer, 2015.

[SRU99] Jurij Silc, Borut Robic, and Theo Ungerer. *Processor architecture - from dataflow to superscalar and beyond*. Springer, 1999.

[Ste18] Rachel Stephens. The state of the time series database market. `https://redmonk.com/rstephens/2018/04/03/the-state-of-the-time-series-database-market/`, April 2018. [Online; accessed 10.03.2022].

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955.

[Tho68] Ken Thompson. Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.

[Vee86] Arthur H. Veen. Dataflow machine architecture. *ACM Comput. Surv.*, 18(4):365–396, December 1986.

[Ven98] Bill Venners. *Inside the Java Virtual Machine*. Mcgraw-Hill, 1998.

[Weia] Alexander Weiss. Event Processing. US 2021081145 A1, March 18, 2021.

[Weib] Alexander Weiss. Event Processing. EP 3792767 A1, March 17, 2021.

[WGJ+21] Alexander Weiss, Smitha Gautham, Athira Varma Jayakumar, Carl R. Elks, D. Richard Kuhn, Raghu N. Kacker, and Thomas B. Preußer. Understanding and fixing complex faults in embedded cyberphysical systems. *Computer*, 54(1):49–60, 2021.

[WLa] Alexander Weiss and Alexander Lange. Trace-Data Processing and Profiling Device. US 9286186 B2, March 15, 2016.

[WLb] Alexander Weiss and Alexander Lange. Trace-Data Processing and Profiling Device. EP 2873983 A1, May 20, 2015.

[XILa] XILINX. 7 Series FPGAs Configurable Logic Block: User Guide. UG474 (v1.8) September 27, 2016.

[XILb] XILINX. UltraScale Architecture Configurable Logic Block: User Guide. UG574 (v1.5) February 28, 2017.

[XILc] XILINX. Vivado Design Suite: AXI Reference Guide. UG1037 (v4.0) July 15, 2017.

[XILd] XILINX. Vivado Design Suite: FIFO Generator, LogiCORE IP Product Guide. PG057 (v13.1) April 5, 2017.

[XILe] XILINX. Vivado Design Suite User Guide: Synthesis. UG901 (v2019.1) June 12, 2019.

[ZJLS00] Jun Zhang, Karl Henrik Johansson, John Lygeros, and Shankar Sastry. Dynamical systems revisited: Hybrid systems with zeno executions. In Nancy A. Lynch and Bruce H. Krogh, editors, *Hybrid Systems: Computation and Control, Third International Workshop, HSCC 2000, Pittsburgh, PA, USA, March 23-25, 2000, Proceedings*, volume 1790 of *Lecture Notes in Computer Science*, pages 451–464. Springer, 2000.