Expressiveness and Complexity of Stream-based Specification Languages

Torben Scheffel

UNIVERSITÄT ZU LÜBECK
INSTITUTE OF SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

From the

Institute for Software Engineering and Programming Languages
of the University of Lübeck

Director: Prof. Dr. Martin Leucker

Expressiveness and Complexity of Stream-based Specification Languages

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

from the Department of Computer Sciences

Submitted by

Torben Scheffel
from Eutin

Lübeck, 2020

First referee: Prof. Dr. Martin Leucker
Second referee: Prof. Dr. Ezio Bartocci

Date of oral examination: 21st of May, 2021

Approved for printing: 25th of May, 2021

Acknowledgements

I got support from many people while writing this thesis, which helped me a lot and there are many ideas, talks and results from colleagues which influenced and advanced this thesis.

To begin with, I want to thank Martin Leucker, my supervisor, who gave me the possibility to write this thesis in the first place. He has helped me to get into research and get connected to many collaborators as well as giving me the freedom and environment to develop ideas and results, while still helping me with guidance over the course of this huge project. Additionally, I want to thank Ezio Bartocci for reviewing this thesis and Till Tantau for leading the examination board.

Furthermore, I am very grateful that I had my fellow colleagues around me, who always inspired me with ideas and with whom I was able to work on a variety of research questions and projects. First, I want to thank Malte Schmitz, with whom I studied and who always was on my side when academic questions or problems had to be discussed. Second, I want to thank Normann Decker and Daniel Thoma, who guided me with their experience in research and who were always open to fruitful discussions. Finally, I want to thank César Sánchez for his inspiration and thoughts regarding research this thesis is based upon.

I am also thankful to the ISP team in general, providing a supportive, helpful and guiding as well as fun and inspiring environment for research. Additionally, I want to thank Fernando Macías to be able to work with him on his PhD topic as well as a publication and the teams of CONIRAS and COEMS as well as the TeSSLa team in general, and all the people who contributed to the TeSSLa project.

Finally, I want to express my love to my parents and all my great friends, who supported me over all these years and without whom I would not have been able to write this thesis.

Thank you!

Abstract

This thesis is about results considering stream-based specification languages in terms of the Temporal Stream-based Specification Language (TeSSLa) regarding expressiveness, fragments, decision problems, and the comparison to other languages. At first, the standard definition of TeSSLa is extended by an operator for specifying future references, as it is also common in other stream languages. Then, TeSSLa as well as this extension by future references are syntactically restricted to obtain a subset of the language such that each formula always only has a unique fixed-point. Additionally, the expressiveness of the full language is considered as well as how the expressiveness changes if one or more operators are missing in the language.

Besides considering the expressiveness of the full language, the thesis also looks at various fragments which only consider input streams over bounded data domains and show their relation to different transducer types and complexity results for different decision problems for those fragments. We do not only consider standard decision problems, but also some regarding the memory usage during evaluation, which may be of interest for practical application. Among the fragments considered are some which allow some kind of unbounded data structures like a stack and some which consider real-time constraints as well as non-deterministic ones.

Lastly, TeSSLa is compared to other stream languages, in terms of expressiveness as well as also on a conceptual level. Besides the general stream model, we also consider a stream model restricted to discrete time steps of a given distance. To the languages compared with TeSSLa belong Esterel, Lustre, Striver, and LOLA and its extensions.

Zusammenfassung

Diese Arbeit befasst sich mit Ergebnissen bezüglich strombasierter Spezifikationssprachen im Bezug auf die Temporal Stream-based Specification Language (TeSSLa). Dabei werden Ausdrucksstärke, Fragmente, Entscheidungsprobleme sowie der Vergleich zu anderen Sprachen betrachtet. Zuerst wird die Standarddefinition von TeSSLa um einen Operator für Zukunftsreferenzen erweitert, wie es üblich ist für andere strombasierte Sprachen. Danach wird TeSSLa sowie diese Erweiterung um Zukunftsreferenzen syntaktisch so eingeschränkt, dass eine Teilmenge der Sprache entsteht, so dass jede Formel dieser Teilmenge nur genau einen Fixpunkt hat. Zusätzlich wird die Ausdrucksstärke der kompletten Sprache betrachtet sowie die Änderung an der Ausdrucksstärke, wenn man einen oder mehrere Operatoren der Sprache als nicht in der Sprache vorhanden betrachtet.

Außer der Betrachtung der Ausdrucksstärke der vollen Sprache, werden in dieser Arbeit verschiedenste Fragmente betrachtet, die nur Eingabeströme über endlichen Datendomänen annehmen. Es wird gezeigt, wie sich diese zu verschiedenen Arten von Transducern verhalten und es werden Ergebnisse bezüglich der Komplexität von verschiedenen Entscheidungsproblemen erlangt. In diesem Rahmen werden nicht nur in der Literatur typische Entscheidungsprobleme betrachtet, sondern auch Entscheidungsprobleme bezüglich dem Speicherverbrauch bei der Auswertung einer Formel, welche insbesondere für praktische Anwendungen interessant sein könnten. Unter den betrachteten Fragmenten sind auch welche, die eine bestimmte Art von unbeschränkten Datendomänen, wie Stacks oder Echtzeitbedingungen, erlauben, sowie nicht-deterministische.

Zuletzt wird TeSSLa noch mit anderen strombasierten Sprachen verglichen, sowohl bezüglich Ausdrucksstärke, aber auch auf einer konzeptionellen Ebene. Neben dem allgemeinen Strommodell wird auch ein Strommodell über diskrete Zeitschritte einer festen Distanz betrachtet. Zu den Sprachen, mit denen TeSSLa verglichen wird, gehören Esterel, Lustre, Striver und LOLA sowie dessen Erweiterungen.

Contents

1 Introduction

The correctness of software and hardware systems is a problem that always exists when building such systems. Failures occur all the time and some are worse than others. Some are related to hardware errors by construction, others to the programming of the hardware or to simple errors in the software and some may be delivered to the system under observation by external hardware, like a broken sensor.

In the past years various techniques evolved which go far beyond simple testing of the systems. The idea is to verify the correctness of the system under test. Testing can only show the absence of errors for the tested cases, but can neither show the absence of errors for the whole program, nor is it reliable to find at least the most crucial errors in a program with testing only.

One of the techniques that have been developed for this purpose during the past decades is model checking. In model checking, an abstraction of the full system is build to reflect every possible trace an execution can produce. Afterwards, this abstraction is checked against a specification of a correctness property given by an automaton or a temporal logic. If every possible execution of the system fulfils the specified property, the system fulfils this property. Otherwise an error in the system is found. The general problem of model checking is, that the whole system is looked at at once. This leads to the state space explosion problem when the system under test gets more complex. Of course various variations of classical model checking have been developed to overcome this problem, such that more complex systems can be verified using model checking. But especially for systems with inherent non-determinism like distributed or multi-core systems, the problem still persists.

Another growing technique for verifying the correctness of systems is runtime verification [HG05, LS09]. This approach checks the correctness of the system under test during the execution and detects if the system runs in a state that eventually leads to an error. For this purpose, the systems needs some sort of trace interface to get the data of the run out of the system. This data is then checked against a specification, generally given in the form of a temporal logic formula in the classical setting. There are two types of runtime verification: first, online runtime verification,

where the fulfilment of the correctness property is checked while the system is running to detect errors during the execution. And second, there is offline runtime verification, which means that the run of a system is stored in a file or database and later analyzed by checking it against a correctness property.

Two advantages of runtime verification are that first, it does not have the problem with the state space explosion because only the current execution is observed independently of the complexity of the system, and second, a real run of the system is considered, possibly also with environmental effects, while model checking only considered an abstraction of the systems behaviour and its environment. A disadvantage is that runtime verification is just less powerful: it can only observe if a system does violate a property during execution, it can not prove that a system fulfils a certain property.

While these advanced verification techniques have been a research topic of simple systems for quite some time and are slowly finding their way into practical usage and into research about more complex systems like multi-core ones, especially the power of what properties can be checked heavily depends on the language used to specify properties. In classical runtime verification approaches, temporal logics have been used and possibly multi-values monitors are created from the specification using different automata theoretic approaches [HR02, BLS11]. Therefore, because the monitor essentially only states fulfilment, violation or some uncertainty in between, only correctness properties are able to be checked with such approaches. But there would also be a benefit in being able to do arbitrary calculations on data values or measuring timing behaviours of the system during the execution, which desires for a more powerful specification language.

The first language for specifying system behaviour was the Linear Temporal Logic (LTL, [Pnu77]) which is able to specify temporal behaviour of a system. Later, is was extended in different ways, for example by real-time constraints with the Timed Linear Temporal Logic (TLTL, [Ras99]) and the Metric Temporal Logic (MTL, [Koy90]). But both were only able to handle boolean propositions as input and lack the ability to handle richer data values and more advanced timing behaviour, even though the idea of handling richer data has been introduced in the Signal Temporal Logic (STL, [MN04]). Additionally, temporal logics are build to make one final statement about fulfilment or violation of the property, but lack the ability to make more distinct statements regarding values or timing behaviour of the input. Languages which are able to handle such things are stream languages. Typical stream languages are Lustre [CPHP87, HCRP91] and Esterel [Ber92, Ber99, Ber00, Ber04], but those are programming languages and not specifically tailored to the specification of properties of a system. Compared to programming languages, specification languages often do not care if they can be evaluated easily, but instead aim to deliver

a way to easily specify the property someone wants to check. Therefore, specification languages may be able to make statements about future behaviour and consider future events for the current evaluation or are able to specify behaviour which leads to an infinite amount of events in a finite timespan. Additionally, specification languages are created in a way such that they give an output for every input and, compared to temporal logics, do not have semantics which can possibly only make a statement in infinity. Such statements as *there has always to occur another a in the future*, which are called *liveness properties*, can not be specified with such specification languages, because they practically make no sense, as they can never be checked completely in a running system. Nevertheless are such specification languages, as programming languages, often Turing complete in general, because they get their expressive power from their more powerful operators and are able to make every calculation a Turing machine is able to do over a finite amount of time.

In the last years, the first stream specification languages have been developed, pioneered by LOLA [DSS+05]. As LOLA only has a restricted view on possible system behaviour, like that it has no explicit notion of time and that all inputs have to be synchronized, we developed a new language called TeSSLa [CHL+18] to extend the view on the system by an explicit notion of time and a more flexible, non-synchronized view on the systems behaviour.

Before we go more into detail, we present a taxonomy for specification languages and fix the wording. Different taxonomies in the field of runtime verification have already been developed in [FKRT18]. The paper presents six taxonomies, all for different areas in this field. While one covers runtime verification in general, the others cover most of the subfields in more depth, as four others are taking a closer look at monitors, traces the monitors get as input, the deployment of monitors and possible reactions when a violation is found, while the last one is considering specifications from which the monitors are build. While we also take a look at specifications, their taxonomy is tailored to the generic model of what and how a specification can be, while our taxonomy in this thesis focusses on the aspects of the languages and not the general area of specifications itself. Even though there is some overlap, we also added the focus on streaming behaviour.

Our taxonomy for specification languages is depicted in Figure 1.1, which shows the four dimensions of such languages. The green part is the dimension about *time*. Time in a specification language can either be *implicit*, which means the language has no specific notion of time and can not handle the timestamps of incoming data, or *explicit*, which means that the language has a view and can do calculations on the timestamps of the input values. An explicit notion of time can either be *discrete*, therefore, the time domain has a certain distance between each element, for example $\mathbb{N}$, or *continuous*, which means that the time domain has no gaps between its elements, for example $\mathbb{R}$. In the continuous case, one can also differentiate if *Zeno* behaviour is allowed in the timestamps of

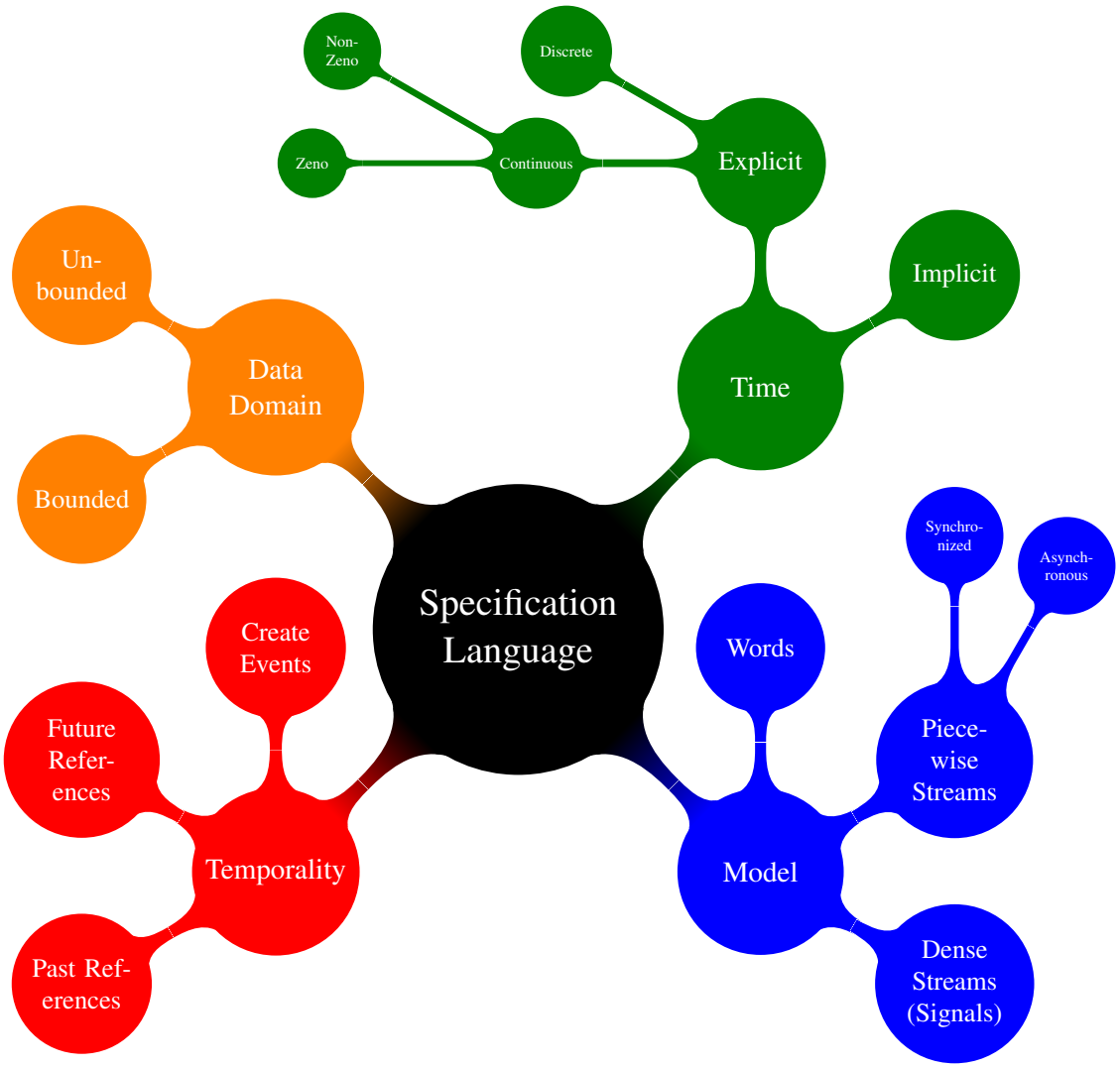


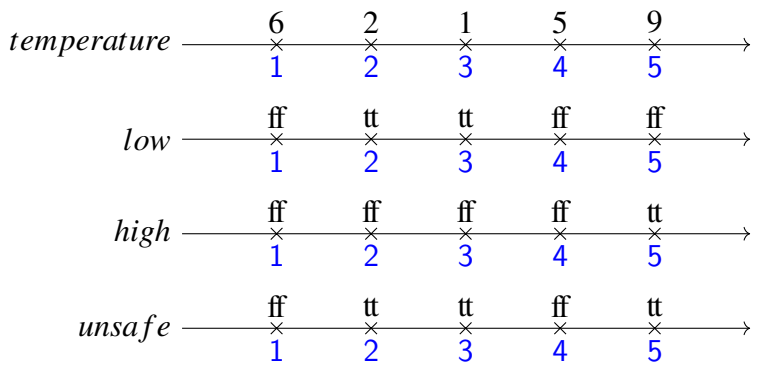
Figure 1.1: A taxonomy for specification languages. It shows the four dimensions, Time (green), Model (blue), Temporality (red) and Data Domain (orange), in which specification languages may differ.

incoming events. If Zeno behaviour is allowed, an infinite number of timestamps of values can occur in a finite timespan and timestamps of values can, for example, converge to a certain timestamp but never reach it, while otherwise, such behaviour would not be allowed. The second dimension is the blue part about how the inputs are *modelled*. These can either be *words*, which are sequences of values like it is in LTL, *dense streams*, which means that the input is a function, mapping a time domain to a value from a data domain such that, at every timestamp, there can be a different value as it is in a sinus curve. The third option are *piece-wise streams*, which are also a function mapping timestamps to values from a data domain, but guarantee that the values only change at concrete timestamps, which means they can be represented as sequences of *events*. Such a stream model can also be either *synchronized* which denotes that every stream has a value at some timestamp or no stream has a value, or *asynchronous*, which means that values can occur at arbitrary timestamps on a stream, independent from the other streams. The third dimension in red is the *temporality*. A specification language can have *past references* and therefore reference events and their values that occurred at past timestamps, *future references* and therefore reference events and their values that will occur at future timestamps or being able to *create events* at timestamps where no events occurred on the input streams, which means it can reference timestamps even though no incoming event exists at this timestamp. The last dimension, the one in orange, is the *data domain* of the values of the events. A data domain can be either bounded or unbounded. If it is *unbounded*, it can be any data domain one can imagine, if it is *bounded*, it is only allowed to have a finite set of finite values. For a specification language, this is equivalent to only allowing boolean values (even though a richer domain may be easier for the user) for the purposes on this thesis, as every finite data domain can be encoded in boolean and vice versa. Therefore, we use the terms *bounded data domain* and *boolean data domain* interchangeably in this thesis.

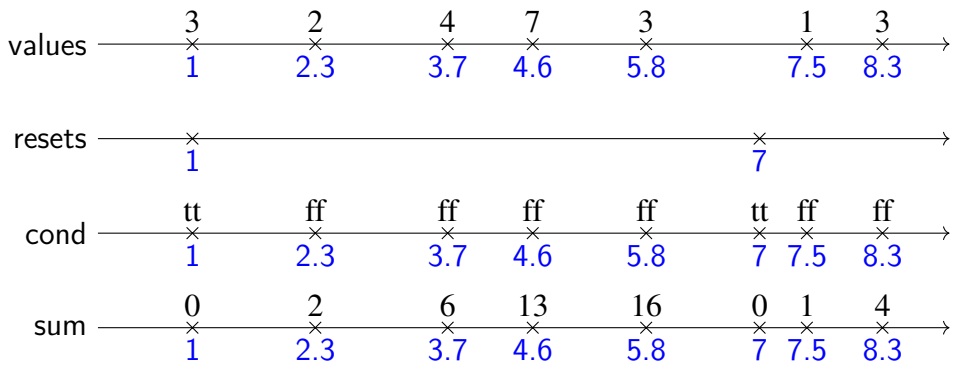
While, as stated before, in the classical approaches mostly by that time well known logics have been used and some logics are also able to handle data by some amount, a language which is able to handle richer data domains naturally and can do arbitrary calculations on those data is required to enhance these verification techniques. Therefore, recently the development of stream runtime verification (SRV, [DSS$^+$05, BS14, CHL$^+$18]) has been pushed forward, using specification languages as mentioned before, especially made for specifying properties to analyse system behaviour, among which are the logic hybrid STL [MN04] and the two stream specification languages LOLA [DSS$^+$05] and TeSSLa [CHL$^+$18]. In SRV and all such languages, the representation of the execution of a system has been changed from words, which contain atomic propositions that are either true or false, to streams, which are functions mapping timestamps to values or denote, that no event occurred at a timestamp and contain values at arbitrary timestamps which are often used or referenced over time by the specification language, where these values can be integers or

from more complex data domains like sets, queues, or maps or anything else. SRV is the sub field of research of runtime verification using a stream based model of computation for the specification languages and the resulting monitors, for which the mentioned specification languages like STL, LOLA and TeSSLa can be used. As an example for SRV, consider the following specification over piece-wise constant streams which checks whether a measured temperature stays within given boundaries [CHL+18]. For every new event (measurement) on the temperature stream, the new events on the derived streams *low*, *high* and *unsafe* are computed:

$$low := temperature < 3$$
$$high := temperature > 8$$
$$unsafe := low \vee high$$



This example from [LSS+19] uses a synchronized stream model, all streams have events at the same timestamps. Furthermore, the model is discrete time wise, because the timestamps are placed on a predetermined grid. Languages like LOLA, or also the programming languages Lustre [CPHP87] and Esterel [Ber92], are limited to such a stream model. TeSSLa requires the events of all streams to be in a global order, but does not require all streams to have simultaneous events, thus, it uses an asynchronous stream model. As a consequence, there does not need to be a certain distance between two events, but instead, streams with events at arbitrary timestamps can be modelled, therefore, even if the frequency at which the events occur changes arbitrarily. As cyber-physical systems often give rise to streams at unstable frequencies or in continuous fashion, this asynchronous setting is especially suitable for such systems [CHL+18]. An example for SRV with an asynchronous stream model can be seen in the following streams:



This example has two input streams, values (with numeric values) and resets (with no internal value). The intention of the specification is to accumulate in the output stream sum all values since the last reset. The intermediate stream cond is derived from the input streams indicating if reset has currently the most recent event, and thus the sum should be reset to 0.

In cyber-physical systems, there is often another view on such streams. In such systems, streams are often called signals, which are dense streams, and are looked at as them having values continuously and not just events at certain timestamps. In general such signals are discretized for calculations at least in the clock speed of the processor. We do not consider such a stream model of continuous values directly in this thesis, but stream specification languages allow this view in parts by using their operators for past references in an event-based stream model, being able to use prior values as if they are still active on the stream. Still, there can be streams like a one representing a sinus curve which not only have values continuously, but also ever changing values in a finite time span, hence, the values are not piece-wise constant. These are also streams which can typically be emitted by a cyber-physical system. We do not consider such streams at all in this thesis. Even though TeSSLa is able to handle such streams, this is a topic for another work and we focus on TeSSLa over piece-wise constant streams.

TeSSLa has been developed exactly for this purpose, being able to handle arbitrary types of streams and values, handle time explicitly, and even being able to do calculations on continuous valued streams like a sinus curve. Even though, it is build to still delivering easy to check memory guarantees when having bounded data structures as well as compositionality of its operators to allow efficient hardware implementations [CHL+18].

A disadvantage of such rich languages is that they are harder to evaluate than a logic for which one can build a monitor in form of an automaton and just execute this automaton, like for example in LTL and its three-valued semantics. Executing an automaton during runtime is especially easy, for the current state and the input, one has to just look up the following state in a table. Therefore,

evaluating LOLA or TeSSLa is much more difficult, as no easy way for creating an automaton for a formula exist. This is mostly because of the richer data domains, but also because the way some operators work. This means it is important to find new ways to evaluate such formulas, like using parallelized software implementations and leads to the question which types of solutions are feasible for this problem.

While there has also been done some work on parallel software implementations for such languages as, for example, for TeSSLa in [LSS+18, LSS+20], recently more and more approaches were brought up to create monitors on hardware, like on FPGAs [DGH+17, DDG+18, CHS+18, RRS14, JBG+15, NBN+16, HR02]. These approaches on FPGAs were used to be able to plug the FPGA containing the monitor directly into the system under observation. Even though FPGA have a low frequency compared to conventional processors, they are made for parallelized evaluations. As long as the single operators of the language are compositional, which holds for TeSSLa and LOLA but not for classical temporal logics, one could implement the formula as a network of single units, one type for each operators, and connect those as they are nested in the formula. It remains to show if the evaluation of a stream languages specification on an FPGA may also lead to performance increases or not. Also, an important question for FPGA implementations is the one of memory guarantees a language can deliver, because FPGA do not have such a huge amount of space.

While the practical problems with a language like TeSSLa explained before are an important point of research, this thesis focusses on the theory behind TeSSLa and stream languages in general. It is important to consider expressiveness, fragments and complexities for decision problems of such languages to be able to classify them and to see how they relate to each other as well as to get a deeper, theoretical understanding. This will be explained in the following section in more detail.

1.1 Contribution

The topic of research of this thesis is TeSSLa, a stream specification language build for specifying properties to find errors in systems, track certain timing behaviour or simply do arbitrary calculations of values arising from the system. TeSSLa and preliminary versions of it have already been studied in various papers [CHL+18, LSS+18, LSS+19, DGH+17, DDG+18, LSS+20]. While the focus in [DGH+17, DDG+18] was more on the practical implementation of TeSSLa specifications on FPGAs or similar hardware, [LSS+18, LSS+20] are focussed on parallel software implementations. In all four papers, a static fragment of TeSSLa has been used with fixed data calculation

functions and without allowing recursion to make first attempts in these directions better handable. In [CHL+18], the full version of TeSSLa has been introduced as well as many results on the types of properties TeSSLa can express, how the relation of certain fragments to transducers looks like, and what the complexities for different decision problems of theses fragments are. On the other hand, it has been shown in [LSS+19] how TeSSLa can be applied to traces with uncertainties, where maybe values and events are unknown at certain timestamps. In this thesis, we focus on the results from [CHL+18] and use those as a basis. From there on, we extend the results to more fragments and statements about complexities, a more concrete comparison to other languages, and look at a possible extension of TeSSLa.

Concretely, the contribution of this thesis is fourfold:

- First, we take a deep look into TeSSLa, consider various fragments, some already considered in [CHL+18] and many new ones, and the complexities of different decision problems for those as well as what kind of properties the full TeSSLa language can express.

- Second, we introduce a new operator compared to TeSSLa as defined in [CHL+18], which allows future references and consider which new fragments and possibilities such an operator adds to TeSSLa.

- Third, we define decision problems based on memory usage and consider the memory usage as well as memory guarantees which TeSSLa and its fragments can deliver. We use the term evaluation strategies to denote an algorithm to evaluate a TeSSLa formula and compare an naive approach to evaluate a formula to the optimal one. Additionally, the memory properties of both approaches are compared.

- Fourth, we consider a method for comparing TeSSLa and stream languages in general to other stream languages, considering expressiveness of these languages and various fragments as well as comparing them on a conceptual level. This method allows us to distinguish the languages and fragments by expressiveness, even though many of them are Turing complete. While we take a look at less closely related languages in the following related work section, we do a formal comparison to closely related languages later.

1.2 Related Work

TeSSLa [CHL+18] is originally designed in the scope of runtime verification, but it also allows for arbitrary evaluation of system properties or any calculation on data it gets as well as analyses, and is not limited to the usage for verification purposes. Originally, logics have been used to specify correctness properties in runtime verification. Because one wants to express properties that consider the changes and actions over time propositional logic was not working in this case. Therefore, regular expressions or temporal logics like the Linear Temporal Logic (LTL, [Pnu77]), the regular extension of LTL, the Regular Linear Temporal Logic (RLTL, [LS07]) or, when considering real-time properties logics like the Timed Linear Temporal Logic (TLTL, [Ras99]), Event-Clock Temporal Logic (ECTL, [RS97]), Timed Regular Expressions (TRE, [ACM02]) or the Metric Temporal Logic (MTL, [Koy90]) and its fragments like Metric Interval Temporal Logic (MITL) [AFH91, AH90], have been developed. Compared to languages used in stream runtime verification, like TeSSLa, those temporal logics or regular expressions all lack important features: Their data domains are limited to atomic propositions and their calculations are limited to output fulfilment or violation of the formula, but nothing else. TeSSLa on the other hand is able to do arbitrary calculations on arbitrary data domains and is even able to output values depending on parts of the input.

Nevertheless, there are logics that can handle richer data domains, for example the Monitoring Module Theories (MMT) approach defined in [DLT16], which extends temporal logics like LTL by data values and allows more complex calculations on data values. The approach results in the Temporal Data Logic (TDL), which replaces the atomic propositions from LTL with data calculations and comparisons. Still, TDL lacks the temporal and timing capabilities of stream runtime verification as it is not able to handle time explicitly, as well as the correspondence between input and output values at different timestamps.

Compared to the MMT approach, STL is a logic defined for handling streams and not just values. While it can handle streams as input, the only way to use values from streams is to discretize the streams by applying comparing functions to their values. These functions are then outputting boolean values that are later used as propositions for the MITL formula inside an STL specification and output a fulfilment or a violation of this MITL formula in the end, hence, STL can only map streams to one final verdict. The same holds for Quantitative Regular Expressions (QRE, [AFR16]) and Time Frequency Logic (TFL, [DMB+12]), as both are only able to handle more complex calculations on data values, but are still bound to the restrictions of temporal logic. Therefore, no calculations on the timestamps are possible and the approaches can only output fulfilment or

violation, but no values which, for example, depict the timing behaviour of the system or give some more explicit hints at the behaviour of the system, which is possible in other stream runtime verification languages like LOLA or TeSSLa.

TeSSLa itself is heavily based on and influenced by Functional Reactive Programming (FRP, [EH97]). FRP is defined over sequences of events. It has an explicit notion of time, which returns the current timestamp and allows the inclusion of timestamps into calculations. Furthermore, it allows the lifting of arbitrary functions on values to functions on event sequences, therefore, it has an operator lift which takes a function on values and returns a function on event sequences. By its timeTransform operator it is also able to transform time and output values at timestamps which did not occur in the input. TeSSLa also consists of those three concepts, extended them to the more general model of streams, not only sequences of events, and adjusts them to the specification of properties. In the end, TeSSLa implements the ideas from FRP into a stream specification framework. If one interprets an input sequence of events for a FRP formula as a piece-wise stream, TeSSLa can do everything FRP could do.

Besides the logics already mentioned, there are some stream languages besides TeSSLa which take a set of input streams and output a set of output streams and allow similar evaluations and aggregations as TeSSLa to produce the output streams from the input streams. Besides the three synchronous stream languages Esterel [Ber92, Ber99, Ber00, Ber04], Signal [LBBG86, GL87] and Lustre [CPHP87, HCRP91], which are more language designed for typical programming tasks and not tailored for the analysis of a system, there is also LOLA [DSS$^+$05, FFST16], which is, like TeSSLa, a stream language designed for analysing a systems behaviour. In contrast to TeSSLa, all those languages are using a synchronized stream model as underlying basis, which means that all input streams have events at the same timestamp and that there is a certain predetermined grid on which the timestamps are placed, as well as an implicit notion of time. Therefore, they are missing the capabilities and the flexibility to operate on the more complex asynchronous stream model, meaning that events can occur at arbitrary timestamps and that events occur on an input stream independently from the other input streams. For example, such languages are missing the possibility of acting on timestamps where the input streams do not have events, or handle arbitrary timestamps which are not on the grid. Additionally due to the implicit handling of time, they are not able to do calculations or comparisons regarding timestamps, which takes away many interesting properties on an asynchronous stream model. TeSSLa is able to do both, handle such an asynchronous stream model as well as handling timestamps explicitly. For LOLA, a more complex analysis of its properties and the properties of a fragment restricted to boolean values has been done in [BS14]. Similar to the stream languages mentioned before, there is Copilot [PGMN10], which

is a synchronous stream languages embedded as a DSL in the functional programming language Haskell. Copilot is heavily influenced by Lustre and LOLA and faces the same drawbacks as those languages. Another language in the field of synchronous stream languages is Focus [BS01]. While Focus can distinguish between three types of streams, all of the types are synchronous over the discrete time domain $\mathbb{N}$ and mainly differ in the addition of integer based timestamps to the events or special symbols for having no event at some time instant on some stream. On its most powerful stream model, it can access the timestamps explicitly, but because those are only integers, it does not increase the capabilities of the language compared to, for example, LOLA, because one could simply count the time by the number of events. Therefore, Focus also suffers the same drawbacks as LOLA does, compared to TeSSLa.

Compared to those languages, there are also two that are naturally defined on an asynchronous stream model, RTLola [FFST19, FFS$^+$19] and Striver [GS18, GS20]. While RTLola only adds a more flexible grid to the stream model as well as the ability of doing calculations in sliding windows, it does not have the possibility of adding events where the chosen grid does not have a timestamp. Striver on the other hand has all those capabilities, but permits streams with an infinite number of events in a finite time span (called Zeno streams) in general, while TeSSLa is able to handle such streams naturally. In the end, both languages are quite similar feature wise, but even though we do not consider streams like a sinus in this thesis, TeSSLa would be able to handle such streams that may occur in cyber-physical systems without a discretization, while Striver is not.

Besides the stream languages mentioned already, there is BeepBeep [HV09], currently available in its third version BeepBeep 3 [HK17, HK18], which is a stream query language for complex event stream processing, embedded as a DSL in Java. It does not contain one specific query language, but instead lets you write Java code as a specification. BeepBeep uses a queue-based approach, consuming events as soon as they arrive in an input queue without considering the timestamps at which the events were emitted. Therefore, BeepBeep has a completely unsynchronized event handling. If someone wants to extend BeepBeep by timestamps, there would be the problem that it is not build to keep their order and one would have to solve this problem as well. TeSSLa on the other hand, while operating on an asynchronous stream model, does retain a certain amount of synchronization, processing events for a given timestamp only when it knows that on all streams, either an event has arrived or it is known that no event will arrive and then process this timestamp at all streams at once. This does allow for keeping the time-wise order on the events, which is lost in BeepBeep and delivers the advantage for TeSSLa of being able to react to the full knowledge one can get at a certain timestamp, which is not possible in BeepBeep, as it processes events before all events at the given timestamp have arrived. Additionally, TeSSLa does handle time explicitly,

while BeepBeep has no notion of time.

Another concept related to stream analysis are Time Series Databases (TSDB, [DMF12]) for which different implementations exist, like influxDB[1] using InfluxQL as query language or OpenTSDB[2] using an unnamed query language. Additionally, there are the Continuous Query Language (CQL, [ABW06]), a framework to extend relational query languages to streams, as well as PipelineDB[3], which is designed to run and evaluate SQL queries continuously on streams. While such time series can also be seen as streams, the concept of TSDBs is quite different from TeSSLa's. TSDBs are build to store huge amounts of data incoming on the streams and then do calculations on those specified in query language. Neither the database nor the query language is constructed in a way such that data can be analysed on the fly, but instead it is assumed that one stores all the values and afterwards has access to the needed information and queries what is needed. Therefore, all those languages are designed to do and focus on calculations on sliding windows over the data. Compared to TeSSLa, they are only able to use time to query values, not to do calculations on the time itself or to modify it. Therefore, they can only act at statically given timestamps which determine the size of the window, not on dynamically generated timestamps during evaluation of the specification. Additionally, they are optimized for database queries and their approach to specify is quite different to the one from TeSSLa, as they can only do calculations on values based on sliding windows. While this is a well working approach for data base queries, it makes it harder to specify certain properties as one could to with a more general specification language like TeSSLa.

A more throughout discussion and comparison of the stream languages which are closely related to TeSSLa is included later in this thesis, where we also present a method how stream languages can be compared formally in terms of expressiveness, even though most of these languages are Turing complete. While it relates to the problem of how Interactive (sometimes called Persistent) Turing Machines relate to normal Turing machines [Weg98, GSW01, Gol00, GSAS04], besides the categorization of certain types of properties in [CHL+18], to the best of our knowledge, no work has been done on the question of how to adapt this discussion to the comparison of different stream languages.

[1] `https://www.influxdata.com/`
[2] `http://opentsdb.net/`
[3] `https://www.pipelinedb.com/`

1.3 Overview

This thesis is structured in six chapters, which span over into introductory chapters, chapter two and three, followed by three chapters containing the main results, as well as a conclusion. In more detail, it is structured as follows:

- In Chapter 2, we present an introduction to all concepts and basic notation we use in this thesis. This includes but is not limited to various types of logics and automata, streams and stream languages, transducers, memory usage and different decision problems.
- In Chapter 3, we introduce the specification language TeSSLa which is the main focus of research in this thesis. Besides the definition of syntax and semantics from [CHL+18], we additionally present a semantics on total streams and an extension of TeSSLa by future references.
- In Chapter 4, we present different language theoretic results on TeSSLa and various fragments of it, mainly by leaving out operators or extending the language with operators. We classify those variants of TeSSLa by the set of stream transformations they can express. Furthermore, we give a notion of well-formed specifications by syntactically restricting TeSSLa, which only has one fixed-point.
- In Chapter 5, we identify and define various interesting TeSSLa fragments and show their relation to different types of transducers. Furthermore, we receive additional results on complexities for decision problems like equivalence as well as memory related ones.
- In Chapter 6, we show how TeSSLa relates to stream languages on continuous streams as well as how TeSSLa relates to stream languages naturally defined over discrete streams when we restrict the stream model to discrete ones. Various variants of LOLA, Lustre, Esterel and Striver are considered in this chapter.
- In Chapter 7, we conclude the results of this thesis and state possible extensions and future works regarding TeSSLa.

2 Preliminaries

Contents

This chapter covers well-known formalisms as well as basic notation we will use in the rest on the thesis. This includes logics like LTL, MTL, MITL or STL, the specification language LOLA, various types of automata and transducers as well as complexity classes but also a classification of certain properties for such formalisms or the formal definition of streams we use. We start by giving some basic notation, followed by the definitions of the classical logics and types of automata and after that, we proceed with streaming semantics, the specification language LOLA as well as the different types of transducers.

2.1 Basic Notation

We give several basic notations and conventions in this section which we will use throughout this work. This includes sets of numbers, intervals, (timed) words, and more.

We denote the set of natural numbers with $\mathbb{N}$, the set of integers with $\mathbb{Z}$ and the set of real numbers with $\mathbb{R}$. We assume 0 is included in these sets.

We write $\mathbb{T}$ to denote an arbitrary *time domain*, which will be either $\mathbb{N}$ or $\mathbb{R}$ in this thesis and denote with $\mathbb{T}_\infty$ the inclusion of ∞ in the time domain, therefore $\mathbb{T}_\infty = \mathbb{T} \cup \{\infty\}$ with $\forall t \in \mathbb{T} : t < \infty$. A time domain only requires a total order and corresponding arithmetic operators. We call the time domain *discrete* iff $\mathbb{T} = \mathbb{N}$ and *continuous* iff $\mathbb{T} = \mathbb{R}$.

In general, a time domain only needs to be a totally ordered semi-ring $(\mathbb{T}, 0, 1, +, \cdot, \leq)$ that is not negative, i.e. $\forall t \in \mathbb{T} : 0 \leq t$, as defined in [CHL+18]. Compared to a ring, a semi-ring does not need to have inverse values for the addition, which suits our purpose perfectly because we do not want time domains to be negative. Additionally, our semi-ring needs to be totally ordered, because we need a strict order and progress in time, which also fits to our purpose of using $\mathbb{N}$ or $\mathbb{R}$ as time domains.

We need $\mathbb{T}_\infty$, and therefore the addition of an infinity element, to later represent the case that nothing happens any more after a certain timestamp. In this case, ∞ is an element which is not already contained in the time domain and greater than every other element. Even though a time domain can be a finite set, we only consider time domains with an infinite number of elements which are strictly growing, like the two above mentioned sets of numbers $\mathbb{N}$ and $\mathbb{R}$, in this thesis.

We call a set of values, which we use to do calculations on that are not considering time, a *data domain*, to clearly distinguish it from a time domain. We denote with $\mathbb{U} = \{\square\}$ the data domain only containing the unit type.

An *interval* is a subset of a certain set of numbers, which contains all elements of the original set between two given boarders. We write $\mathbb{I}_X = \{[a,b] \mid n \in [a,b] \Leftrightarrow n \in X \wedge a \leq n \leq b\}$ where $X \in \{\mathbb{N}, \mathbb{R}\}$ for the sets of intervals over certain data domains. We call an interval $[a,b]$ *punctual* iff $a = b$.

We write $\mathbb{B} = \{\mathrm{tt}, \mathrm{ff}\}$ for the boolean domain, where the two values tt and ff indicate true and false, respectively.

Let S be a set. We denote by S^* the set of finite sequences of elements of S, and by S^ω the set of infinite sequences of elements of S and by $S^\infty = S^* \cup S^\omega$ the set of all sequences of elements of S.

We call a finite, non-empty set an *alphabet*. A *word* is a sequence over an alphabet Σ. A *finite word* w is a finite sequence $w = w_0 w_1 \dots w_n \in \Sigma^*$ and an *infinite word* is an infinite sequence $w = w_0 w_1 w_2 \dots \in \Sigma^\omega$. A *timed word* is a sequence of tuples $w = (w_0,t_0)(w_1,t_1)\dots(w_n,t_n) \in (\Sigma \times \mathbb{T})^*$, or $w = (w_0,t_0)(w_1,t_1)(w_2,t_2)\dots \in (\Sigma \times \mathbb{T})^\omega$ respectively, where $\forall 0 \leq i < n : t_i < t_{i+1}$. One could define timed words with $t_i \leq t_{i+1}$, but this would lead to more complicated semantics for some formalisms which are based on those, because it can not be assumed any more that a timestamp has really been surpassed when it occurs, because it can occur multiple times in a row. On the other hand, this would not deliver any advantage, because the same information can be encoded on a single position of the word.

Given a partial order $(A, \leq)$, a set $D \subseteq A$ is called *directed* if $\forall a,b \in D : a \leq b \vee b \leq a$. A partial order $(A, \leq)$ is called *directed-complete partial order (dcpo)* if a supremum $\bigvee D$ exists for every directed subset $D \subseteq A$.

We call a stream language, a logic, a class of automata, or a class of transducers a *specification formalism*, or just *formalism* for short. A concrete formula, automaton or transducer of a certain formalism F is called an *instance* of F.

We call a function or a set of words a *language*. For an instance φ of a specification formalism, we write $\mathcal{L}(\varphi)$ to denote the language of the instance φ, thus the set of words the instance accepts, if it has an acceptance condition (instances of logics or automata), or the set of input/output tuples the function, which maps the input to an output, represents, in case of an input/output relation (instances of transducers or stream languages).

We denote complexity classes in the following way: We write LINTIME, PTIME and EXPTIME for linear time, polynomial time or exponential time, respectively, as well as PSPACE and EXPSPACE for polynomial or exponential space.

Lastly in this section, we give the notion of expressiveness which we use for the rest of this thesis to compare different formalisms.

Definition 2.1 (*Expressiveness*)

Let F and F' be two specification formalisms. We say F is at least as expressive *as F', denoted by $F' \subseteq F$ iff for every instance i' of F' an instance i of F exists, such that*

$$\mathcal{L}(i) = \mathcal{L}(i')$$

We say F and F' are equally expressive, *denoted by $F' = F$, iff $F' \subseteq F$ and $F \subseteq F'$.*

We say F is more expressive *than F', denoted by $F' \subset F$, iff $F' \subseteq F$ and not $F \subseteq F'$.*

We say F and F' are incomparable *iff neither $F' \subseteq F$ nor $F \subseteq F'$.*

Since formalisms represent a set of languages, we use the set inclusion symbols in this thesis to denote relations regarding expressiveness.

2.2 Functions and Fixed Points

In this section we will start with defining two important properties of functions before we present the fixed-point theorem from Kleene. Those properties are monotonicity and continuity of functions, which is also called Scott-Continuous [Vic89]. It is also important to note that a continuous function as defined here is always also monotonic.

Definition 2.2 (*Monotonic Functions, [Vic89]*)

Let $f \in A \to B$ be a function and $(A, \leq)$, $(B, \leq')$ partial orders. f is called monotonic *iff*

$$\forall a_1, a_2 \in A : a_1 \leq a_2 \Rightarrow f(a_1) \leq' f(a_2)$$

Informally, a function is called monotonic if it preserves the order.

Definition 2.3 (*Continuous Functions, [Vic89]*)

Let $f \in A \to B$ be a function and $(A, \leq)$, $(B, \leq')$ partial orders. f is called continuous *iff*

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

for all directed subsets $D \subseteq A$.

Informally, a function is called continuous if it preserves the supremum.

Next, we will state the Kleene fixed-point theorem which is used for proofs later in this thesis .

Theorem 2.4 (*Kleene fixed-point theorem, [Tar55, SLG94]*)

Let $(L, \leq)$ be a dcpo with a least element $\bot$ and let $f : L \to L$ be a monotonic and continuous function. The ascending Kleene chain of f is the chain

$$\bot \leq f(\bot) \leq \cdots \leq f^n(\bot) \leq \ldots$$

Then f has a least fixed-point, which is the supremum of the ascending Kleene chain of f.

By the Kleene fixed-point theorem, every monotonic and continuous function $f : A \to A$ has a least fixed point $\mu(f)$ if $(A, \leq)$ is a dcpo with a least element $\bot$. $\mu(f)$ is the least upper bound of the chain iterating f, starting with the bottom element: $\mu(f) = \bigvee\{f^n(\bot) \mid n \in \mathbb{N}\}$.

2.3 Logics

We define different well known logics in this section, namely the Linear-time Temporal Logic (LTL), the Metric Temporal Logic (MTL), the Metric Interval Temporal Logic (MITL) and Signal Temporal Logic (STL) in this section. LTL, MTL, and MITL are central to many verification techniques and, even though we do not use them later formally, it is important to know how those work in detail to understand the difference between classical temporal logics and our streaming approach. Additionally, STL is a logic which is based on streams, which makes it an interesting formalism to consider later in this thesis.

2.3.1 Linear-Time Temporal Logic

The Linear-Time Temporal Logic (LTL, [Pnu77]) allows for specifying properties over the sequential order of system states. In the following, let $\Sigma = 2^{AP}$ be an alphabet and AP be the set of atomic propositions.

Definition 2.5 (*LTL syntax, [Pnu77]*)

Let $p \in AP$ be an atomic proposition. The syntax of an LTL formula φ is given by the following grammar:

$$\varphi := tt \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \varphi \mathcal{U} \varphi$$

Given an infinite word in Σ^ω, the semantics of an LTL formula determines if the word fulfils the formula or not. The semantics is given in form of a relation $\models$, as described in the following definition.

Definition 2.6 (*LTL semantics, [Pnu77]*)

Let $w = a_0 a_1 a_2 \cdots \in \Sigma^\omega$ be an infinite word, $i \in \mathbb{N}$ an index, and φ and ψ LTL-formulas. Then the semantics for each LTL formula is defined inductively as follows:

$$\begin{aligned}
&w, i \models tt \\
&w, i \models p && \Leftrightarrow && p \in a_i \\
&w, i \models \neg\varphi && \Leftrightarrow && w, i \not\models \varphi \\
&w, i \models \varphi \vee \psi && \Leftrightarrow && w, i \models \varphi \text{ or } w, i \models \psi \\
&w, i \models \bigcirc\varphi && \Leftrightarrow && w, i+1 \models \varphi \\
&w, i \models \varphi \mathcal{U} \psi && \Leftrightarrow && \exists j \geq i : w, j \models \psi \wedge \forall i \leq k < j : w, k \models \varphi
\end{aligned}$$

We say that a word w is a model for an LTL-formula φ if $w, 0 \models \varphi$. Each LTL formula φ defines a language which we denote as $\mathcal{L}(\varphi) = \{w \in \Sigma^\omega \mid w, 0 \models \varphi\}$.

The LTL semantics relates formulas to words that fulfil the formula. All in all, an LTL formula represents the language of words that fulfil the formula. As stated before, only properties over the sequential order of events (propositions) can be specified with LTL, there is no explicit notion of time or different data domains.

2.3.2 Metric Temporal Logic

The Metric Temporal Logic (MTL, [Koy90]) extends LTL by a notion of time such that also real-time properties can be specified. Let again in the following $\Sigma = 2^{AP}$ be an alphabet, where AP is again the set of atomic propositions, and let $\mathbb{T}$ be a time domain. Then the syntax of MTL is given as follows:

Definition 2.7 (*MTL syntax, [Koy90]*)

Let $p \in AP$ be an atomic proposition, $\mathbb{T}$ be a time domain, and let $I \in \mathbb{I}_{\mathbb{T}_\infty}$ be an interval. Then the syntax of an MTL-formula φ is given by the following grammar:

$$\varphi := tt \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \varphi \mathcal{U}_I \varphi$$

In contrast to LTL, the semantics of MTL now determines for a given timed word if this timed word fulfils the formula or not. By doing this, using the until operator ($\mathcal{U}$) now parameterized with an interval, one can specify properties with real-time constraints.

Definition 2.8 (*MTL semantics, [Koy90]*)

Let $\mathbb{T}$ be a time domain, $w = (a_0,t_0)(a_1,t_1)(a_2,t_2)\cdots \in (\Sigma \times \mathbb{T})^\omega$ be an infinite timed word, $I \in \mathbb{I}_{\mathbb{T}_\infty}$ be an interval, $i \in \mathbb{N}$ be an index, and φ and ψ MTL formulas. Then the semantics for each MTL formula is defined inductively as follows:

$$\begin{aligned}
&w,i \models tt \\
&w,i \models p && \Leftrightarrow && p \in a_i \\
&w,i \models \neg\varphi && \Leftrightarrow && w,i \not\models \varphi \\
&w,i \models \varphi \vee \psi && \Leftrightarrow && w,i \models \varphi \text{ or } w,i \models \psi \\
&w,i \models \varphi \mathcal{U}_I \psi && \Leftrightarrow && \exists j \geq i : w,j \models \psi \wedge t_j - t_i \in I \wedge \forall i \leq k < j : w,k \models \varphi
\end{aligned}$$

We say that a word w is a model for an MTL-formula φ if $w,0 \models \varphi$. Each MTL formula φ defines a language which we denote as $\mathcal{L}(\varphi) = \{w \in (\Sigma \times \mathbb{T})^\omega \mid w,0 \models \varphi\}$.

Compared to LTL, because MTL is able to handle real-time constraints, it is a much more powerful logic. If the time domain is for example $\mathbb{R}$ on infinite words it can even simulate faulty Turing machines [OW06]. The drawback is, that many typical decision problems, like satisfiability of a formula, are undecidable for MTL [OW06].

2.3.3 Metric Interval Temporal Logic

The Metric Interval Temporal Logic (MITL, [AFH91]) is a fragment of MTL which still allows the real-time specifications of MTL, but with one exception: The intervals can now no longer be punctual ones, e.g. for $[a,b]$ with $a,b \in \mathbb{R}$ it has to hold that $a < b$. Other than that, the syntax and semantics of MITL and MTL coincide.

Definition 2.9 (*MITL syntax, [AFH91]*)

Let $p \in AP$ be an atomic proposition, $\mathbb{T}$ be a time domain, and let $I \in \mathbb{I}_{\mathbb{T}_\infty}$ be an interval which is not punctual. Then the syntax of an MITL formula φ is given by the following grammar:

$$\varphi := tt \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \varphi \mathcal{U}_I \varphi$$

Definition 2.10 (*MITL semantics, [AFH91]*)

Let $\mathbb{T}$ be a time domain, $w = (a_0,t_0)(a_1,t_1)(a_2,t_2)\cdots \in (\Sigma \times \mathbb{T})^\omega$ be an infinite timed word, $I \in \mathbb{I}_{\mathbb{T}_\infty}$ be a non punctual interval, $i \in \mathbb{N}$ be an index, and φ and ψ MITL formulas. Then the semantics for each MITL formula is defined inductively as follows:

$$\begin{aligned}
&w,i \models tt \\
&w,i \models p && \Leftrightarrow && p \in a_i \\
&w,i \models \neg\varphi && \Leftrightarrow && w,i \not\models \varphi \\
&w,i \models \varphi \vee \psi && \Leftrightarrow && w,i \models \varphi \text{ or } w,i \models \psi \\
&w,i \models \varphi \mathcal{U}_I \psi && \Leftrightarrow && \exists j \geq i : w,j \models \psi \wedge t_j - t_i \in I \wedge \forall i \leq k < j : w,k \models \varphi
\end{aligned}$$

We say that a word w is a model for an MITL-formula φ if $w,0 \models \varphi$. Each MITL formula φ defines a language which we denote as $\mathcal{L}(\varphi) = \{w \in (\Sigma \times \mathbb{T})^\omega \mid w,0 \models \varphi\}$.

Due to the removal of punctual intervals, many decision problems for MITL are, in contrast to MTL, decidable while most of the expressiveness, especially of practical properties, stays the same.

2.3.4 Signal Temporal Logic

The last logic we consider is the Signal Temporal Logic (STL, [MN04]), which can be considered as a special case of MITL by removing the next-operator ($\bigcirc$) and having propositions which map multiple real values to a single boolean value. This boolean value is then used like a normal proposition's value in the given MITL formula. Besides this new type of propositions, the syntax and semantics of STL are the same as for MITL.

Definition 2.11 (*STL syntax, [MN04]*)

Let $U = \{\mu_1, \ldots, \mu_n\}$ be a set of functions with $\mu_i : \mathbb{R}^m \to \mathbb{B}$ for all $i \in \{1, \ldots, n\}$ and let $I \in \mathbb{I}_{\mathbb{T}_\infty}$ be an interval that is not punctual. Then the syntax of an STL(U)-formula φ is given by the following grammar:

$$\varphi := tt \mid \mu_i \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \mathcal{U}_I \varphi$$

The semantics of STL differs in the sense that the input for an STL-formula is not a word anymore, but instead a so called signal s, which is a function $s : \mathbb{T} \to \mathbb{R}^m$, where $\mathbb{T}$ is a given time domain. Compared to words, a signal has a value at every timestamp and not only certain timestamps with the state of a system. Such a signal may even be a sinus curve, having values which continuous rise and fall. Because those signals may have values at any point in time, the next-operator does not make sense in this case. It is also important to note that STL is not really handling the signal directly, but instead by using the new type of propositions. The input signals are discretized after being processed by the μ_i at the points where the boolean values change into a word, which is than forwarded to the MITL formula.

Definition 2.12 (*STL semantics, [MN04]*)

Let $\mathbb{T}$ be a time domain, $U = \{\mu_1, \ldots, \mu_n\}$ be a set of functions, s with $s : \mathbb{T} \to \mathbb{R}^m$ be a signal, $I \in \mathbb{I}_{\mathbb{T}_\infty}$ be a non punctual interval, $t \in \mathbb{T}$ for a given time domain $\mathbb{T}$ be a timestamp, and φ and ψ STL formulas. Then the semantics for each STL formula is inductively defined as follows:

$$\begin{aligned}
&s,t \models tt \\
&s,t \models \mu_i && \Leftrightarrow && \mu_i(s(t)) \\
&s,t \models \neg\varphi && \Leftrightarrow && s,t \not\models \varphi \\
&s,t \models \varphi \vee \psi && \Leftrightarrow && s,t \models \varphi \text{ or } s,t \models \psi \\
&s,t \models \varphi \mathcal{U}_I \psi && \Leftrightarrow && \exists t' \geq t : w,t' \models \psi \wedge t' - t \in I \wedge \forall t \leq t'' < t' : s,t'' \models \varphi
\end{aligned}$$

We say that a signal s is a model for an STL formula φ if $s,0 \models \varphi$. Each STL formula φ defines a language which we denote as $\mathcal{L}(\varphi) = \{s \mid s,0 \models \varphi\}$.

The following example shows some kind of formulas that are valid or invalid for the given logics as well as some examples for fulfilment and violation of the property.

Example 2.13 (*Temporal Logics*)

The formula $\Box a \vee b \mathcal{U} c$ is a valid LTL formula over the set of atomic propositions $AP = \{a,b,c\}$ which states that either a has to hold at every position in the word or b has to hold until c held once. A word $\{b\}\{b\}\{b,c\}\emptyset^\omega$ would fulfil the formula, while a word $\{b\}^\omega$ would not.

Assume $\mathbb{T} = \mathbb{R}$. The formula $\Box a \vee b \mathcal{U}_{[3,3]} c$ is a valid MTL formula, but not a valid MITL formula, and therefore also not a valid STL formula, because the interval is punctual. A word

$$(\{b\},1.2)(\{b\},1.8)(\{b,c\},3)(\emptyset,4)(\emptyset,5)\ldots$$

would fulfil the formula, while a word

$$(\{b\},1.2)(\{b\},1.8)(\{b,c\},3.1)(\emptyset,4)(\emptyset,5)\ldots$$

would not.

For STL, the propositions have to be functions, for example the formula $s_1 \geq s_2 \mathcal{U}_{[3,5]} s_2 = s_3$ is a valid STL formula, where s_1, s_2 and s_3 relate to the three values on the input signal $s : \mathbb{T} \to \mathbb{R}^3$ from left to right. s would fulfil the formula if it is defined as follows:

$$s(t) = \begin{cases} (3,2,1) & \text{if } t < 4 \\ (3,1,1) & \text{otherwise} \end{cases}$$

It would not be fulfilled by the following signal:

$$s(t) = \begin{cases} (2,3,1) & \text{if } t < 4 \\ (3,1,1) & \text{otherwise} \end{cases}$$

2.4 Automata

In this section we will define different kinds of automata which all have a notion of state and acceptance in common, but differ in the exact acceptance condition, time handling, and memory data structures like stacks. We later define transducers based on these types of automata.

2.4.1 Automata on finite Words

An automaton on finite words, or short, finite automaton in this work, takes a finite word as input and either accepts or rejects it, which depends on whether the automaton is in an accepting state after reading the word or not.

Definition 2.14 (*Finite Automaton, [HMU06]*)

A Non-deterministic Finite Automaton *(NFA) is a 5-tuple* $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ *with*

- *a finite input alphabet* Σ,
- *a finite set of states* Q,
- *a set of initial states* $q_0 \subseteq Q$,
- *a set of accepting states* $F \subseteq Q$, *and*
- *a transition function* $\delta : Q \times \Sigma \to 2^Q$.

For a finite input word $w = w_0 w_1 w_2 \dots w_n \in \Sigma^*$ *with* $n \in \mathbb{N}$ *we call a sequence*

$$\rho = s_0 \xrightarrow{w_0} s_1 \xrightarrow{w_1} s_2 \xrightarrow{w_2} \cdots \xrightarrow{w_n} s_{n+1}$$

a run *of an NFA* $\mathcal{A}$ *iff* $s_0 \in q_0$ *and* $s_{i+1} \in \delta(s_i, w_i)$ *for all* $n \geq i \geq 0$. *The run* ρ *is called* accepting *iff* $s_{n+1} \in F$. *An NFA is called* deterministic, *or DFA, iff* $\forall q \in Q : \forall \sigma \in \Sigma : |\delta(q, \sigma)| = 1$.

The language of an NFA $\mathcal{A}$ *is the set of the words for which an accepting run on* $\mathcal{A}$ *exists, thus* $\mathcal{L}(\mathcal{A}) = \{w \mid$ *There exists an accepting run for* w *on* $\mathcal{A}.\}$.

A DFA takes finite input words and after reading a word, its run ends in a state. It accepts a word if the state is an accepting state and rejects it otherwise. Note that in the case of an NFA, there

exist multiple possible runs for a single word. In this case, the word is accepted if at least one of the runs is accepting.

The NFA and DFA are the simplest automata, which the following extend by infinite words, explicit time handling and stacks.

2.4.2 Automata on infinite Words

The next automaton is the natural extension of NFAs to infinite words, using the Büchi acceptance condition [Büc90] which is needed because the automaton does not stop anymore at the end of the word and thus can not use the acceptance criteria of an NFA. Formally, it is defined as follows:

Definition 2.15 (*Büchi-Automaton, [Büc90]*)

A Büchi Automaton (BA) *is a 5-tuple* $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ *with*

- *a finite input alphabet* Σ,
- *a finite set of states* Q,
- *a set of initial states* $q_0 \subseteq Q$,
- *a set of accepting states* $F \subseteq Q$, *and*
- *a transition function* $\delta : Q \times \Sigma \to 2^Q$.

For an infinite input word $w = w_0 w_1 w_2 \cdots \in \Sigma^\omega$ *we call a sequence*

$$\rho = s_0 \xrightarrow{w_0} s_1 \xrightarrow{w_1} s_2 \xrightarrow{w_2} \cdots$$

a run *of a BA* $\mathcal{A}$ *iff* $s_0 \in q_0$ *and* $s_{i+1} \in \delta(s_i, w_i)$ *for all* $i \geq 0$. *Let* $\inf(\rho)$ *be the set of states that are visited infinitely often during a run* ρ. *Then* ρ *is called* accepting *iff* $\inf(\rho) \cap F \neq \emptyset$. *A BA is called* deterministic, *or DBA, iff* $\forall q \in Q : \forall \sigma \in \Sigma : |\delta(q, \sigma)| = 1$.

The language an NBA $\mathcal{A}$ *describes is the set of the words for which an accepting run on* $\mathcal{A}$ *exists, thus* $\mathcal{L}(\mathcal{A}) = \{w \mid$ *There exists an accepting run for* w *on* $\mathcal{A}.\}$.

A Büchi automaton takes an infinite word, hence its runs are also infinitely long. The acceptance condition then checks if at least one of the infinitely often visited states is an accepting state. Unlike

NFAs and DFAs, NBAs and DBAs do not have the same expressive power because languages exist that can be recognized by an NBA but not by a DBA [MH84].

Next, we define a type of automaton that adds stacks to Büchi automata, called a pushdown automaton. The stack in such an automaton is used to store data when using a transition and to check which data was stored when considering which transition can be taken next. Therefore, it is used as a restricted type of memory.

Definition 2.16 (*Pushdown Automaton, [HMU06]*)

A Pushdown Automaton (PA) *is a 6-tuple* $\mathcal{A} = (\Sigma, Q, q_0, F, \Lambda, \delta)$ *with*

- *a finite input alphabet* Σ,
- *a stack alphabet* Λ *with* $\# \in \Lambda$,
- *a finite set of states* Q,
- *a set of initial states* $q_0 \subseteq Q$,
- *a set of accepting states* $F \subseteq Q$, *and*
- *a transition function* $\delta : Q \times \Sigma \times \Lambda \to 2^Q \times \Lambda^*$.

For an input word $w = w_0 w_1 w_2 \cdots \in \Sigma^\omega$ *we call a sequence*

$$\rho = s_0, \sigma_0 \xrightarrow{w_0, \lambda_0, \sigma_0'} s_1, \sigma_1 \xrightarrow{w_1, \lambda_1, \sigma_1'} s_2, \sigma_2 \xrightarrow{w_2, \lambda_2, \sigma_2'} \cdots$$

a run *of a PA* $\mathcal{A}$ *iff*

- $s_0 \in q_0$ *and* $\sigma_0 = \#$ *(starting at one start state and with empty stack) and*
- *for all* $i \geq 0$ *the following holds:* $(S_{i+1}, \sigma_i'') = \delta(s_i, w_i, \lambda_i)$ *with*
  - $s_{i+1} \in S_{i+1}$ *(follow-up state is in the set of possible next states),*
  - $\sigma_i = \lambda_i \sigma_i'$ *(right symbol is on top of stack) and*
  - $\sigma_{i+1} = \sigma_i'' \sigma_i'$ *(new stack is output of taken transition added on top of rest of old stack).*

Let $\inf(\rho)$ *be the set of states which are visited infinitely often during a run* ρ*. Then* ρ *is called* accepting *iff* $\inf(\rho) \cap F \neq \emptyset$*. A PA is called* deterministic*, or DPA, iff*

$$\forall q \in Q : \forall w \in \Sigma : \forall \lambda \in \Lambda : |\delta(q, w, \lambda)| = 1$$

The language a PA $\mathcal{A}$ *describes is the set of the words for which an accepting run on* $\mathcal{A}$ *exists, thus* $\mathcal{L}(\mathcal{A}) = \{w \mid$ *There exists an accepting run for* w *on* $\mathcal{A}.\}$.

The stack of a PA is a way to remember possibly infinite amounts of data. Elements can be pushed to and read from the stack to take transitions or when taking a transition, a check on emptiness of the stack can be done as condition. Each transition reads and deletes the top element of the stack if it is the correct one for the transition and pushes a finite number of elements (or no elements at all) to the rest of the stack.

As for Büchi automata, PAs are more expressive than DPAs [HMU06].

Last but not least, we define timed automata in this section. Instead of extending Büchi automata with a stack, timed automata are an extension by a notion of time and timing constraints in the transition function. This allows for checking constraints on the real-time distance between events instead of just checking the order of the events.

Before we get to the definition of timed automata themselves, we define clock constraint first. Clocks are used in timed automata to keep track of the time which can then be checked using clock constraints. There are two ways to use clocks for timed automata in the literature. The first is to be able to reset clocks to 0 on transitions and let them run implicitly in the background. The second is to be able to set clocks to the current time and just check against that time at a later stage in the run. While both variants do not make a difference for the properties of the timed automata, they change how clock constraints and the definition of the automata look. We use the latter variant of clocks in this thesis.

Definition 2.17 (*Clock Constraints, [AH92]*)

Let C *be a set of variables, called clocks, and* $\mathbb{T}$ *be a time domain. A* clock constraint $\vartheta \in \Theta(C)$ *over the set of clock constraints* $\Theta(C)$ *is defined by the grammar*

$$\vartheta := true \mid T \leq x + c \mid T \geq x + c \mid \neg\vartheta \mid \vartheta \wedge \vartheta$$

where $x \in C$*, and* $c \in \mathbb{T}$ *is a constant and* T *refers to the current time.*

With the definition of clock constraints, we can now define timed automata. In a timed automaton the transition function additionally takes a timing constraint which must be fulfilled for being able to take the transition. Also, every transition outputs the set of clocks that are to be reset to T, the current time.

Definition 2.18 (*Timed Automaton, [AH92]*)

A Timed Automaton *(TA) is a 6-tuple* $\mathcal{A} = (\Sigma, Q, q_0, F, C, \delta)$ *with*

- *a finite input alphabet* Σ,
- *a finite set of states* Q,
- *a set of initial states* $q_0 \subseteq Q$,
- *a set of accepting states* $F \subseteq Q$,
- *a set of clocks* C, *and*
- *a transition function* $\delta : Q \times \Sigma \times \Theta(C) \to 2^Q \times 2^C$.

For an input word $w = (w_0, t_0)(w_1, t_1)(w_2, t_2)\ldots$ *we call a sequence*

$$\rho = s_0, v_0 \xrightarrow{(w_0,t_0),\vartheta_0,r_0} s_1, v_1 \xrightarrow{(w_1,t_1),\vartheta_1,r_1} s_2, v_2 \xrightarrow{(w_2,t_2),\vartheta_2,r_2} \cdots$$

a run of a TA $\mathcal{A}$ *where* $v_i : C \to \mathbb{R}$ *are functions mapping every clock to its current value iff*

- $s_0 \in q_0$,
- $\forall c \in C : v_0(c) = 0$, *and*
- $\forall_{i \geq 0}$:
  - $\delta(s_i, w_i, \vartheta_i) = (s_{i+1}, r_i)$,
  - $\forall_{c \in r_i} v_{i+1} = v_i[c \leftarrow t_i]$ *(reset clocks in* r_i*), and*
  - $t_i, v_i \models \vartheta_i$ *(which means* ϑ_i *resolved to* true *if* T *is replaced with* t_i *and the corresponding values from* v_i *are used for the clocks).*

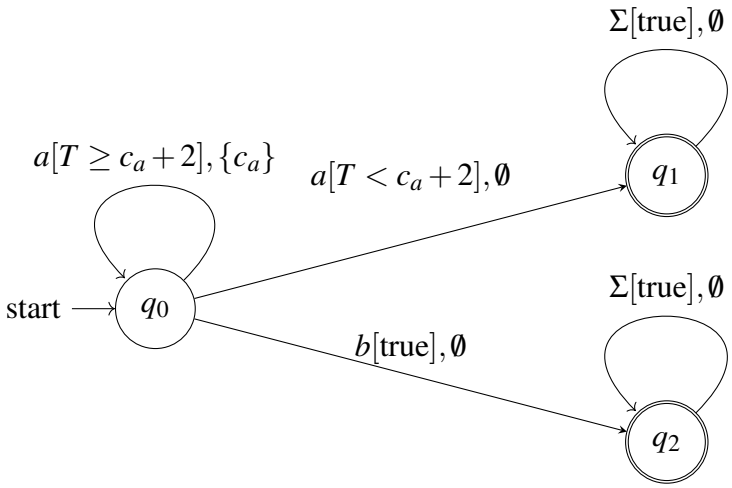


Figure 2.1: A TA over the alphabet $\Sigma = \{a, b\}$ with three states, out of which two are accepting, and one clock c_a. The Σ at the transitions is a short notion for both input symbols being valid for these transitions.

> *Then ρ is called* accepting *iff* $\inf(\rho) \cap F \neq \emptyset$. *A TA is called deterministic, or DTA, iff for every two different transitions* $(q, \sigma, \vartheta_1, q'_1, R_1), (q, \sigma, \vartheta_2, q'_2, R_2) \in \delta$ *the conjunction of their clock constraints* $\vartheta_1 \wedge \vartheta_2$ *is unsatisfiable.*
>
> *The language a TA $\mathcal{A}$ describes is the set of the words for which an accepting run on $\mathcal{A}$ exists, thus* $\mathcal{L}(\mathcal{A}) = \{w \mid$ *There exists an accepting run for w on $\mathcal{A}$.*$\}$.

The following example shows how timed automata work in general.

Example 2.19 (*Timed Automaton*)

> Consider the timed automaton in Figure 2.1. It accepts all runs where either a b occurs somewhere, ignoring time, or where there is any symbol with an a which is less then two seconds away from the previous one. That is because the loop at the start state resets the clock to the current time, if the last event was two seconds or more ago and the transition from q_0 to q_1 can only be taken if an a occurs and the last resets of the clock was less than two seconds away.

Compared to the deterministic versions of Büchi automata and pushdown automata, the definition of determinism for timed automata is a bit different. As long as it is not possible to fulfil both timing constraints on two transitions with the same input symbol starting from the same state, the automaton is called deterministic, because there is always only a single path that can be chosen. Hence, even in a DTA, multiple transitions can start from the same state with the same input symbol, which is not possible in DBAs oder DPAs.

As for the last two types of automata, TAs are strictly more expressive than DTAs [AD94]. MITL is strictly less expressive than TAs, but not less expressive than DTAs [AFH91]. Also, MTL is incomparable to TAs [OW05].

2.5 Turing Machines

Next we define the model of Turing machine [Tur36], which is a model representing all computable functions. Because Turing machines are equally expressive in their non-deterministic and deterministic versions, we will stick with deterministic ones which fit better to our use of them. We will also directly give a definition of Turing machines with three tapes, which are used to remember data during calculations, which best fits to this thesis. Additionally, more tapes do not change anything in terms of properties of the Turing machine.

Definition 2.20 (*Turing Machine, [Tur36]*)

A Deterministic Turing Machine with 3 Tapes *(3DTM) is a 5-tuple* $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ *with*

- *an input alphabet* Σ *with* $\square \in \Sigma$ *(symbol for empty tape positions) and* $\Sigma \cap \{L, R\} = \emptyset$,
- *a finite set of states* Q,
- *an initial state* $q_0 \in Q$,
- *a set of accepting states* $F \subseteq Q$, *and*
- *a transition function* $\delta : Q \times \Sigma \times \Sigma \times \Sigma \to Q \times \Sigma \times \{L, R\} \times \Sigma \times \{L, R\} \times \Sigma \times \{L, R\}$.

A configuration *of a Turing machine is a 7-tuple* $(q, w_1, w_2, w_3, n_1, n_2, n_3)$ *where*

- $q \in Q$ *is the current state of the machine,*
- $w_1, w_2, w_3 \in \mathbb{N} \to \Sigma$ *are the contents of the three tapes, and*
- $n_1, n_2, n_3 \in \mathbb{N}$ *are the current positions of the tapes.*

We call a sequence

$$c_0 c_1 c_1 \dots c_n$$

with $c_i = (q^i, w_1^i, w_2^i, w_3^i, n_1^i, n_2^i, n_3^i), 0 \leq i \leq n$ *a run of a 3DTM* $\mathcal{A}$ *iff*

- $q^0 = q_0$,

- $\forall 1 \leq i \leq n : \delta(q^{i-1}, w_1^{i-1}(n_1^{i-1}), w_2^{i-1}(n_2^{i-1}), w_3^{i-1}(n_3^{i-1})) = (q^i, w_1^i(n_1^{i-1}), x_1^i, w_2^i(n_2^{i-1}), x_2^i, w_3^i(n_3^{i-1}), x_3^i)$,
- $\forall 1 \leq i \leq n : \forall 1 \leq m \leq 3 : n_m^i = \begin{cases} n_m^{i-1} + 1 & \text{if } x_m^i = R \\ n_m^{i-1} - 1 & \text{if } x_m^i = L \wedge n_m^{i-1} \geq 1 \\ n_m^{i-1} & \text{otherwise} \end{cases}$
- $\forall 1 \leq i \leq n : \forall 1 \leq m \leq 3, \forall x \neq n_m^{i-1} : w_m^i(x) = w_m^{i-1}(x)$, *and*
- *in c_n, no transition can be taken.*

A run is called accepting *iff $q^n \in F$.*

The language a 3DTM $\mathcal{A}$ describes is the set of the initial contents for the tapes for which the run on $\mathcal{A}$ is accepting, thus $\mathcal{L}(\mathcal{A}) = \{(w_0^0, w_1^0, w_2^0) \mid$ The run for w_0^0, w_1^0 and w_2^0 on $\mathcal{A}$ is accepting.$\}$.

Compared to a DFA, a Turing machine is also an automaton with states some of which are accepting. But it contains a special mechanism for memory, called tapes. Every transition it takes, a Turing machine can read one symbol from each of its tapes, write one symbol to each of its tapes and move the position pointer for each tape either one position to the right or one to the left. Being in a state with certain symbols on each tape and a position for every tape is called a configuration. Every time a Turing machine takes a transition it switches from one configuration to another by changing the state, the values at the positions of the tapes, and the position pointers. It accepts a tuple of initial contents of the tapes if the state of the last configuration (which is reached when no transition is active anymore) the Turing machine is in when it stops is accepting. Note that not every Turing machine stops on every input and it is undecidable whether a Turing machine stops on an input in general. This problem is called the *halting problem* [Dav58].

2.6 Streaming Semantics and Transducers

In the previous three sections, we defined many well known logics and automata as well as Turing machines. One of the main goals of this thesis is to show how TeSSLa and various fragments of it relate to these formalisms as well as to get complexity results on different decision problems. But, while TeSSLa works on streams which means the input / output relation works in a more interactive way, the previously mentioned formalisms do not, which makes it hard to compare. A solution would be to define a streaming semantics for these logics and automata. But there is no

single and direct way of defining this as a natural extension, because there are multiple possibilities with different properties and such, many results shown in this thesis would depend on and change with the definition of these streaming semantics for the well known formalisms.

In this section, we will overcome these issues by introducing well known formalisms which operate directly on streams or are easily extendable to do so by always using finite prefixes of the complete word as input word and generate an output symbol for all of these prefixes. The possibly infinite sequence of these output symbols is then the output stream. We will call this input / output behaviour a *stream transformation*, which will be the underlying model of the formalisms defined in this section, as well as the one the semantics for TeSSLa, as defined in the next section, operate on.

The specification language LOLA, for example, is a stream language to specify correctness properties or properties for doing analysis on system behaviour and it already operates naturally on streams, so no additional semantics which is able to implement stream transformations is needed. Compared to that, in case of the types of automata defined in the last section, these extensions lead to the corresponding types of transducers, which are already well established in literature. Most of the formalisms defined in the previous section serve as basic knowledge for the streaming formalisms defined in this section, which we will later compare to TeSSLa and different fragments of it.

In general, it is important for functions on streams to be monotonic and continuous, because informally, these properties allow a function on streams to work in a step-by-step (i.e., prefix by prefix, sometimes also called interactive) evaluation of the input streams. Otherwise, an incremental evaluation would not be possible and the streaming semantics would not be suitable for runtime verification. As mentioned before, such a type of functions will be defined as stream transformations in the following subsection.

Before giving the definition of the stream language LOLA and definitions of the stream semantics for different formalisms, we will define a notion of streams and stream transformations first.

2.6.1 Streams and Stream Transformations

Compared to words, intuitively, a stream is a continuous flow of data over a certain data domain and a certain time domain. In timed words, every input symbol has a timestamp and the timed word only expresses which data we saw at those timestamps. In streams, at every timestamp available in

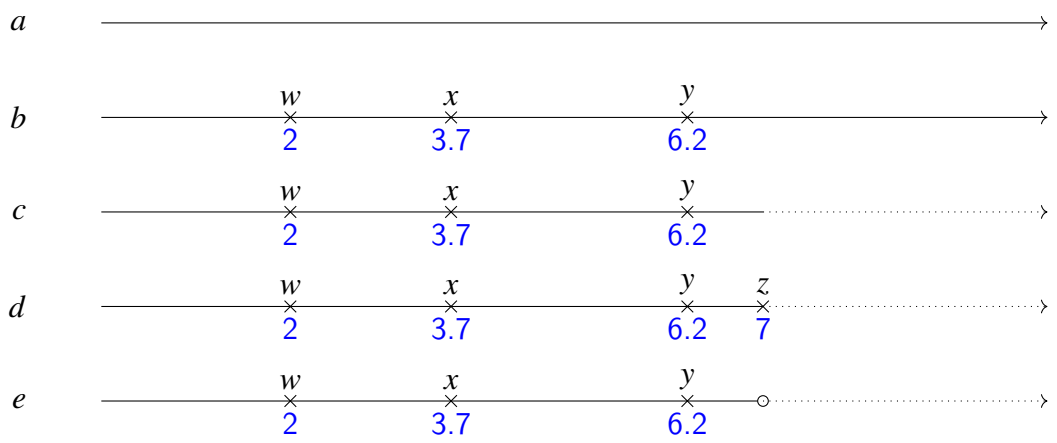


Figure 2.2: Shows the different kinds of elements that can occur on a stream. A line indicates that there is no event, a cross indicates an event with a value above and a timestamp, in blue, below. A dotted line indicates that the knowledge about the stream ended. If the line directly changes to a dotted line, it means that the knowledge ended inclusively, while a circle at the end indicates that it ended exclusively. If an event is exactly at the end of the stream, here at timestamp 7, we indicate this as usual.

the given time domain, we assume that some information exists, which either indicates that there is no event at that timestamp or that there exists an event with a timestamp and a value.

Before we get to the formal definitions of a stream, let us first get some intuition. Consider the streams depicted in Figure 2.2. On an intuitive level, a stream s is a function $s : \mathbb{T} \to \mathbb{D}$ for a given time domain $\mathbb{T}$ and a data domain $\mathbb{D}$, even though our definitions later are a bit more complex. Now, the stream a in the figure is the empty stream with complete knowledge. We know that on this stream no event occurs. We denote the absence of an event with $\bot$ in a stream, which means that $\forall t \in \mathbb{T} : a(t) = \bot$. On stream b, we have three events at timestamps 2, 3.7, and 6.2 with values w, x, and y, respectively. This means that $b(2) = w$, $b(3.7) = x$, and $b(6.2) = y$ as well as $b(t) = \bot$ for all other timestamps. With the dotted part at the end of the arrow line, stream c now indicates that after timestamp 7, the rest of the stream is currently unknown. We call this the progress of a stream. In the case of c, this progress is inclusively. Additionally, we denote with ? that a timestamp is beyond the current progress of the stream, therefore $\forall t > 7 \in \mathbb{T} : c(t) =?$ and for all other t the same holds as for b. Then, the streams d and e depict the different kinds of knowledge endings a stream can have. Thus, in stream d, the progress ends on an event, which compared to c means that $d(7) = z$ instead of $c(7) = \bot$. The stream e indicates exclusive progress: while for c it holds that $\forall t > 7 \in \mathbb{T} : c(t) = ?$, for e it holds that $\forall t \geq 7 \in \mathbb{T} : e(t) = ?$, therefore the $>$ is changed to $\geq$.

We define streams in terms of a general notion for a time domain which only requires a total order

and corresponding arithmetic operators.

We consider two types of streams in this thesis: completed streams and event streams. Even though intuitively, streams can be seen as functions, defining them just as functions mapping timestamps to values would allow strange behaviour, like two areas with an infinite number of events converging to two timestamps, which we want to get rid of. Therefore, we define them as a sequence to forbid mentioned behaviour. Such a sequence is a finite or infinite timed word whose timestamps are possibly converging to some timestamp. As introduced after the definitions, we still interpret streams as a function, which helps us later to simplify definitions, theorems or examples later in this thesis.

A completed stream is a completely known stream with a finite or infinite number of events. If it has a finite number of events, it is still $\bot$ until infinity, to indicate that we know that no further events occur in the future.

Definition 2.21 (*Completed Stream*)

> *A* completed stream *over a time domain* $\mathbb{T}$ *and a data domain* $\mathbb{D}$ *is a finite or infinite sequence* $s = a_0 a_1 \cdots \in \mathcal{S}^{\infty}_{\mathbb{D}} = (\mathbb{T} \cdot \mathbb{D})^{\omega} \cup ((\mathbb{T} \cdot \mathbb{D})^* \cdot \{\infty\})$ *where* $a_{2i} < a_{2(i+1)}$ *for all* i *with* $0 < 2(i+1) < |s|$ *(*$|s|$ *is* ∞ *for an infinite number of events).*

Our notion of a completed stream works as follows: It is mostly an alternating sequence of timestamps and data values where each following timestamp has to be strongly monotonically increasing. The sequence is either an infinite sequence of those timestamp-data-pairs or a finite sequence of such pairs followed by ∞ to denote that no further data will follow (therefore have complete knowledge of the stream and no ? occurs). All timestamps which do not occur explicitly in the stream notion are assumed to be $\bot$, this means that there is no event at that timestamp.

Note that a completed stream can be a timestamp-wise converging sequence, which converges to some timestamp but may never reach it. However it is not possible that one stream converges to multiple timestamps due to its definition. Additionally, every such stream can be represented as some kind of a timed word, even though timed words normally do not converge and ∞ has to be encoded in a special way.

As a completed stream is only the first step to the stream model we use throughout this thesis, let us first move on to the next definition and consider more examples afterwards.

Next, we give a definition of the stream model we use throughout this thesis. Completed streams are also part of this stream model, but also uncompleted streams exist which neither are an infinite

sequence of events nor end with an ∞, but instead are only known (have progress) up to some finite timestamp.

Conceptually, event streams are still similar to timed words but are possibly only known inclusively or exclusively up to a certain timestamp, the timestamp up to which we know the values of the stream, that might be infinite (then called a completed stream as defined before). A stream might contain an infinite number of events even if the stream is only known up to a finite timestamp.

Definition 2.22 (*Event stream, [CHL$^+$18]*)

An event stream *over a time domain $\mathbb{T}$ and a data domain $\mathbb{D}$ is a finite or infinite sequence $s = a_0 a_1 \cdots \in \mathcal{S}_{\mathbb{D}} = \mathcal{S}^{\infty}_{\mathbb{D}} \cup (\mathbb{T} \cdot \mathbb{D})^+ \cup (\mathbb{T} \cdot \mathbb{D})^* \cdot (\mathbb{T} \cup \mathbb{T} \cdot \{\bot\})$ where $a_{2i} < a_{2(i+1)}$ for all i with $0 < 2(i+1) < |s|$ ($|s|$ is ∞ for infinite number of events).*

Informally, we say an event stream has an event with value d at time t if in its sequence d directly follows t. We say an event stream is known at time t if it contains a strictly larger timestamp or a non-strictly larger timestamp followed by a data value or $\bot$. Where convenient, we also see streams as functions.

Definition 2.23 (*Streams as Functions, [CHL$^+$18]*)

A completed stream $s \in \mathcal{S}^{\infty}_{\mathbb{D}}$ can be seen as function $s : \mathbb{T} \to \mathbb{D} \cup \{\bot\}$ with

$$s(t) = \begin{cases} d & \text{if } s \text{ contains } td \\ \bot & \text{otherwise} \end{cases}$$

An event stream $s \in \mathcal{S}_{\mathbb{D}}$ can be seen as function $s : \mathbb{T} \to \mathbb{D} \cup \{\bot, ?\}$ as follows:

$$s(t) = \begin{cases} d & \text{if } s \text{ contains } td \\ \bot & \text{if } s \text{ does not contain } t \wedge \exists t' > t : s \text{ contains } t \vee s \text{ ends in } t\bot \\ ? & \text{otherwise} \end{cases}$$

Intuitively, for a completed stream s, $s(t)$ is a value d if s has an event with value d at the timestamp t or $\bot$ if there is no event at time t.

Intuitively, for an event stream s, $s(t)$ is a value d if s has an event with value d at the timestamp t or $\bot$ if there is no event at time t. For timestamps after the known part of the stream $s(t)$ is ?. We

call the timestamp at which the known part of the stream ends the progress of the stream.

Definition 2.24 (*Progress of Streams*)

For a stream $s \in \mathcal{S}_{\mathbb{D}}$ over a time domain $\mathbb{T}$ we call a timestamp $t \in \mathbb{T}_\infty$ the exclusive progress *of s if it is the maximal timestamp such that*

$$\forall t' < t : s(t') \neq ?$$

On the other hand, we call t the inclusive progress *of s if it is the maximal timestamp such that*

$$\forall t' \leq t : s(t') \neq ?$$

If ? occurs the first time on the stream at a timestamp, we call it the exclusive progress (the stream ends with a timestamp), if the first ? occurs directly after the timestamp at which the stream ends we call it the inclusive progress (the stream ends with $\perp$ or an event).

We use $ticks(s)$ for the set $\{t \in \mathbb{T} \mid s(t) \in \mathbb{D}\}$ of timestamps where a stream s has events. We use the data domain $\mathbb{U}$ for stating that events of a stream carry only the single value $\square$.

For the rest of the thesis, when we write *stream* we mean an event stream and we write *completed stream* to specifically address completed streams. Furthermore, with *uncompleted stream* we describe the set of streams which are not completed streams, thus for the set of uncompleted streams $\mathcal{S}^*_{\mathbb{D}}$ it holds that $\mathcal{S}^*_{\mathbb{D}} = \mathcal{S}_{\mathbb{D}} \setminus \mathcal{S}^\infty_{\mathbb{D}}$.

Next, we define the notion of continuous and discrete streams.

Definition 2.25 (*Continuous Streams*)

We call a stream continuous *iff its time domain is continuous, hence the time domain is the set of real numbers, $\mathbb{R}$.*

While continuous streams are continuous in their time domain (events can be anywhere and at any distance to each other), discrete streams are not only allowing events at certain points, but also force events to be at those timestamps. We use $\mathbb{N}$ as the time domain for discrete streams, even though it can be any time domain with only a finite number of elements between two elements.

Definition 2.26 (*Discrete Streams*)

We call a stream s over a data domain $\mathbb{D}$ discrete *iff it is uncompleted, its time domain is discrete,*

hence the time domain is the set of natural numbers, and

$$\forall t \in \mathbb{T} : s(t) \in \mathbb{D} \vee s(t) = ?$$

In our setting to stick close to the definition of streams for synchronous stream languages like LOLA, Lustre, or Esterel for the comparison of those to TeSSLa in the setting of discrete streams, a discrete stream must have an event at every timestamp before the progress ends. This is only a technical detail, however, if there would be the possibility that there is no event at some timestamp, we would need to add events there with a special symbol for some of the languages mentioned before.

Before we get to the definition of prefixes for streams, we will show some examples for different types of streams as well as how we depict them in the rest of this thesis.

Example 2.27 (*Streams*)

This example contains three different streams from different types to show the differences and how they are depicted as a sequence and as a function.

The stream $s = 2\ a\ 3.7\ b\ \infty$ is a *completed stream* over time domain $\mathbb{R}$ ($\mathbb{T} = \mathbb{R}$). As a function, it looks as follows for every timestamp $t \in \mathbb{T}$:

$$s(t) = \begin{cases} a & \text{if } t = 2 \\ b & \text{if } t = 3.7 \\ \bot & \text{otherwise} \end{cases}$$

The stream $s' = 2\ a\ 3.7\ b\ 5$ is an *event stream* but not an infinite stream, again over the time domain $\mathbb{R}$. As a function, it looks as follows for every timestamp $t \in \mathbb{T}$:

$$s'(t) = \begin{cases} a & \text{if } t = 2 \\ b & \text{if } t = 3.7 \\ \bot & \text{if } 0 \leq t < 5 \wedge t \neq 2 \wedge t \neq 3.7 \\ ? & \text{otherwise} \end{cases}$$

Because their time domain is $\mathbb{R}$, both streams are continuous. The stream $s'' = 0\ a\ 1\ a\ 2\ a\ 3\ b$ over the time domain $\mathbb{N}$ is an *event stream* that is *discrete*. As a function, it looks as follows for

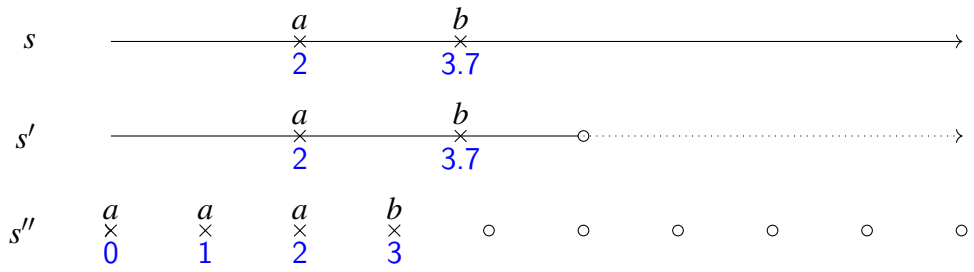


Figure 2.3: Shows the three streams s, s', and s''. The drawn line indicates $\bot$ at the position, so it is known that no event exists at those timestamps. The crosses mark the positions of events on the streams, with the value being denoted above and the timestamp below. The arrow at the end of stream s shows that the stream would go on with $\bot$ forever, while on stream s' the circle denotes that the progress of the stream ends there. Because the stream s'' has $\mathbb{N}$ as time domain, the stream contains only integer timestamps, hence the drawn line is left out because nothing is in between the four events. The circles after timestamp 3 indicate that the progress of this discrete stream has ended there after timestamp 3.

every timestamp $t \in \mathbb{T}$:

$$s''(t) = \begin{cases} a & \text{if } 0 \leq t \leq 2 \\ b & \text{if } t = 3 \\ ? & \text{otherwise} \end{cases}$$

A drawing of the three streams can be seen in Figure 2.3.

Next in this section, we will define the notion of a prefix of a stream as well as the notion of a maximum prefix and a supremum that come along with prefixes.

Definition 2.28 (*Prefixes of Streams, [LSS+19]*)

The prefix relation *over $\mathcal{S}_{\mathbb{D}}$ is the least relation that satisfies $s \sqsubseteq s$, $u \sqsubseteq s$ if $\exists v : uv \sqsubseteq s$ and $ut' \bot \sqsubseteq s$ if $ut \sqsubseteq s$, $t' < t$, $t \in \mathbb{T}_\infty$ and $t' \in \mathbb{T}$.*

Intuitively, a stream s' is a prefix of a stream s if s' contains all the events of s up to a certain point in time and the progress ends before the next event on s would happen. Note that regarding the prefix relation, the stream $s = 0$ is the lowest element, as it is the stream without any progress and for all $s' \in \mathcal{S}_{\mathbb{D}}$ it holds that $s \sqsubseteq s'$. Additionally, we abuse notation and write $(s_1, \ldots, s_n) \sqsubseteq (s'_1, \ldots, s'_n)$ to denote that $\forall 1 \leq i \leq n : s_i \sqsubseteq s'_i$.

We refer to the supremum of all known timestamps of a stream as inclusive or exclusive progress, as defined before, depending on whether it is itself a known timestamp. The prefix relation realises

the intuition of cutting a stream at a certain point in time while keeping or removing the cutting point and leading to inclusive or exclusive progress at that point.

In general, streams are a model which allows for the occurrence of some possibly unwanted behaviour, like under the time domain $\mathbb{R}$ it can happen that the data value changes an infinite number of times in a finite time interval on a stream. Even though, as discussed before, we will not consider streams like a sinus, we still allow such behaviour of streams consisting of events with timestamps converging to a certain threshold timestamp. The notion of Zenoness describes this behaviour and while a sinus is also Zeno, we will use the term in this thesis to refer to streams with events with converging timestamps. Informally, a stream is called Zeno if there exist any two timestamps with an infinite number of events between them. This can happen if the timestamps of an infinite number of events converge towards the timestamp 1 on a stream, but never reach 1. This results in an infinite number of events between the timestamps 0 and 1.

Definition 2.29 (*Zenoness of Streams*)

Let $\mathbb{D}$ be a data domain. A stream $s \in \mathcal{S}_{\mathbb{D}}$ is called Zeno *iff it holds that:*

$$\exists t_1, t_2 \in \mathbb{T} : s(t_1), s(t_2) \in \mathbb{D} \wedge |\{t \mid t_1 < t < t_2 \wedge s(t) \in \mathbb{D}\}| = \infty$$

Note that discrete streams can not be Zeno, while continuous streams can be, because the time domain needs to fulfil that there are infinitely many timestamps between two timestamps such that streams over this domain can be Zeno.

Next in this section, we define a notion of functions on streams, called stream transformations, representing the streaming behaviour as described before. Besides operating on streams, stream transformations have to be monotonic and continuous, which means that these functions output can be calculated by calculating an output for each finite prefix of the input streams and seeing these outputs in order as the output of the function, without changing older parts of the output later. Additionally, if the input streams are completely known, therefore do not contain a ?, then the output streams should also be completely known. This fits to the definition of a function transforming streams, because if we have complete knowledge, outputting only incomplete knowledge makes no sense. The reason is that the input can not get any more information in this case and so the output should also have full information.

Definition 2.30 (*Stream Transformation*)

We call a function f on streams with $f : \mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_n} \to \mathcal{S}_{\mathbb{D}'_1} \times \cdots \times \mathcal{S}_{\mathbb{D}'_m}$ a stream transform-

ation *iff* f *is monotonic and continuous and*

$$\forall x_1, \ldots, x_n \in \mathcal{S}^{\infty}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}^{\infty}_{\mathbb{D}_n} : f(x_1, \ldots, x_n) \in \mathcal{S}^{\infty}_{\mathbb{D}'_1} \times \cdots \times \mathcal{S}^{\infty}_{\mathbb{D}'_m}$$

The following example illustrates how the stream semantics of a stream transformation works compared to any arbitrary function on streams.

Example 2.31 (*Stream Transformations*)

Consider a stream x over time domain $\mathbb{N}$ with

$$x(t) = \begin{cases} b & \text{if } t = 1 \vee t = 2 \vee t > 6 \\ a & \text{if } t = 0 \vee t = 4 \\ \bot & \text{otherwise} \end{cases}$$

which looks like $0\ a\ 1\ b\ 2\ b\ 4\ a\ 7\ b\ 8\ b\ 9\ b \ldots$. Further, consider a function f on streams with

$$f(s)(t) = \begin{cases} \text{tt} & \text{if } s(t) = b \wedge s(t+1) = b \\ \text{ff} & \text{if } s(t) = a \vee s(t) = b \wedge s(t+1) \neq b \\ \bot & \text{otherwise} \end{cases}$$

and a function f' on streams with

$$f'(s)(t) = \begin{cases} \text{tt} & \text{if } \forall t' \geq t \Rightarrow s(t') = b \\ \text{ff} & \text{if } s(t) = a \vee s(t) = b \wedge \exists t' > t \wedge s(t') \neq b \\ \bot & \text{otherwise} \end{cases}$$

as well as a function f'' on streams with

$$f''(s)(t) = \begin{cases} (0, |s|) & \text{if } t = 0 \\ \bot & \text{otherwise} \end{cases}$$

The outputs of the functions for the input stream x would be

$$f(x) = 0\ \text{ff}\ 1\ \text{tt}\ 2\ \text{ff}\ 4\ \text{ff}\ 7\ \text{tt}\ 8\ \text{tt}\ 9\ \text{tt}\ldots$$
$$f'(x) = 0\ \text{ff}\ 1\ \text{ff}\ 2\ \text{ff}\ 4\ \text{ff}\ 7\ \text{tt}\ 8\ \text{tt}\ 9\ \text{tt}\ldots$$
$$f''(x) = 0\ \infty$$

As one can see, f' can make statements about behaviour regarding the infinite future. Therefore, it is not continuous since on every finite prefix of the stream, f' would output ff from timestamp 7 on, it only outputs tt on the infinite stream. Thus, f' is not continuous. The function f'' changes the output at the timestamp 0 depending on the length of the stream, thus, it is not monotonic. As such, both functions are not stream transformations. f on the other hand does fulfil both properties and is therefore a stream transformation.

Lastly, based on having defined stream transformations, we define the notion of a maximum prefix. Compared to a prefix, a maximum prefix describes if the output of a stream transformation is maximal in the sense that for all elongations of the input, the previous output is a prefix of the new output.

Definition 2.32 (*Maximum Prefix*)

Let f be a stream transformation with $f(s_1,\ldots,s_n) = s'_1,\ldots,s'_m$. Then $s'_1,\ldots,s'_m$ is called a maximum prefix *regarding f and $s_1,\ldots,s_n$ iff $\forall x_1,\ldots,x_n : (\forall 1 \leq i \leq n : s_i \sqsubseteq x_i) \rightarrow (\forall 1 \leq i \leq m : s'_i \sqsubseteq y_i)$ where $f(x_1,\ldots,x_n) = y_1,\ldots,y_m$.*

2.6.2 LOLA

LOLA [DSS+05] is a specification language which, compared to logics such as LTL, MTL, or MITL, is build to reason over arbitrary data domains and to execute all kind of operations on such data, instead of just reading a word and deciding the fulfilment or violation of a property. Intuitively, a LOLA specification takes a set of input streams and uses stream transformations to transform those into a set of output streams with the help of various intermediate streams.

In the following we will define syntax and semantics of LOLA and various fragments of it. The basic version of LOLA is defined over discrete streams, while there has been an informal extension of LOLA to continuous streams called Real-time LOLA (RTLola, [FFST19, FFS+19]). Thus, for the definition of LOLA, we assume discrete streams.

A LOLA specification is syntactically a system of equations consisting of possibly mutually recursive applications of functions as well as some special operators to streams.

Definition 2.33 (*LOLA syntax, [DSS+05]*)

Let I be a set of input streams. Further, let c be a constant, f be a k-ary function, and $i \in \mathbb{Z} \setminus \{0\}$. Then a LOLA specification φ is a system of equations with equations of the form $x := e$, where the syntax of each e is given through the following grammar, with $s \in I$ being an input stream and y the left hand side of an equation of φ:

$$e ::= y \mid s \mid f(e, \ldots, e) \mid e[i, c]$$

Thereby, f is a function on streams implementing the behaviour of a function on values for the values of the events happening at the same timestamp. The semantics are given by the individual function definition. For $e[i,c]$, the semantics are given in the following definition. Intuitively, $e[i,c]$ is the operator used to access older values or to relate to future values. i defines how many timestamps forward (in the positive case) or backward (in the negative case) we have to look on e for the value. c is the constant which is the result if i specifies a position outside of the beginning or the end of a stream. Because our streams do not have distinct endings, we do not consider this case in the following definition.

Definition 2.34 (*LOLA semantics, [DSS+05]*)

Let φ be a LOLA specification. Then the semantics of the LOLA specification φ is a function $\varphi : \mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_n} \to \mathcal{S}_{\mathbb{D}'_1} \times \cdots \times \mathcal{S}_{\mathbb{D}'_m}$ over finite, discrete streams and we write $\varphi(I) = O$, where O is the set of output streams after the equations have been evaluated using the input streams in I. The semantics for the system of equations that is φ is given as the least fixed-point of the equations interpreted as a function of the stream variables and fixed input streams.

The semantics for each operator for a timestamp $t \in \mathbb{T}$ is given as follows:

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

$$e[i,c](t) = \begin{cases} c & \text{if } t + i < 0 \\ s(t+i) & \text{otherwise} \end{cases}$$

The language a LOLA specification φ represents is given as $\mathcal{L}(\varphi) = \{(I, O) \mid \varphi(I) = O\}$.

The LOLA semantics are monotonic and continuous in the input streams, which we state with the following proposition. As such, a LOLA specification is a stream transformation on discrete streams.

Proposition 2.35 (*Properties of LOLA Semantics*)

The semantics of a LOLA specification is monotonic and continuous in the input streams.

Next, we define the dependency graph of a LOLA specification which we use afterwards to define the term of well-formedness for such specifications.

Definition 2.36 (*Dependency Graph of LOLA Specifications, [DSS+05]*)

Let φ be a LOLA specification. The dependency graph *for φ is a weighted and directed multi-graph $G = (V,E)$ where $V = \{s \mid s \text{ is an equation or an input stream.}\}$. An edge (s_i, s_j, w) is in E iff s_i and s_j are equations and s_i contains $s_j[w,c]$, for any c, as a subexpression. Additionally, an edge $(s_i, s_j, 0)$ is in E iff s_i and s_j are equations and s_i does contain s_j as a subexpression but not $s_j[w,c]$, for any c and any w.*

The definition of well-formed specifications is an important one, because it states which specifications only have one possible output (i.e., fixed-point) for each set of input streams and thus allow a deterministic evaluation of the input [DSS+05].

Definition 2.37 (*Well-formed LOLA Specifications, [DSS+05]*)

A LOLA specification is called well-formed *iff its dependency graph (V,E) has no cycles $(s_1, s_2, w_1) \dots (s_n, s_1, w_n) \in E^*$ with*

$$\sum_{i=1}^{n} w_i = 0$$

For the rest of this thesis, we will only consider well-formed LOLA specifications unless explicitly stated otherwise.

Lastly for LOLA, we define two fragments which restrict the use of future references in some way or completely.

Definition 2.38 (*LOLA Fragments, [DSS+05]*)

A LOLA specification is called a LOLA_{eff} specification iff there exists a k such that only paths $(s_1, s_2, w_1) \dots (s_n, s_{n+1}, w_n) \in E^*$ in the dependency graph (V,E) of the specification exist with

$$\sum_{i=1}^{n} w_i < k$$

A LOLA specification is called a $LOLA_{past}$ specification iff for every subexpression of type $s[w,c]$ it holds that $w \leq 0$.

We define LOLA_{eff} as the fragment of LOLA which is efficiently monitorable, thus future references are only used in a bounded way. This means there is no recursion which always depends on a future value within itself without a finite timestamps in the future where it is surely solved. Because then a potentially unbounded number of references have to be remembered. A similar fragment has already been defined in [DSS+05], but the constraint there is stronger than ours, being a syntactical property while ours is a semantic property. Thus, the fragment defined there is a subset of ours, but for the rest of this thesis, considering the fragment as defined above is sufficient for our purposes and makes comparison easier.

Furthermore, we define $\text{LOLA}_{\text{past}}$ as the fragment of LOLA which contains only past references, completely removing future references. Obviously, it follows that $\text{LOLA}_{\text{past}}$ is a fragment of LOLA_{eff}.

For LOLA and the given fragments, we denote with LOLA^b, $\text{LOLA}^b_{\text{eff}}$ and $\text{LOLA}^b_{\text{past}}$, respectively, the corresponding fragments where only bounded data structures are allowed. This is equivalent to using only boolean values, hence we will use bounded data structures and boolean values interchangeably.

2.6.3 Types of Transducers

As mentioned before, we use transducer as the corresponding streaming formalism for automata. Transducer take an input word and produce an output symbol for each input symbol. Well known types are Moore- and Mealy-Machines. In the following we will define a type of transducers for each type of automata we defined before, which includes finite state transducer, timed transducer, and pushdown transducer. Compared to an automaton, a transducer does not depend on accepting states in general, because the output stream consisting of the output symbols is the result of a run.

First, we will start with the simplest version of a transducer: the deterministic finite state transducer. Those are very similar to DFAs or deterministic Büchi automata without an accepting condition but with output symbols on every transition.

Definition 2.39 (*Deterministic Finite State Transducer, [BB79]*)

A deterministic finite state transducer (DFST) *is a 5-tuple* $\mathcal{A} = (\Sigma, \Gamma, Q, q_0, \delta)$ *with*

- *an input alphabet* Σ,
- *an output alphabet* Γ,
- *a finite set of states* Q,
- *an initial state* $q_0 \in Q$, *and*
- *a transition function* $\delta : Q \times \Sigma \to Q \times \Gamma$.

For an input word $w = w_0 w_1 w_2 \dots$ *we call a sequence*

$$s_0 \xrightarrow{w_0/o_0} s_1 \xrightarrow{w_1/o_1} s_2 \xrightarrow{w_2/o_2} \cdots$$

a run of a DFST $\mathcal{A}$ *with output* $[\![\mathcal{A}]\!](w) = o_0 o_1 o_2 \cdots \in \Gamma^\infty$ *iff* $s_0 = q_0$ *and* $\delta(s_i, w_i) = (s_{i+1}, o_i)$ *for all* $i \geq 0$.

The language a DFST $\mathcal{A}$ *describes is the set of tuples of the input word and the word outputted by a run of the input word on* $\mathcal{A}$, *thus* $\mathcal{L}(\mathcal{A}) = \{(i, o) \mid$ *There exists a run for* i *on* $\mathcal{A}$ *such that* $[\![\mathcal{A}]\!](i) = o.\}$.

Because an acceptance condition does not exist, a DFST can run on finite and infinite words. Depending on the type of the input word, the output word is also finite or infinite, respectively.

One may question at this point why we talk of words again while the topic of these sections are streaming semantics. Traditionally, transducers are, as logics and automata, defined on words. But as said before, they are the way to go to achieve a type of streaming semantics for automata, hence a semantics that processes a stream transformation for the input stream which results in the output stream. The transducer would then react every time an event arrives on the stream, thus when the events value (possibly) changes. We will use this type of streaming behaviour later, but because transducer are already defined in a way such that they output a symbol for every input symbol directly when the input symbol occurs, we do not need to define a special semantics to get a streaming behaviour as long as we stick with finite streams, as explained in more detail later.

Next, we define a non-deterministic version of DFSTs. To represent non-determinism, we will add back accepting states and an accepting condition into these transducers. The output is then the output of an accepting run which is non-deterministically chosen.

Definition 2.40 (*Non-deterministic Finite State Transducer, [BB79]*)

A non-deterministic finite state transducer (NFST) *is a 6-tuple* $\mathcal{A} = (\Sigma, \Gamma, Q, q_0, F, \delta)$ *with*

- *an input alphabet* Σ,
- *an output alphabet* Γ,
- *a finite set of states* Q,
- *an initial state* $q_0 \in Q$,
- *a set of accepting states* F, *and*
- *a transition function* $\delta : Q \times \Sigma \to 2^{Q \times \Gamma}$.

For an input word $w = w_0 w_1 w_2 \ldots$ *we call a sequence*

$$\rho = s_0 \xrightarrow{w_0/o_0} s_1 \xrightarrow{w_1/o_1} s_2 \xrightarrow{w_2/o_2} \cdots$$

a run of a NFST $\mathcal{A}$ *with output* $[\![\mathcal{A}]\!](w) = o_0 o_1 o_2 \cdots \in \Gamma^\infty$ *iff* $s_0 = q_0$ *and* $\delta(s_i, w_i) = (s_{i+1}, o_i)$ *for all* $i \geq 0$.

A run ρ *is called* accepting *iff the last state is accepting (in case of a finite input word) or if infinitely many accepting states are visited during the run (in case of an infinite input word). Hence the acceptance condition of NFAs or NBAs is used, depending on the type of the input word.*

The language an NFST $\mathcal{A}$ *describes is the set of tuples of the input word and the word outputted by a run of the input word on* $\mathcal{A}$, *thus* $\mathcal{L}(\mathcal{A}) = \{(i, o) \mid$ *There exists an accepting run for* i *on* $\mathcal{A}$ *such that* $[\![\mathcal{A}]\!](i) = o.\}$.

NFST are a natural extension of DFSTs to non-determinism. Intuitively, NFSTs are NFAs or NBA extended to transducers, hence producing an output word in addition to the acceptance criteria. Note that NFSTs are strictly more powerful than DFSTs, as shown by the transducer in Figure 2.4.

Next, we extend DFSTs and NFSTs to be able to specify real-time properties as we did for NBAs.

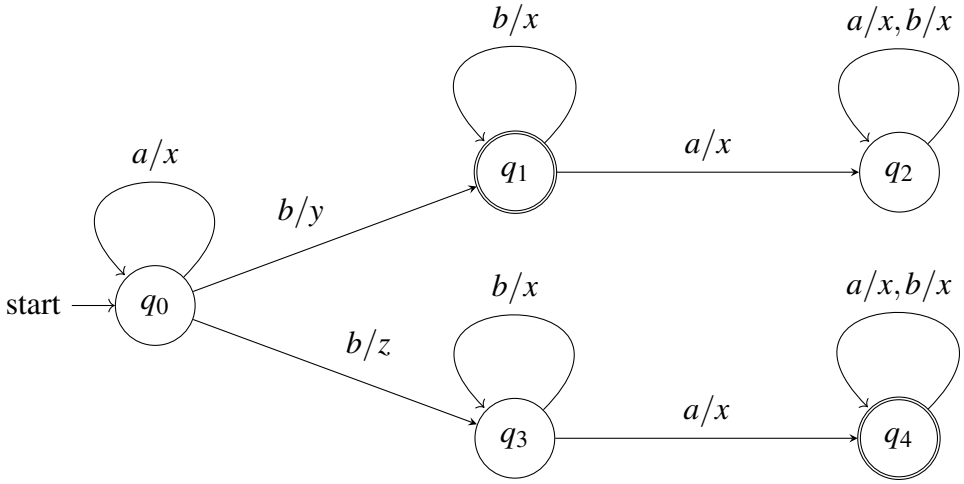


Figure 2.4: An NFST $\mathcal{A}$ over $\Sigma = \{a, b\}$ and $\Gamma = \{x, y, z\}$, for which no DFST $\mathcal{B}$ exists with $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$. The NFST $\mathcal{A}$ guesses when it reads the first b if only bs will occur afterwards or if any more as will occur. Depending on the guess, $\mathcal{A}$ outputs y or z. The output has to be generated directly when the b occurs, so there is no corresponding DFST because it can not make the decision at a later state.

We again use clocks and clock constraints and the resulting timed transducers are very much like TA. We will again define the deterministic version first, while the non-deterministic version is defined by applying the same changes as we did to get to NFSTs from DFSTs, while additionally lifting the constraint that the disjunction of two clock constraints on transitions with the same input symbol has to be unsatisfiable.

Definition 2.41 (*Timed Deterministic Finite State Transducer*)

A timed deterministic finite state transducer (DTFST) *is a 6-tuple* $\mathcal{A} = (\Sigma, \Gamma, Q, q_0, C, \delta)$ *with*

- *an input alphabet* Σ,
- *an output alphabet* Γ,
- *a finite set of states* Q,
- *an initial state* $q_0 \in Q$,
- *a set of clocks* C, *and*
- *a transition function* $\delta : Q \times \Sigma \times \Theta(C) \to Q \times 2^C \times \Gamma$ *where for any two different transitions* $(q_1, \sigma_1, \vartheta_1, q_1', R_1, \gamma_1), (q_2, \sigma_2, \vartheta_2, q_2', R_2, \gamma_2) \in \delta$ *the conjunction of their clock constraints* $\vartheta_1 \wedge \vartheta_2$ *is unsatisfiable.*

For an input word $w = (w_0,t_0)(w_1,t_1)(w_2,t_2)\dots$ *we call a sequence*

$$s_0, v_0 \xrightarrow[o_0]{(w_0,t_0),\vartheta_0,r_0} s_1, v_1 \xrightarrow[o_1]{(w_1,t_1),\vartheta_1,r_1} s_2, v_2 \xrightarrow[o_2]{(w_2,t_2),\vartheta_2,r_2} \cdots$$

a run of a DTFST $\mathcal{A}$ *where* $v_i : C \to \mathbb{R}$ *are functions mapping every clock to its current value iff*

- $s_0 = q_0$,
- $\forall c \in C : v_0(c) = 0$, *and*
- $\forall_{i\geq 0}$:
  - $\delta(s_i, w_i, \vartheta_i) = (s_{i+1}, r_i, o_i)$,
  - $\forall_{c\in r_i} v_{i+1} = v_i[c \leftarrow t_i]$, *and*
  - $t_i, v_i \models \vartheta_i$ *(which means* ϑ_i *resolved to* true *if* T *is replaced with* t_i *and the corresponding values from* v_i *are used for the clocks).*

The output of such a run is the sequence $[\![\mathcal{A}]\!](w) = (o_0,t_0)(o_1,t_1)(o_2,t_2)\dots \in (\Gamma \times \mathbb{T})^\infty$.

The language a DTFST $\mathcal{A}$ *describes is the set of tuples of the input word and the word outputted by a run of the input word on* $\mathcal{A}$, *thus* $\mathcal{L}(\mathcal{A}) = \{(i,o) \mid$ *There exists a run for* i *on* $\mathcal{A}$ *such that* $[\![\mathcal{A}]\!](i) = o.\}$.

The non-deterministic version of DTFSTs is called NTFSTs. Note again that, as for DFSTs and NFSTs, NTFSTs are strictly more expressive than DTFSTs which can be shown with a similar example as for DFSTs and NFSTs.

Lastly in this section, we define the transducer that resembles PAs. A pushdown transducer is a DFST with a stack that can be used in the same way as in PAs.

Definition 2.42 (*Deterministic Pushdown Transducer, [BB79]*)

A deterministic pushdown transducer (DPT) *is a 6-tuple* $\mathcal{A} = (\Sigma, \Gamma, Q, q_0, \Lambda, \delta)$ *with*

- *an input alphabet* Σ,
- *an output alphabet* Γ,
- *a finite set of states* Q,

- *an initial state* $q_0 \in Q$,
- *a stack alphabet* Λ *with* $\# \in \Lambda$, *and*
- *a transition function* $\delta : Q \times \Sigma \times \Lambda \to Q \times \Lambda^* \times \Gamma$.

For an input word $w = w_0 w_1 w_2 \ldots$ *we call a sequence*

$$\rho = s_0, \sigma_0 \xrightarrow[o_0]{w_0, \lambda_0, \sigma_0'} s_1, \sigma_1 \xrightarrow[o_1]{w_1, \lambda_1, \sigma_1'} s_2, \sigma_2 \xrightarrow[o_2]{w_2, \lambda_2, \sigma_2'} \cdots$$

a run *of a DPT* $\mathcal{A}$ *with output* $[\![\mathcal{A}]\!](w) = o_0 o_1 o_2 \cdots \in \Gamma^\infty$ *iff*

- $s_0 = q_0$ *and* $\sigma_0 = \#$ *(starting at one start state and with empty stack) and*
- *for all* $i \geq 0$ *the following holds:* $(s_{i+1}, \sigma_i'', o_i) = \delta(s_i, w_i, \lambda_i)$ *with*
  - $\sigma_i = \lambda_i \sigma_i'$ *(right symbol is on top of stack) and*
  - $\sigma_{i+1} = \sigma_i'' \sigma_i'$ *(new stack is output of taken transition added on top of rest of old stack).*

The language a DPT $\mathcal{A}$ *describes is the set of tuples of the input word and the word outputted by a run of the input word on* $\mathcal{A}$, *thus* $\mathcal{L}(\mathcal{A}) = \{(i, o) \mid$ *There exists a run for* i *on* $\mathcal{A}$ *such that* $[\![\mathcal{A}]\!](i) = o.\}$.

Automata as Acceptors of Transducers As already mentioned before, transducers are an extension of classical automata by adding output symbols which result in an output word for a run. But other than that, automata can also be used to simulate transducers in the sense of acceptors. This is quickly outlined in this section. We will use this technique later to conclude complexities for decision problems for certain types of transducers.

An automaton can be used as an acceptor for a transducer in the sense that different types of automata can be used to check whether an output word is the correct one for a given input word of the corresponding transducer. This can be done by giving the input and output words as input of the automaton and accepting it if it is a correct pair and otherwise rejecting it.

As a rough overview, the translation can in general be done as follows:

- Use the same states

- Merge the input and output alphabets of the transducer to a new input alphabet for the automaton
- Change outputs on edges to be additional inputs on these edges
- Make all existing states accepting and add one rejecting sink for all other edges.

This relationship between automata and transducers is interesting because we can use it later to transfer certain results for classical types of automata to transducers and thus to TeSSLa.

2.6.4 Stream Turing Machines

A Turing machine has already some parts which can be reused for defining a semantics on streams, i.e. Turing machines have tapes which can be seen as input and output tapes. Hence for a streaming semantics we remove the acceptance condition and see one of the tapes as input tape, one as work tape, and one as output tape, which contains the output after every input symbol. It is well known that having only one work tape is still enough to keep the expressive power of a Turing machine.

A similar type of Turing Machines has already been defined in [Gol00, GSAS04] as Persistent Turing Machines (PTM), but our definition of stream Turing machines here is, while equivalent in expressiveness, different, because it is more tailored to our specific definition of streams and is also taking real time explicitly into consideration.

Definition 2.43 (*Stream Turing Machines*)

Let $\Sigma = \Gamma \times \mathbb{T}$ be an alphabet (encoded in binary). A Stream Turing Machine *(STM) is a 3DTM $A = (\Sigma, Q, q_0, Q, \delta)$ where the three tapes have distinct tasks: One is an input tape the machine can only read from, one is a normal work tape, and one is the output tape, containing the output word. A run on an STM*

$$c_0 c_1 c_2 \dots c_n$$

has the following additional restrictions:

- $\forall 1 \leq i \leq n : w_1^i = w_1^0$ *(input tape can not be changed and may be left out when configurations are used),*
- *the timestamps on the input and output tape are strongly monotonic from left to right,*
- $\forall n \in \mathbb{N} : w_3^0(n) = \square$ *(output tape is empty in the beginning),*

- $n_1^0 = n_3^0 = 0$ *(input and output tape are starting left), and*
- $\forall 1 \leq i \leq n : \forall x \in \mathbb{N} : w_3^{i-1}(x) \neq \square \Rightarrow w_3^i(x) = w_3^{i-1}(x)$ *(output tape can not be changed after a value is set).*

The five restrictions on the tapes ensure an evaluation in a streaming-like manner. While the first, third, fourth, and the last restriction ensure that am STM works similar to a transducer, because neither the input nor the output tapes can be used for calculations, the second ensures the correct handling of time, which we explicitly handle in an STM, because time has to move forward in following parts of the calculation and it is necessary for certain results later in this thesis to handle time explicitly.

It is also worth noting, that even though there are five restrictions and all states are accepting, an STM can still calculate everything a normal Turing machine can. Consider a Turing machine M. First, we transform it into an equivalent Turing machine with only one tape. We can now transform this Turing machine into an STM S by doing the following:

- copy all the content on the one tape of M into the input tape of S and add arbitrary timestamps,
- copy the states and transitions of M to S, the transitions now work on the working tape of S,
- add states and transitions to S which copy the content of the input tape to the working tape, and
- add additional transitions which output a corresponding symbol if M would not accept.

We can now run S to do the same calculation as we would do on M, indicating rejection of a run by a certain output symbol. It is important to note, that while an STM is able to do the same calculations, a Turing machine still represents more functions, because it can, if timestamps are for example denoted explicitly, output timestamps in a non-monotonic order.

Finally, let us fix two properties of STMs. As noted before, these state that the semantics given for STMs are working in a way streaming semantics should, as explained earlier in this section.

Proposition 2.44 (*Basic Properties of STMs*)

STMs are monotonic and continuous in the input streams.

This proposition means that the calculation an STM does is a stream transformation.

2.7 Properties of Formalisms

In this section, we define various properties some or all of the previously defined formalisms can have. We will use or prove these later to categorize and compare different types of transducers or fragments of TeSSLa.

2.7.1 General Properties

At first, we define the term of functionality in this section. Functionality does not relate to the meaning that "it works" but instead that it represents a function, not a relation or something else. Starting with defining the functionality of instances, we will proceed with defining the functionality of formalisms.

Definition 2.45 (*Functionality of Instances*)

Let F be a formalism and f be an instance of F, which takes an input and delivers an output. We call f functional *iff for every two tuples $(i,o),(i',o') \in \mathcal{L}(f)$ it holds that $i = i' \Rightarrow o = o'$.*

Now, we extend the previous definition to whole formalisms, not just single instances of a formalism.

Definition 2.46 (*Functionality of Formalisms*)

We call a formalism F functional *iff every instance of F is functional.*

For example, a transducer is called *functional* iff for every input word, there is only one possible output word, which means that the transducer is representing a function. Deterministic transducers are always functional, while non-deterministic transducers can be functional, but are not necessarily.

Example 2.47 (*Functionality of NFSTs*)

Consider the two NFSTs $\mathcal{A}$ and $\mathcal{B}$ from Figure 2.5. Both transducers are non-deterministic, because when reading an a in q_0, there are two possible paths to take. But while $\mathcal{B}$ is functional, $\mathcal{A}$ is not. This is because in $\mathcal{A}$ the output can either be x^ω or xyx^ω when reading an input word ab^ω. On $\mathcal{B}$ on the other hand, the output is always xy^ω no matter which way is taken. Because every other input word only allows one possible way to be taken, $\mathcal{B}$ is a functional NFST.

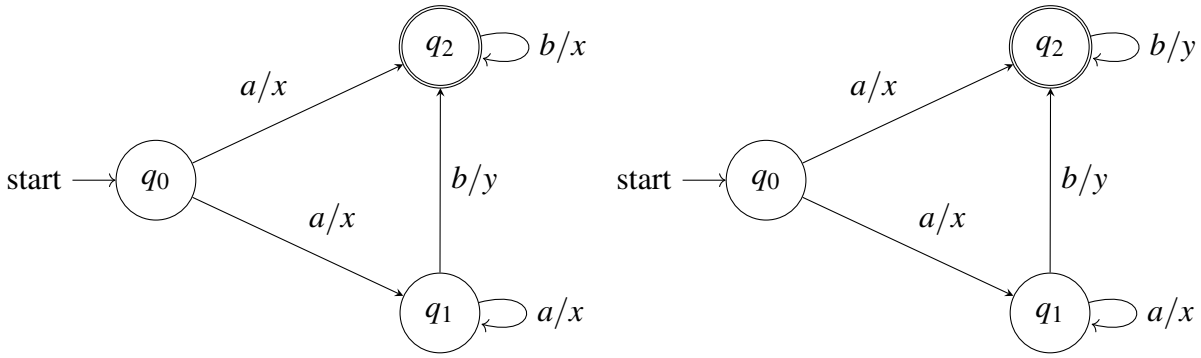


Figure 2.5: Two NFSTs: $\mathcal{A}$ on the left and $\mathcal{B}$ on the right. The only difference is that the loop on q_2 outputs an x in $\mathcal{A}$ and a y in $\mathcal{B}$.

Furthermore, note that every well-formed LOLA specification, thus also well-formed $\text{LOLA}_{\text{past}}$ and LOLA_{eff} specifications, are functional, because only one fixed-point exists [DSS+05].

2.7.2 Properties of Stream Transformations

In this section we define several properties we use later to classify TeSSLa and different fragments of it. Two properties we already defined to categorize functions in general are monotonic and continuous functions, which we defined in the beginning of this thesis and are called stream transformations when they are defined on streams. In this section we will define the additional properties of timestamp conservative and future independent stream transformations. Because these make only sense when the input has some sort of meaning over time, like a stream, we define these properties only for functions on streams.

The first property we define is timestamp conservatism. It states that every output stream of a stream transformation can only have an event at timestamps, at which at least one of its input streams had an event.

For a stream $s \in \mathcal{S}_{\mathbb{D}}$ over the time domain $\mathbb{T}$ we denote with

$$T(s) = \{t > 0 \in \mathbb{T} \mid s(t) \in \mathbb{D}\}$$

the set of timestamps present in the stream s, which means s has an event at some $t > 0$. Note that we still allow events at timestamp 0, for example for the purpose of initializing the stream. This is necessary for many calculations. For multiple streams $s_1, \ldots, s_n$ we denote with $T(s_1, \ldots, s_n) := \bigcup_{1 \leq i \leq n} T(s_i)$ the union of the timestamps of events present in any of those streams.

Definition 2.48 (*Timestamp Conservatism, [CHL+18]*)

A stream transformation $f \in \mathcal{S}_{\mathbb{D}_1} \times \ldots \times \mathcal{S}_{\mathbb{D}_n} \to \mathcal{S}_{\mathbb{D}'_1} \times \ldots \times \mathcal{S}_{\mathbb{D}'_m}$ *is called* timestamp conservative *iff it does not introduce new timestamps, i.e. for input streams* $S \in \mathcal{S}_{\mathbb{D}_1} \times \ldots \times \mathcal{S}_{\mathbb{D}_n}$ *and output streams* $S' \in \mathcal{S}_{\mathbb{D}'_1} \times \ldots \times \mathcal{S}_{\mathbb{D}'_m}$ *it holds that*

$$f(S) = S' \Rightarrow T(S) \supseteq T(S')$$

The second property is future independence. The values at a timestamp t for each of the output streams of a future independent stream transformation only depend on the values at timestamps t' on the input streams, which are equal or lower than the considered timestamp at the output streams, thus $t' \leq t$. The other way around this means, that new values on the input streams can not influence older outputs.

For a stream $s \in \mathcal{S}_{\mathbb{D}}$ we denote with

$$s|_t = \begin{cases} s & \text{if } s(t) = ? \\ s' & \text{otherwise, where } \forall t' < t : s'(t') = s(t') \wedge \forall t' \geq t : s'(t') = ? \end{cases}$$

the prefix of s with exclusive progress t. If t is greater than the current progress of s, than $s|_t$ returns s.

Definition 2.49 (*Future Independence, [CHL+18]*)

A stream transformation $f \in \mathcal{S}_{\mathbb{D}_1} \times \ldots \times \mathcal{S}_{\mathbb{D}_n} \to \mathcal{S}_{\mathbb{D}'_1} \times \ldots \times \mathcal{S}_{\mathbb{D}'_m}$ *is called* future independent *iff output events only depend on current or previous events, i.e. for input streams* $s_1, \ldots, s_n \in \mathcal{S}_{\mathbb{D}_1} \times \ldots \times \mathcal{S}_{\mathbb{D}_n}$ *and output streams* $s'_1, \ldots, s'_m \in \mathcal{S}_{\mathbb{D}'_1} \times \ldots \times \mathcal{S}_{\mathbb{D}'_m}$ *it holds that*

$$f(s_1, \ldots, s_n) = s'_1, \ldots, s'_m \Rightarrow \forall t \in \mathbb{T} : f(s_1|_t, \ldots, s_n|_t) \sqsupseteq (s'_1|_t, \ldots, s'_m|_t)$$

Informally, it states that if we cut the progress of the input streams at a timestamp t we get at least the original output until timestamp t.

The following example shows some stream transformations and their properties.

Example 2.50 (*Timestamp Conservatism and Future Independence*)

Recall the stream transformation f from Example 2.31, which was defined as follows for every

$t \in \mathbb{T}$:

$$f(s)(t) = \begin{cases} \text{tt} & \text{if } s(t) = b \wedge s(t+1) = b \\ \text{ff} & \text{if } s(t) = a \vee s(t) = b \wedge s(t+1) \neq b \\ \bot & \text{otherwise} \end{cases}$$

This stream transformation is timestamp conservative because it only outputs an event (with value tt or ff) if there was either an event with a or a b at the current timestamp. But it is not future independent because the output value at timestamp t depends on the timestamp $t+1$ in the future. However, it would be future independent if the $+1$ would be changed to -1. One could break the timestamp conservatism by, for example, letting f always output tt at timestamp 1, independently from the input (even if the input is $\bot$).

Lastly in this section, we give the notion of behavioural equivalence, which classifies stream transformations into certain equivalence classes, depending on their behaviour on infinite streams. This means, that the stream transformations which are in the same equivalence class output the same streams when getting infinite streams as input, which means streams with complete progress (hence, without ?).

Definition 2.51 (*Behavioural Equivalence*)

We say two stream transformations $f : \mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_n} \to \mathcal{S}_{\mathbb{D}'_1} \times \cdots \times \mathcal{S}_{\mathbb{D}'_m}$ *and* $f' : \mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_n} \to \mathcal{S}_{\mathbb{D}'_1} \times \cdots \times \mathcal{S}_{\mathbb{D}'_m}$ *are* behavioural equivalent *iff*

$$\forall S \in \mathcal{S}^{\infty}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}^{\infty}_{\mathbb{D}_n} : f(S) = f'(S)$$

Note that if two stream transformations are behavioural equivalent and future independent, they also produce the same events on the output streams for any input streams, even if they are not infinite streams. The only thing which may differ is the progress, hence the point at which ? starts after the last event.

2.8 Decision Problems

In this section we define all decision problems which we consider throughout this thesis. Decision problems are questions about properties of an instance and how hard it is to decide if it has the property or not. Among the considered decision problems are typical ones like equivalence but

also others which are related to the finite memory property and can be of practical relevance for a specification language.

2.8.1 The Equivalence Problem

We will start off with defining the equivalence problem. The equivalence problem is the question whether or not two given languages are equal.

Definition 2.52 (*Equivalence Problem, [HMU06]*)

> *The* equivalence problem *for a formalism F asks whether the languages $\mathcal{L}(i)$ and $\mathcal{L}(i')$ of two instances i and i' of F are equal, hence $\mathcal{L}(i) = \mathcal{L}(i')$ holds.*

In the case of DFAs, for example, the equivalence problem asks if two given DFAs A and A' accept the same language of words, which means whether $\mathcal{L}(A) = \mathcal{L}(A')$.

2.8.2 Decision Problems for Memory Usage

Questions regarding the memory usage for the evaluation of a formula is an interesting one for a specification language. Even though on a first glance, memory usage might not be so important. However, using recursive formulas can quickly consume a lot of memory, without having a chance to understand why from a user perspective. This is even more important when the formula is evaluated in an embedded system with limited memory or a specialized hardware like an FPGA. Getting certain memory guarantees will allow you to see if a formula can be evaluated on such a system and if the memory guarantees are easy to decide for a formula, it can be automatically checked if the formula is appropriate for the use case. We will use the decision problems in this section to categorize TeSSLa and different fragments by their memory guarantees later.

Before we get into the definitions of the decision problems, we will define the term of an evaluation strategy and the notion of finite memory first. An evaluation strategy for a formalism represents an algorithm which describes how to calculate the output from an input for any instance of the formalism. An evaluation strategy is supposed to do this by reading the symbols of the input incrementally (in a streaming fashion) which means that a possible output is created before the next input symbol is read. For this purpose, we assume in this thesis that the values on input streams arrive synchronously, hence every value with the same timestamp arrives at once and the

events with the next timestamp arrive only after the ones with the previous timestamp have been processed completely. Furthermore, we do not consider timing at all in the scope of evaluation strategies. Both assumptions are sufficient for the purpose of evaluation strategies in this thesis, because we only want to check if memory usage during evaluation is finite or not. For a more general approach of calculating the outputs from the inputs at the example of TeSSLa without the assumption of a synchronized arrival and processing of the values, see [LSS+18, LSS+20].

Definition 2.53 (*Evaluation Strategy for Stream Transformations*)

Let F be a formalism over stream transformations. Further, let $M = \{0,1\}^l$ with $l \in \mathbb{N} \cup \infty$ be a set of binary strings of length l and let $m : M \times (\mathbb{T} \times \mathbb{D}_1 \times \cdots \times \mathbb{D}_n) \to M$ and $o : M \times (\mathbb{T} \times \mathbb{D}_1 \times \cdots \times \mathbb{D}_n) \to (\mathbb{T} \times \mathbb{D}'_1 \times \cdots \times \mathbb{D}'_m)^k$ with $0 \leq k \in \mathbb{N}$ be two functions. We call $E = (m,o)$ an evaluation strategy *for F iff for every instance i of F and every set of input streams $S = s_1, \ldots, s_n$ with $i(s_1, \ldots, s_n) = (s'_1, \ldots, s'_m)$ and $T = \{t_1, t_2, \ldots\}$ and $T' = \{t'_1, t'_2, \ldots\}$ with $t_i < t_{i+1}$ and $t'_i < t'_{i+1}$ being the ordered sets of all timestamps in $s_1, \ldots, s_n$ or $s'_1, \ldots, s'_m$, respectively, it holds that:*

$$\forall t_x \in T : e_x = o(m(\ldots m(\varepsilon, S_1) \ldots, S_{x-1}), S_x) = \{(t', s'_1(t'), \ldots, s'_m(t')) \mid t' \in \mathcal{T} \wedge \mathcal{T} \subseteq T'\}$$

with

$$S_x = t_x, s_1(t_x), \ldots, s_n(t_x)$$

and

$$\forall t' \in T' : \exists x : (t', s'_1(t'), \ldots, s'_m(t')) \in e_x \wedge \forall (t'', s'_1(t''), \ldots, s'_m(t'')) \in e_{x+1} : t' < t''$$

We call l the amount of memory *an evaluation strategy needs.*

Informally, an evaluation strategy simply describes how the semantics of a formalism for a given instance and an input can be applied incrementally. Thereby, M is used as memory bits to save information between different input symbols. Thus, older input symbols can only be accessed later by storing them, as the function m is applied to all earlier inputs, delivering only the bits it produces as memory to the function o, which calculates an output symbol for the given memory bits and an input symbol. The last three formulas ensure that the evaluation strategy creates the correct output for every timestamp when evaluating a formula for a given input. This is necessary due to the synchronization of the input streams, which simulates, for example, additional $\perp$ values when a stream has no event, but another one does. Therefore, these do also exist in the generated output, as well as additional ? values, which have to be mapped accordingly to ensure that the evaluation

strategy generates the same output as the semantics. Informally, the first formula ensures that each output generated by an evaluation strategy just contains timestamps which the semantics would output and that the values match with the ones the output streams (in function representation) have at this timestamp. The second formula ensures a correct synchronization of the inputs by having an input for each timestamp, which existed in at least one of the input streams, adding additional $\bot$ and ? values (via function representation) on streams which do not contain this timestamp. The last formula ensures that every timestamp in the output streams the semantics deliver also occurs in the output of an evaluation strategy and that the timestamps of the outputs are ordered.

Example 2.54 (*Evaluation Strategy*)

A possible evaluation strategy for LOLA would be to simply evaluate every operator on its own, independently from the other operators it depends on, as soon as an input arrives from the operators it depends on and output the results of the evaluation. By doing so, one would get the output streams according to the semantics.

A possible evaluation strategy for $\text{LOLA}^{b}_{\text{past}}$ would be to transform a specification into an NFA as described in [BS14] and then evaluate that NFA step-by-step after transforming the input streams into an input word.

Another possibility would be to transform a LOLA specification into a semantically equivalent STM, encoding the input streams accordingly and run the STM. By interpreting the output accordingly, we would get the same result the LOLA semantics would output.

Based on the previous definition of evaluation strategies, we will define the term of finite memory. With our notion of memory usage, we only take into account the working memory, hence neither the size or length of the input or output word or stream is of importance. It is important to note that our notion of memory usage is not only about how many values are stored (variables necessary using the given evaluation strategy), but also includes whether a value can grow to a possibly unlimited size or not.

First, we define finite memory on instances of formalisms to express that an instance only needs finite memory to be evaluated. Our definitions are based on evaluation strategies and how much memory is needed for the calculation of the output for a given input, as said before, ignoring the input and output memory-wise.

Definition 2.55 (*Finite Memory of Instances*)

Let F be a formalism and i be an instance of F and let E be an evaluation strategy for F. We

> *call i* finite memory under *E iff the evaluation of i with E only needs a finite amount of memory, thus for E it holds that $l < \infty$.*

Now, we extend the previous definition to formalisms.

Definition 2.56 (*Finite Memory of Formalisms*)

> *Let F be a formalism and E be an evaluation strategy for F. We call a formalism F* finite memory under *E iff every instance of F is finite memory under E.*

As stated at the beginning of this section, we also want to consider decision problems which are related to finite memory as defined previously in this thesis. The first one is the question if, given an evaluation strategy for a formalism F, an instance of F is finite memory under E or not.

Definition 2.57 (*Finite Memory Problem*)

> *Let F be a formalism and E be an evaluation strategy for F. The* finite memory problem *(FMP) for F under E asks whether an instance i of F is* finite memory *when evaluated using E.*

As we will see in the rest of the thesis, for some formalisms, all instances are finite memory under a given evaluation strategy, but for most this is not the case. It is important to note that FMP asks whether an instance i is finite memory given a certain evaluation strategy E. This does not mean that, if i is not finite memory for E, there can not be another evaluation strategy E' under which i is finite memory. We will use this to distinguish the memory usage of intuitive evaluation strategies for TeSSLa formulas and the memory usage of an optimal one later in this thesis. In the previously described case, the question can be raised if an instance i of such a formalism, which is not finite memory under the given evaluation strategy E, and therefore $l = \infty$, can be transformed into one which accepts the same language but is finite memory under E. This would result in a new evaluation strategy E', which, for example, could rewrite the formula first and then evaluate it using E. E' now only needs $l < \infty$ memory. The general problem whether an instance can be evaluated using only finite memory is defined next.

Definition 2.58 (*Rewrite to Finite Memory Problem*)

> *The* rewritable to finite memory problem *(RFM) for a formalism F asks whether for an instance i of F a semantically equivalent STM M and an $n \in \mathbb{N}$ exist, such that M needs only at most n binary memory cells on the working tape at any point of the evaluation for any given input.*

Because the question of RFM is if there is any possibility to evaluate an instance i with only finite memory, the problem if any such evaluation strategy exists corresponds to the question if a semantically equivalent STM exists for the considered instance, which only needs a finite number of cells on the working tape for calculating the output from the input. The definition of RFM also indicates the relation of finite memory and Turing machines, which corresponds to its use of the working tape. As said before, the previous definition is equivalent to asking whether there exists an instance i' of F, which is finite memory under E and for which $\mathcal{L}(i) = \mathcal{L}(i')$ holds.

3 Temporal Stream-Based Specification Language

Contents

In this section, we are introducing the Temporal Stream-based Specification Language (TeSSLa, [CHL+18]), which is the main focus of this thesis. TeSSLa is, like LOLA, a stream transformation language which transforms a set of input streams into a set of output streams, but while LOLA is defined over discrete streams, TeSSLa is able to reason not only over discrete streams, but even over continuous streams, which leads to events that arrive at arbitrary timestamps and not every stream has events at the same timestamps.

Over the course of this section, we at first define the syntax of TeSSLa specifications and the notion of flatness, which serves as an easier representation for the later comparison of TeSSLa to transducers. After that, we give two semantics for TeSSLa formulas, one on only completed streams, which is easier to understand but not considered any further in this thesis, and the original one from [CHL+18] which is defined over prefixes and extends the semantics over completed streams by a notion of streams which are only known until a certain timestamp. Lastly in this section, we will add a completely new operator to the existing TeSSLa operators, which allows specifications with future references.

3.1 Syntax of TeSSLa

A TeSSLa specification φ is a system of equations which consists of a set of possibly mutually recursive stream definitions defined over a finite set of variables, which are either references to an equations or an input stream. Each equation consists of combinations of the six basic operators **nil**, **unit**, **lift**, **time**, **last** and **delay**, which are possibly nested.

Definition 3.1 (*TeSSLa Syntax, [CHL+18]*)

Let I be a set of input streams. Then a TeSSLa specification φ is a system of equations with equations of the form $x := e$, where the syntax of each e is given through the following grammar, with s_{ref} being a constant reference to an input stream $s \in I$, which is interpreted as a stream:

$$e ::= \textbf{nil} \mid \textbf{unit} \mid s_{ref} \mid y \mid \textbf{lift}(f)(e,\ldots,e) \mid \textbf{time}(e) \mid \textbf{last}(e,e) \mid \textbf{delay}(e,e)$$

where f can be any k-ary function and y is the left hand side of an equation of φ.

All variables not occurring on the left-hand side of equations are called *input variables* and all variables occurring on the left-hand side of an equation are called *output variables*.

In the following section, we will restrict the syntax of TeSSLa such that only one operator exists per equation. We will call such a specification *flat*.

3.1.1 Flat Specifications

In this section, we define the notion of flatness for TeSSLa. While flat specifications do not differ in expressiveness to general TeSSLa specifications, they lead to specifications where it is much easier to handle each equation when transforming it to or comparing it with other formalisms. For example, we will use flat TeSSLa specifications later, for example, to transform a TeSSLa formula into a transducer by just transforming every single equation in the flat version of the given specification into a transducer and then merging them using a composition algorithm.

In general, a specification is called *flat* if in each equation, only one TeSSLa operator occurs on the right-hand side, therefore, there is no nesting of operators per equation.

In the following definition, we will show how the syntax of TeSSLa can be restricted such that we get specifications which are flat.

Definition 3.2 (*Flat TeSSLa Specifications*)

Let I be a set of input streams. Then a TeSSLa specification φ is called flat *iff for every equation $x := e$ the right-hand side e is given through the following grammar, with s_{ref} being a constant reference to an input stream $s \in I$, which is interpreted as a stream:*

$$e ::= \textbf{nil} \mid \textbf{unit} \mid s_{\text{ref}} \mid y_1 \mid \textbf{lift}(f)(y_1, \ldots, y_n) \mid \textbf{time}(y_1) \mid \textbf{last}(y_1, y_2) \mid \textbf{delay}(y_1, y_2)$$

where f can be any k-ary function and $y_1, \ldots, y_n$ are left hand sides of equations of φ.

It is also important to note that every specification can be transformed into a flat specification by using additional variables and equations.

3.2 Semantics

A TeSSLa specification φ is a function which maps a number of input streams to a number of output streams using the TeSSLa operators mentioned before as well as stream variables in possibly recursive equations. This is quite similar to stream transformations and in fact, in the next chapter we will show that a TeSSLa specification with the semantics given in the rest of this chapter is nothing else than a stream transformation.

In the following two subsections, two different semantics will be presented. Compared to the definition in [CHL+18], where TeSSLa was defined over a semantics based on prefixes, we also give a semantics here which only works on total streams, because it is easier to define as a first step and the semantics on prefixes is directly based on this semantics on total streams.

For the rest of the thesis after this section, we will only consider the second semantics, which is defined over prefixes.

3.2.1 Semantics over Completed Streams

In this section we will define a semantics for TeSSLa which is only able to operate on completed streams. In the next section we will then give the TeSSLa semantics defined over prefixes in the way it is already defined in [CHL+18]. Compared to the prefix semantics, the one over completed streams is easier for a first understanding of the operators because fewer cases exist, ignoring the

case that streams may only have a certain finite timestamp until which they progressed, therefore, until which their values are known. The prefix semantics is then only different in that it can handle input streams without infinite progress. If all input streams have progress at a given point in time, both semantics are equivalent for that timestamp.

As mentioned before, a TeSSLa specification consists of a collection of stream variables and possibly recursive equations over these variables using the TeSSLa operators. We will give the general semantics first and the semantics for each single operator afterwards.

Definition 3.3 (*TeSSLa Semantics over Completed Streams*)

Let φ be a TeSSLa specification. Then the semantics of the TeSSLa specification φ over completed streams is the semantic function $[\![\varphi]\!]_\infty : \mathcal{S}^\infty_{\mathbb{D}_1} \times \cdots \times \mathcal{S}^\infty_{\mathbb{D}_n} \to \mathcal{S}^\infty_{\mathbb{D}'_1} \times \cdots \times \mathcal{S}^\infty_{\mathbb{D}'_m}$ and we write $[\![\varphi]\!]_\infty(I) = O$, where O is the set of output streams after the equations have been evaluated using the input streams in I. The semantics for the system of equations that is φ is given as the fixed-point of the equations interpreted as a function of the stream variables and fixed input streams. The semantics for each single operator is given below.

The language a TeSSLa specification φ represents is given as $\mathcal{L}(\varphi) = \{(I,O) \mid [\![\varphi]\!]_\infty(I) = O\}$.

In the following, we abuse notation by interpreting a TeSSLa formula φ directly as a function and write $\varphi(I) =_\infty O$ for $[\![\varphi]\!]_\infty(I) = O$.

We will now define the semantics of the six TeSSLa operators over completed streams, from which the equations can be build. For the rest of the thesis, we denote $\mathbb{D}^\perp := \mathbb{D} \cup \{\perp\}$.

The first operator is **nil**, which is the stream without any events.

Definition 3.4 (nil *Operator over Completed Streams*)

The **nil** $\in \mathcal{S}^\infty_\emptyset$ *operator over completed streams is defined as follows:*

$$\mathbf{nil} = \infty$$

Recall that the stream consisting of only the single letter ∞ represents the stream that has no events. Thus, for every point in time, **nil** returns $\perp$ and as a function, it is defined as $\forall t \in \mathbb{T} : \mathbf{nil}(t) = \perp$.

Example 3.5 (nil *Operator*)

As a stream picture, the **nil** operator looks as follows:

The second operator is **unit**, which represents the stream with an unit event at timestamp 0 and no other events.

Definition 3.6 (unit *Operator over Completed Streams*)

The **unit** $\in \mathcal{S}^{\infty}_{\mathbb{U}}$ *operator over completed streams is defined as follows:*

$$\textbf{unit} = 0 \,\square\, \infty$$

When **unit** is seen as a function, it holds that $\textbf{unit}(0) = \square$ and $\forall 0 < t \in \mathbb{T} : \textbf{unit}(t) = \bot$.

Example 3.7 (unit *Operator*)

As a stream picture, the **unit** operator looks as follows:

The third operator is **time**, which maps the value of each event on a stream to the event's timestamps and stays $\bot$ if no event occurred at a timestamp.

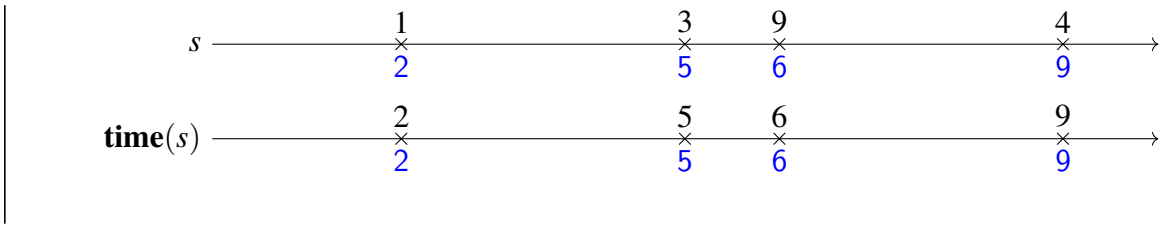
Definition 3.8 (time *Operator over Completed Streams*)

The **time** $: \mathcal{S}^{\infty}_{\mathbb{D}} \to \mathcal{S}^{\infty}_{\mathbb{T}}$ *operator over completed streams is defined as* $\textbf{time}(s) := z$*, where* z *is defined at every timestamp* $t \in \mathbb{T}$ *as follows:*

$$z(t) = \begin{cases} t & \text{if } t \in \mathit{ticks}(s) \\ \bot & \text{otherwise} \end{cases}$$

Example 3.9 (time *Operator over Completed Streams*)

For a given input stream $s \in \mathcal{S}^{\infty}_{\mathbb{N}}$, the **time** operator looks as follows:



While the first three operators we defined, **nil**, **unit** and **time**, more or less represent single streams or just map the timestamps to their values, the following three operators are those that do calculations on streams, refer to older values or set timeouts for outputting events in the future. The first of those is **lift**, which allows for lifting a function on values to a function on streams by applying the function to all values of the events on the streams for every timestamp. Before we get to the definition of **lift**, we define a certain type of functions.

Definition 3.10 (*Non-creating Functions*)

We call a function $f : \mathbb{D}_1^\perp \times \cdots \times \mathbb{D}_n^\perp \to \mathbb{D}^\perp$ non-creating *iff the following holds:*

$$f(\perp, \ldots, \perp) = \perp$$

A non-creating function does not output a data values if neither of its input stream contains a data value, therefore, it does not does create a new data value out of nowhere.

Definition 3.11 (lift *Operator over Completed Streams*)

Let $f : \mathbb{D}_1^\perp \times \cdots \times \mathbb{D}_n^\perp \to \mathbb{D}^\perp$ *be a non-creating function on n data values. Then the* **lift** $: (\mathbb{D}_1^\perp \times \cdots \times \mathbb{D}_n^\perp \to \mathbb{D}^\perp) \to (\mathcal{S}^\infty_{\mathbb{D}_1} \times \cdots \times \mathcal{S}^\infty_{\mathbb{D}_n} \to \mathcal{S}^\infty_{\mathbb{D}})$ *operator over completed streams is defined as* $\mathbf{lift}(f)(s_1, \ldots, s_n) := z$*, where z is defined at every timestamp* $t \in \mathbb{T}$ *as follows:*

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

This operator is used to do arbitrary calculations on streams by using functions which are defined over values. This means that **lift** has no knowledge about values at previous timestamps, just taking into account the current values on all streams.

The allowed functions for f are restricted to the ones that do not generate new events, because otherwise one would be able to create arbitrary streams using the **lift**, which would concentrate too much expressive power in the **lift** operator and eliminate the clear separation of the tasks each

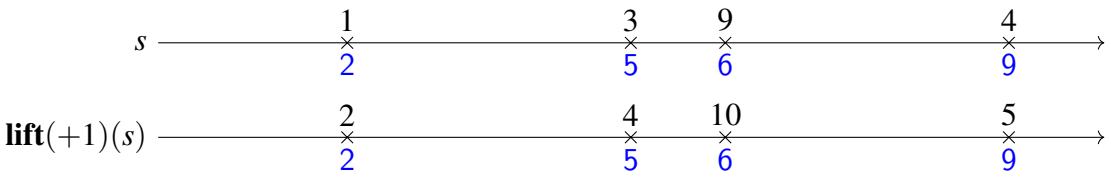
operator fulfils in TeSSLa. Other than that, TeSSLa is working with all kinds of functions, and a TeSSLa specification is possibly not computable if a function used is not computable. But it does not make much sense to allow such functions, thus for the rest of the paper, we assume that functions used in any **lift** are computable unless explicitly noted otherwise, even though, in general, TeSSLa would work even when using non-computable functions.

Example 3.12 (*Incrementation over Completed Streams*)

Let $+1 : \mathbb{N}^{\perp} \to \mathbb{N}^{\perp}$ be a function that increments its input as follows:

$$+1(n) = \begin{cases} n+1 & \text{if } n \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

Incrementation as a stream operation is then done as $\mathbf{lift}(+1)(s)$, where $s \in \mathcal{S}^{\infty}_{\mathbb{N}}$ is an input stream. For a given input stream s, this looks as follows:



The following operator is one called **last**. This operator takes two streams. One of those is a stream v from which the values are taken, which we will call *value stream* and the other is a stream r whose events mark the timestamps at which $\mathbf{last}(v,r)$ generates an output event with the previous value on the value stream if one exists. We will call the stream r the *trigger stream*.

Definition 3.13 (last *Operator over Completed Streams*)

The **last** $: \mathcal{S}^{\infty}_{\mathbb{D}} \times \mathcal{S}^{\infty}_{\mathbb{D}'} \to \mathcal{S}^{\infty}_{\mathbb{D}}$ *operator over completed streams is defined as* $\mathbf{last}(v,r) := z$*, where* z *is defined at every timestamp* $t \in \mathbb{T}$ *as follows:*

$$z(t) = \begin{cases} d & t \in \mathit{ticks}(r) \text{ and } \exists t' < t : \mathit{isLast}(t,t',v,d) \\ \perp & \mathit{otherwise} \end{cases}$$

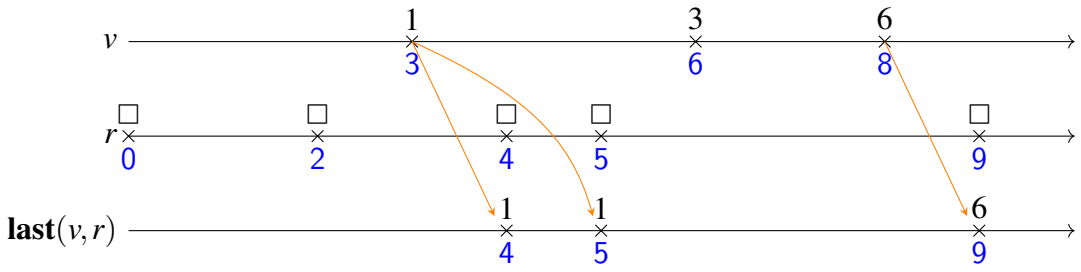
where

$$\mathit{isLast}(t,t',v,d) \Leftrightarrow v(t') = d \wedge \forall t' < t'' < t : v(t'') = \perp$$

The predicate *isLast* is used to check if the value to be outputted is really the last one that occurred on v. It holds if $t'd$ is the last event on v until timestamp t. Informally, **last** allows us to access previous data values, which works by always remembering the last value that occurred on v and outputting the remembered value if an event on r occurs.

Example 3.14 (last *Operator over Completed Streams*)

For two given input streams $v : \mathcal{S}^{\infty}_{\mathbb{N}}$ and $r : \mathcal{S}^{\infty}_{\mathbb{U}}$, the **last** operator looks as follows:



The final operator defined for TeSSLa is **delay**, which takes two streams. One of those is a stream d which represents the time in the future that the event to be outputted has to be delayed and the other is a stream r whose events represent resets for the delay. Informally, **delay** emits a unit event in the resulting stream after the delay passes when no reset occurs in between. 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.

Definition 3.15 (delay *Operator over Completed Streams*)

The **delay** $: \mathcal{S}^{\infty}_{\mathbb{T}\setminus\{0\}} \times \mathcal{S}^{\infty}_{\mathbb{D}} \to \mathcal{S}^{\infty}_{\mathbb{U}}$ *operator over completed streams is defined as* $\mathbf{delay}(d,r) := z$, *where* z *is defined at every timestamp* $t \in \mathbb{T}$ *as follows:*

$$z(t) = \begin{cases} \square & \exists t' < t : d(t') = t - t' \wedge setable(z,r,t') \wedge noreset(r,t',t) \\ \bot & otherwise \end{cases}$$

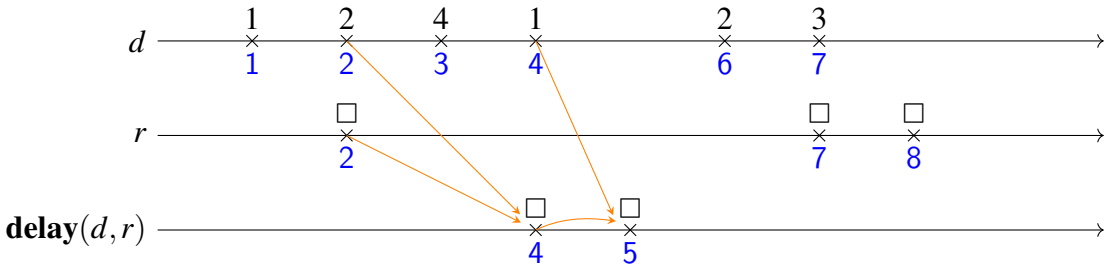
where

$$setable(z,r,t') \Leftrightarrow z(t') = \square \vee t' \in ticks(r)$$
$$noreset(r,t',t) \Leftrightarrow \forall t' < t'' < t : r(t'') = \bot$$

The predicate *setable* checks if for a given timestamp, either a reset or an output event occurred, because only then a new delay should be able to be set. On the other hand, *noreset* checks if between two given timestamps, no reset occurred, such that the delay times out without being cancelled.

Example 3.16 (delay *Operator over Completed Streams*)

For two given input streams $d \in \mathcal{S}^{\infty}_{\mathbb{N}}$ and $r \in \mathcal{S}^{\infty}_{\mathbb{U}}$, the **delay** operator looks as follows:



The orange arrows show how the output events are created. In case of the events at timestamps 1 and 3 and 6, *setable* is not fulfilled because neither an event on the reset stream not an event on the output stream exists at these timestamps. For the event at 7, *noreset* is not fulfilled, because a reset occurs before the event would be outputted.

As a conclusion for all the six defined operators, **nil** and **unit** refer to constant streams which are used to initialize non-input streams in TeSSLa. Depending on what the stream should look like, one or the other operator is necessary. The **time** operator makes time a first class citizen in TeSSLa. It is the only way to access timestamps in TeSSLa and use them as data values. The timestamps itself can not be modified, they have to be copied to the values of the events to do arbitrary calculations with them. On the other hand, **lift** is the operator which is used to lift arbitrary functions on values to streams, such that calculations on the data values of the streams can be done. Arbitrary functions on arbitrary data domains can be lifted to functions on streams of these data domains using **lift**, which can then be applied to corresponding streams. By doing so, the calculation represented by the lifted function is applied to all events with the same timestamp on these streams and output values with the corresponding timestamp are produced, which results in a new stream as a result. Compared to the previous operators, **last** and **delay** are the temporal operators in TeSSLa. **last** is used to access the last value of an event on a stream when an event on the trigger stream occurs. This enables TeSSLa specifications to refer to older values on streams. On the other hand, **delay** is used to set a delay in the future to a point where an event is outputted when the delay times

out, which can be used, for example, as a deadline to check if something happens in a real-time constraint.

Having defined the six basic operators above we can now use them to derive the following utility functions, which we use later in this thesis as a shorthand for the specification they represent. Of course, we don't need to define these functions, but the specifications later in this thesis are much easier to read by using them, because they represent typical patterns and combinations of basic TeSSLa operators. The following examples show how typical functions can be derived from the six core operators and at the same time, define functions we will use later.

Example 3.17 (*Constant Values in TeSSLa*)

The first function we consider is $\mathsf{const} : \mathbb{D} \times \mathcal{S}^{\infty}_{\mathbb{D}'} \to \mathcal{S}^{\infty}_{\mathbb{D}}$ which maps the values of all events on a stream s to the constant c. This function is defined as follows:

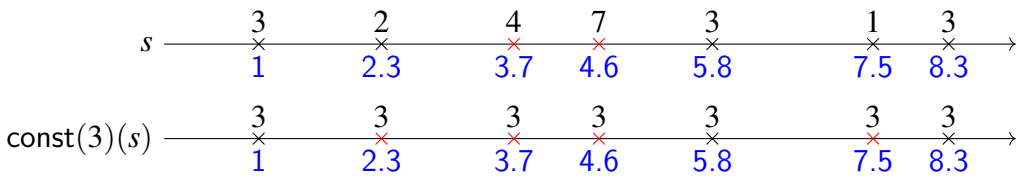
$$\mathsf{const}(c)(s) := \mathbf{lift}(f_c)(s)$$

with

$$f_c(d) := \begin{cases} c & \text{if } d \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

The function f_c, which is parameterized by a constant, takes care of mapping every event's value to the given constant c.

The following stream picture shows the usage of const on a stream s and a constant 3:



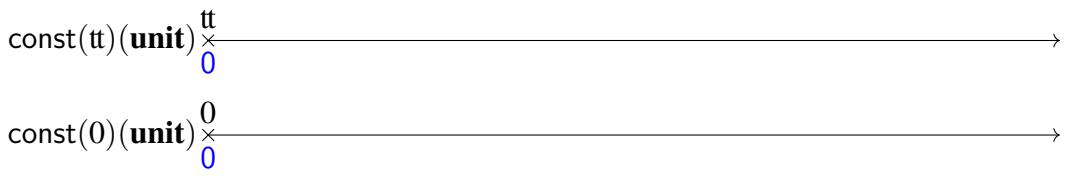
The numbers of the events in red show which values have been changed. In the end, all values are 3. Additionally, we can use this to create streams for arbitrary constants, like a true or zero stream as

$$\mathsf{true} := \mathsf{const}(\mathsf{tt})(\mathbf{unit})$$

or

$$\mathsf{zero} := \mathsf{const}(0)(\mathbf{unit})$$

which look like



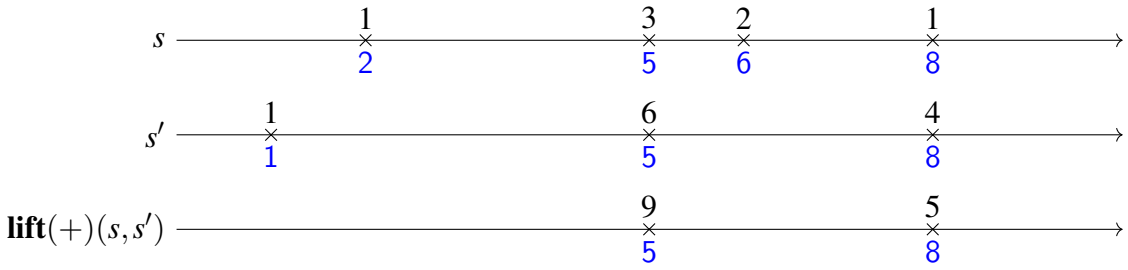
Next we show how standard operations can be done using **lift**, like addition, subtraction, division or boolean operations like and, or or.

Example 3.18 (*Standard Operations in TeSSLa*)

Let again $\circ : \mathbb{D}^{\perp} \times \mathbb{D}^{\perp} \to \mathbb{D}^{\perp}$ with $\circ \in \{+, -, /, \wedge, \vee\}$ be a function that executes a standard operation on its input as follows:

$$\circ(n,m) = \begin{cases} n \circ m & \text{if } n \neq \perp \wedge m \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

Addition as a stream operation is then done as $\mathbf{lift}(+)(s,s')$, where $s, s' \in \mathcal{S}_{\mathbb{N}}$ are input streams. For given input streams s and s', this looks as follows:



The following two examples show how one can combine two streams into one or how to filter events out of a stream, depending on a given condition.

In the first example, again the **lift** operator is used with a corresponding function being lifted such that the two input streams of the **lift** are merged into one by copying the events of the two input streams to the single output stream.

Example 3.19 (*Merging Streams in TeSSLa*)

The next function we consider is merge : $\mathcal{S}^{\infty}_{\mathbb{D}} \times \mathcal{S}^{\infty}_{\mathbb{D}} \to \mathcal{S}^{\infty}_{\mathbb{D}}$ which merges the events of two streams over the same data domain into one stream of this data domain. If both streams have an event at

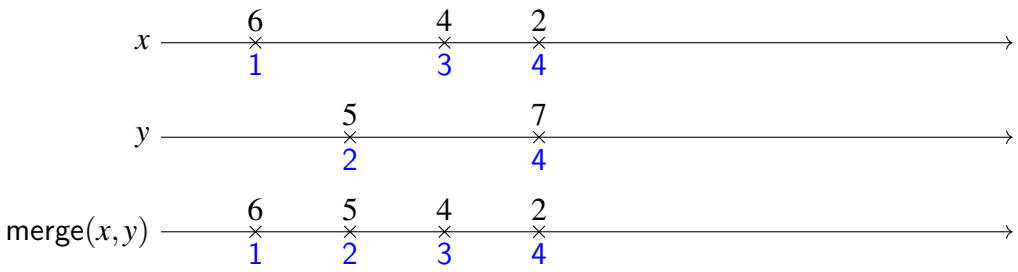
the same timestamp, the event on the first parameter stream is prioritized. This function is, for example, often needed to start recursions. Formally, it is defined as

$$\mathsf{merge}(x,y) := \mathbf{lift}(f)(x,y)$$

with

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

The function f takes care of the merging for a given point in time by either taking the value of a if there is one or b otherwise. For two given streams x and y, merge works as follows:



The second example also uses a **lift** to implement a filtering function, but this time also uses a **last**. The idea of the filtering is that as long as the condition is or was true before, events on the stream to be filtered are left through. To implement that the prior fulfilment of the condition is still used at a current timestamp without having an event, the **last** is used.

Example 3.20 (*Filtering Events in TeSSLa*)

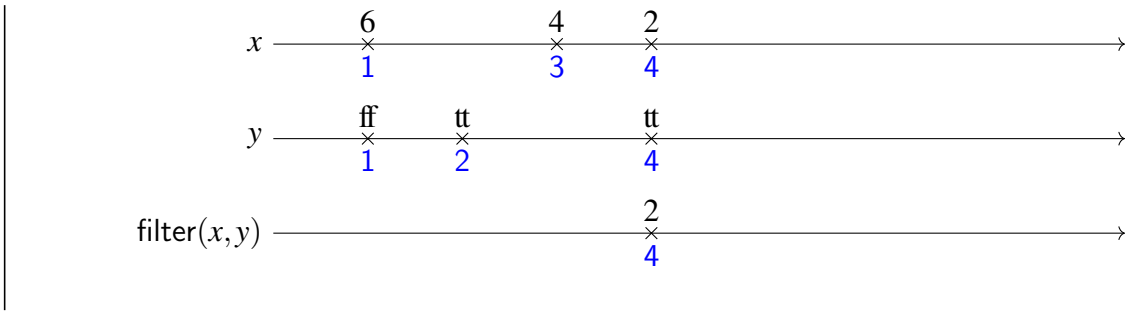
The next function we consider is $\mathsf{filter} : \mathcal{S}^{\infty}_{\mathbb{D}} \times \mathcal{S}^{\infty}_{\mathbb{B}} \to \mathcal{S}^{\infty}_{\mathbb{D}}$ which filters the events of a stream depending on the truth values on the second input stream. If the second stream is true, the event is passed to the output stream, otherwise it is erased. Formally, it is defined as

$$\mathsf{filter}(x,y) := \mathbf{lift}(f)(x, \mathsf{merge}(y, \mathbf{last}(y,x)))$$

with

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

The function f takes care of the filtering for a given point in time by either taking the value of a if b is true or outputting $\bot$ otherwise. For two given streams x and y, filter works as follows:



The last utility function can be used to allow TeSSLa to interpret streams in a slightly different model. Even though the streams TeSSLa operates on are piecewise constant, therefore its events can only occur at distinct timestamps, even if the time domain is $\mathbb{R}$, it is able to resemble streams having a values continuously, therefore no $\perp$ between two events. These streams are piecewise linear and are normally called *signals*. A typical example of a signal is a Sinus curve, which can not be represented with a finite number of events.

Of course, because TeSSLa is only defined over piecewise constant streams in this thesis, it can not express or handle a Sinus curse (this can be changed, but would go beyond the scope of this thesis). But one could rebuild this signal view for piecewise constant streams in TeSSLa using **lift** and **last**, such that it is assumed that a value is on a stream until a new one arrives, interpreting all positions with $\perp$ between two events as if they have the value of the previous event. This does still not allow TeSSLa as defined in this thesis to work on a Sinus curve, but the idea that a values stays on a stream until a newer value arrives can be represented like this.

While the defined **lift** operation alone only takes into account the events that happen at the same timestamp, we can implement a signal semantics, called slift, in TeSSLa which means that if at some timestamp there is an event on one stream but no event on another, the last events values are used for those streams that have no event at the timestamp. This represents that events values stay on a stream continuously until a new events arrives, overriding the previous events value. This behaviour is explained in the following example.

Example 3.21 (*Signal Semantics in TeSSLa*)

Using **last** one can query the last known value of an event stream s and interpret the events on s as points where a piece-wise constant signal changes its value. By combining the **last** and **lift** operators, we can realize the mentioned slift as follows, where we require that $f : \mathbb{D} \times \mathbb{D}' \to \mathbb{D}''$ is a total function (can not output $\perp$):

$$\mathsf{slift}(f)(x,y) := \mathbf{lift}(g_f)(x',y')$$

with

$$x' := \mathsf{merge}(x, \mathbf{last}(x, y))$$

and

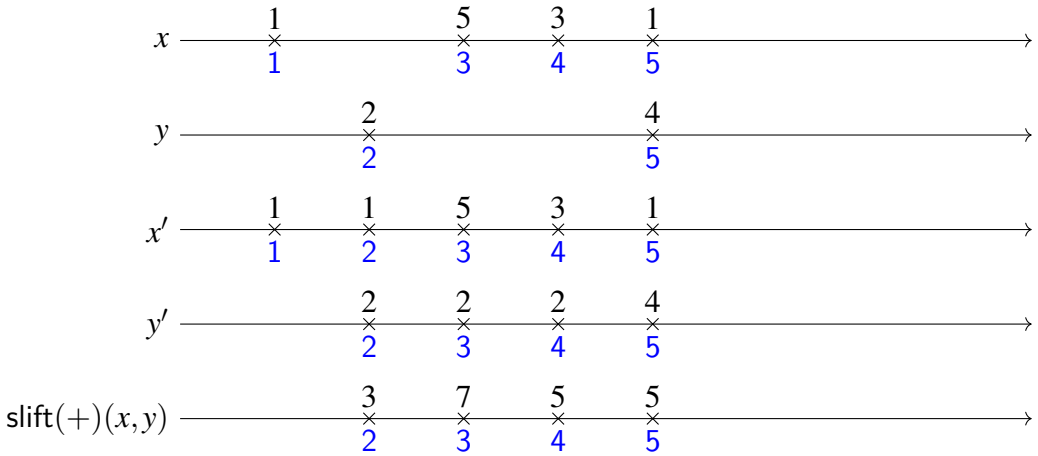
$$y' := \mathsf{merge}(y, \mathbf{last}(y, x))$$

as well as

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

The two streams x' and y' are used to always get the value of the previous event on x and y, respectively, if one stream has an event, but the other has none, such that we always have two values and implement the mentioned signal semantics. The function f is then calculating a new value out of the given ones, while g_f takes care of the case where one of the two streams did not have any event at all, such that still $\bot$ exists on either x' or y'.

For two given streams x and y and the standard addition $+$, the slift works as follows:



The next example shows a more complex specification and what the results of the evaluation of every single equation is. The idea is to sum up the values of the events occurring on an input stream while a second input stream resets the sum to zero if an event occurs there, after which the summing up starts again.

We will use the following example as a running example in different places in this thesis to show the differences of the semantics and other approaches. While the example is not too complicated, it shows the features and possibilities stream languages deliver very well.

Example 3.22 (*TeSSLa Specification*)

We can now specify the stream transformation shown in Figure 3.1 in TeSSLa. Informally, the specification shows how the values of a stream can be summed up while every event on a second stream may reset the current sum to 0.

Let $\mathsf{resets} \in \mathcal{S}^{\infty}_{\mathbb{U}}$ and $\mathsf{values} \in \mathcal{S}^{\infty}_{\mathbb{Z}}$ be two input event streams. We then derive $\mathsf{cond} \in \mathcal{S}^{\infty}_{\mathbb{B}}$ and $\mathsf{lst}, \mathsf{sum} \in \mathcal{S}^{\infty}_{\mathbb{Z}}$ as follows:

$$\begin{aligned} \mathsf{cond} &= \mathsf{slift}(\leq)(\mathbf{time}(\mathsf{resets}), \mathbf{time}(\mathsf{values})) \\ \mathsf{lastsum} &= \mathsf{merge}(\mathbf{last}(\mathsf{sum}, \mathsf{values}), \mathsf{zero}) \\ \mathsf{sum} &= \mathsf{slift}(f)(\mathsf{cond}, \mathsf{lastsum}, \mathsf{values}) \end{aligned}$$

where

$$f : \mathbb{B} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \text{ with}$$
$$f(c, l, v) = \begin{cases} 0 & \text{if } c = \text{true} \\ l + v & \text{otherwise} \end{cases}$$

This specifies the summation of the stream of numbers values which is reset to 0 (because zero is the macro $\mathsf{zero} = \mathsf{const}(0, \mathsf{unit})$) if an event on resets occurs. cond represents the reset condition which is the question if the last event on resets occurred later than the one on values. The stream lst is used to recursively sum up the values and to add a 0 as starting value. Because the lifted function f resembles an if-then-else, sum is then the result stream, which resets to 0 if cond is true and adds a new value to the current sum otherwise.

We added the semantics over completed streams in this thesis to show how such a semantics would work for TeSSLa and to explain how the semantics from the original paper [CHL+18] relates to such a semantics over completed streams. But as for this paper, we will mainly stick to the semantics defined via prefixes, which is defined in the following section.

3.2.2 Prefix Semantics

In this section, we define the prefix semantics, which corresponds to the semantics as defined for TeSSLa in [CHL+18, LSS+19]. These are the semantics we consider throughout the rest of this thesis, unless indicated otherwise.

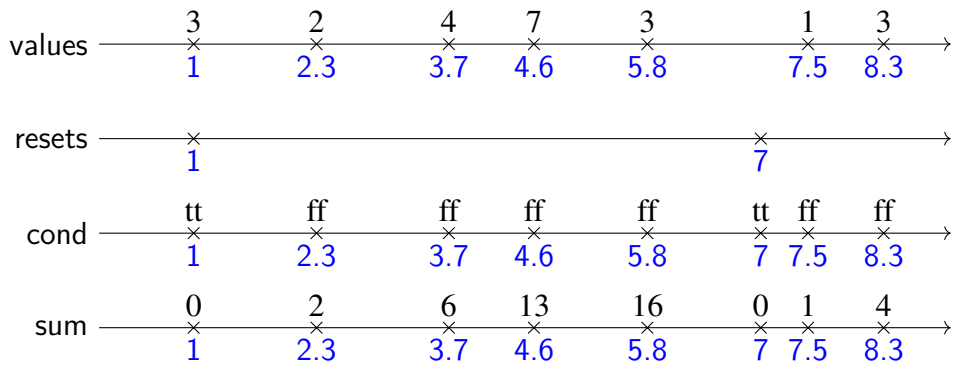


Figure 3.1: Example trace for a TeSSLa specification with two input streams values (with numeric values) and resets (with no internal value). The intention of the specification is to accumulate all values since the last reset in the output stream sum. The intermediate stream cond is derived from the input streams indicating if reset has currently the most recent event, and thus the sum should be reset to 0.

Compared to the semantics over completed streams defined in the last section, the prefix semantics takes an additional fact into account, namely that streams can have an ending. An ending in our setting does not mean that no event is coming from there on, but instead that we do not know any more what happens after the ending, whether events occur in the future and which values these may have. So the input streams for a TeSSLa formula over these semantics are event streams which are either completed streams as before, or streams with an unknown suffix, hence a prefix of a completed stream.

As before, a TeSSLa specification still consists of a collection of stream variables and possibly recursive equations over these variables using the TeSSLa operators but for the prefix semantics a TeSSLa specification resembles a function $\varphi : \mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_n} \to \mathcal{S}_{\mathbb{D}'_1} \times \cdots \times \mathcal{S}_{\mathbb{D}'_m}$.

Definition 3.23 (*Prefix TeSSLa Semantics, [CHL+18]*)

Let φ be a TeSSLa specification. Then the semantics of the TeSSLa specification φ with equations $y_1 := e_1, \ldots, y_n := e_n$ and input streams I over event streams is the semantic function $[\![\varphi]\!] : \mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_n} \to \mathcal{S}_{\mathbb{D}'_1} \times \cdots \times \mathcal{S}_{\mathbb{D}'_m}$ and we write $[\![\varphi]\!](I) = O$, where O is the set of output streams after the equations have been evaluated using the input streams in I. The semantics for the system of equations that is φ is given as the least fixed-point of the equations interpreted as a function of the stream variables and fixed input streams as follows:

$$[\![\varphi]\!](I) = \mu([\![e_1]\!](I), \ldots, [\![e_n]\!](I))$$

The semantics for each single operator is given below.

The language a TeSSLa specification φ represents is given as $\mathcal{L}(\varphi) = \{(I,O) \mid [\![\varphi]\!](I) = O\}$.

The ? value available on event streams, which are the stream model in the prefix semantics, is used to denote positions that are beyond the current ending of knowledge on the stream, which allows these semantics to evaluate streams incrementally, hence while the data is coming, in practice. This fact will be considered in more detail at the end of this section.

We again abuse notation and interpret a TeSSLa formula φ directly as a function and therefore write $\varphi(I) = O$ for $[\![\varphi]\!](I) = O$ for the rest of this thesis.

We will now define the prefix semantics of the six TeSSLa operators from which the equations can be build.

Definition 3.24 (nil *Operator on Event Streams, [CHL+18]*)

The **nil** $\in \mathcal{S}_\emptyset$ *operator on event streams is defined as follows:*

$$\mathbf{nil} = \infty$$

As for the semantics over completed streams, it will return $\bot$ for every point in time and as a function, it is defined as $\forall t \in \mathbb{T} : \mathbf{nil}(t) = \bot$.

Definition 3.25 (unit *Operator on Event Streams, [CHL+18]*)

The **unit** $\in \mathcal{S}_\mathbb{U}$ *operator on event streams is defined as follows:*

$$\mathbf{unit} = 0 \,\square\, \infty$$

As for **nil**, this operator also stays the same in both semantics, hence, when **unit** is seen as a function is holds that $\mathbf{unit}(0) = \square$ and $\forall 0 < t \in \mathbb{T} : \mathbf{unit}(t) = \bot$.

Next, we define **time** on event streams, which maps the value of each event on a stream to the event's timestamps and stays $\bot$ or ?, respectively, if no event occurred at a timestamp. Compared to the semantics over completed streams, the only change is that now also the newly existing ? values, therefore the point where the progress ends, are copied instead of only the $\bot$ values.

Definition 3.26 (time *Operator on Event Streams, [CHL+18]*)

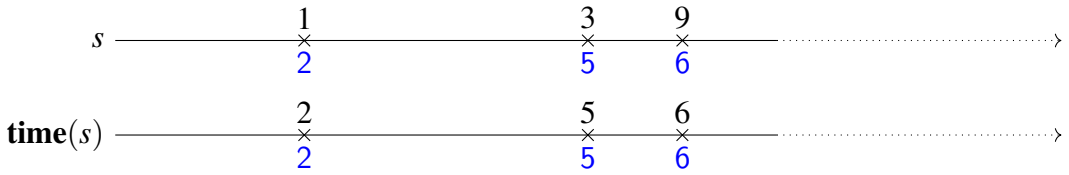
The **time** $: \mathcal{S}_\mathbb{D} \to \mathcal{S}_\mathbb{T}$ *operator over event streams is defined as* $\mathbf{time}(s) := z$*, where z is defined*

at every timestamp $t \in \mathbb{T}$ as follows:

$$z(t) = \begin{cases} t & \text{if } t \in \mathit{ticks}(s) \\ z(t) & \text{otherwise} \end{cases}$$

Example 3.27 (time *Operator over Event Streams*)

For a given input stream $s \in \mathcal{S}_{\mathbb{N}}$, the **time** operator looks as follows:



Compared to **time** over completed streams from Example 3.9, we now have the case where the progress ended, at timestamp 7 in this example. The **time** operator on event streams just copies this behaviour.

Compared to the semantics over completed streams, a case for z is added for the **lift** operator when one of the streams is ?. Because we can then not infer which value the stream will have in the future, the whole **lift** returns ?. We use the version from [LSS+19] instead of the original one from [CHL+18], because its definition is easier but semantically the same.

Definition 3.28 (lift *Operator on Event Streams, [LSS+19]*)

Let $f : \mathbb{D}_{1\perp} \times \cdots \times \mathbb{D}_{n\perp} \to \mathbb{D}_{\perp}$ be a non-creating function on n data values. Then the **lift** *: $(\mathbb{D}_{1\perp} \times \cdots \times \mathbb{D}_{n\perp} \to \mathbb{D}_{\perp}) \to (\mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_n} \to \mathcal{S}_{\mathbb{D}})$ operator over event streams is defined as $\mathbf{lift}(f)(s_1, \ldots, s_n) := z$, where z is defined at every timestamp $t \in \mathbb{T}$ as follows:*

$$z(t) = \begin{cases} f(s_1(t), \ldots, s_n(t)) & \text{if } s_1(t) \neq ? \wedge \cdots \wedge s_n(t) \neq ? \\ ? & \text{otherwise} \end{cases}$$

Note that, even though one can think of cases where **lift** could not output ? even if one of the streams is ?, like for an if-then-else where the stream is ? which is currently not the stream used for the output value because of the condition, **lift** would still output ?. This is because the **lift** can

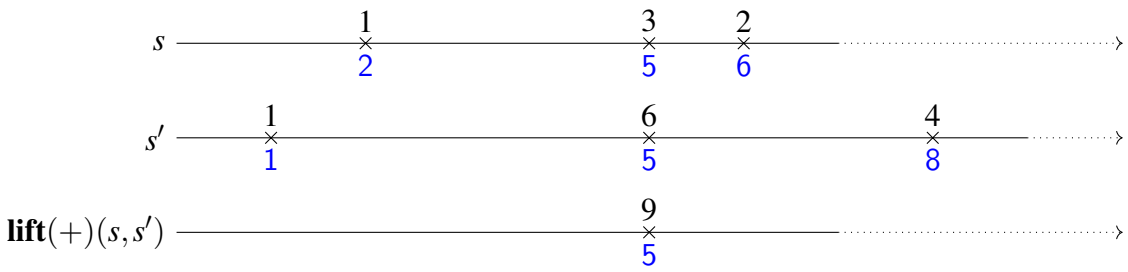
not look into the functions and because the functions have no state, the **lift** has to take care of the ? output to keep the premise that once a stream is ?, it is so forever. Therefore, because it is an invariant on event streams that once ? occurred, it will not change in the future, the **lift** has to take care of this case globally, instead of leaving it to the function which is lifted. Otherwise, using an if-then-else as described before, it could output other values after a ? was outputted before.

Example 3.29 (*Addition over Event Streams*)

Let again $+ : \mathbb{N}^{\perp} \times \mathbb{N}^{\perp} \to \mathbb{N}^{\perp}$ be a function that adds the input as follows:

$$+(n,m) = \begin{cases} n+m & \text{if } n \neq \perp \wedge m \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

Addition as a stream operation is then done as $\mathbf{lift}(+)(s,s')$, where $s,s' \in \mathcal{S}_{\mathbb{N}}$ is an input stream. For given input streams s and s', this looks as follows:



Compared to Example 3.18, we also consider the end of progress this time to depict the interplay between two input streams in the prefix semantics for **lift**. One can see that, if the progress on one input stream ends, the progress on the output stream does also end, no matter what the other stream does. The lifted function, $+$ in this case, has no influence on that, because ? is handled by the **lift** itself.

As for the other operators, for the **last** operator a ? case is added for the prefix semantics and an additional assistance function *defined* which checks if a ? was outputted before. *defined* holds at a timestamp t if z is defined (has no ?) until t (exclusive). Other than that, the definition of **last** stays the same as before.

Definition 3.30 (last *Operator on Event Streams, [CHL+18]*)

The **last** $: \mathcal{S}_{\mathbb{D}} \times \mathcal{S}_{\mathbb{D}'} \to \mathcal{S}_{\mathbb{D}}$ *operator over event streams is defined as* $\mathbf{last}(v,r) := z$*, where* z *is*

defined at every timestamp $t \in \mathbb{T}$ *as follows:*

$$z(t) = \begin{cases} d & t \in ticks(r) \text{ and } \exists t' < t : isLast(t,t',v,d) \\ \bot & r(t) = \bot \text{ and } defined(z,t), \text{ or } \forall t' < t : v(t') = \bot \\ ? & otherwise \end{cases}$$

where

$$isLast(t,t',v,d) \Leftrightarrow v(t') = d \wedge \forall t' < t'' < t : v(t'') = \bot$$

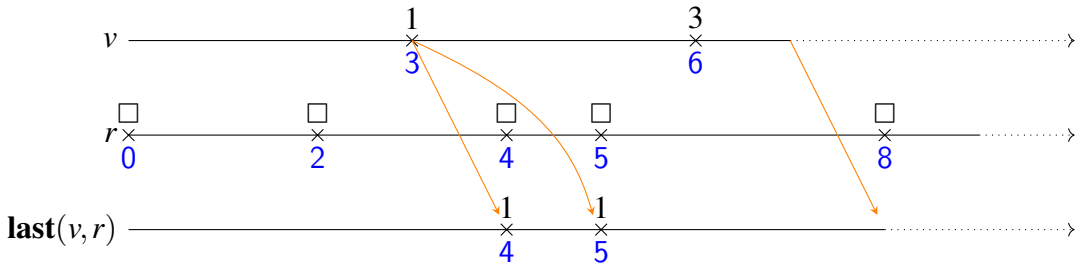
and

$$defined(z,t) \Leftrightarrow \forall t' < t : z(t') \neq ?$$

All in all, **last** outputs a ? when either a ? occurs on the second input stream or when an event occurs on the second stream after a ? occurred on the first input stream.

Example 3.31 (last *Operator over Event Streams*)

For two given input streams $v : \mathcal{S}_{\mathbb{N}}$ and $r : \mathcal{S}_{\mathbb{U}}$, the **last** operator over event streams looks as follows:



Compared to the streams in Example 3.14, in this example the stream v's progress ends at timestamp 7. Compared to the operators before, this does not lead directly to an end of the progress on the output stream, because we know that it is still $\bot$, due to r having no event. Therefore, the progress of the output stream finally ends at timestamp 8, when r has an event, since we do not know which value we would have to output there. On the other hand, if the progress of r ends first, also the progress of the output stream ends directly, as long as v had at least one event.

The final operator that has to be defined for the prefix semantics for TeSSLa is **delay**, which uses two additional predicates for representing the ?. Compared to the **delay** in the semantics over

completed streams, this **delay** has again an additional case for ?. Because $\bot$ occurs exactly in those cases, where $\square$ does not occur, minus the ending cases, we build the duals of *setable* and *noreset* and call them *unsetable* and *reset*, respectively, which we use for the $\bot$ case.

Definition 3.32 (delay *Operator on Event Streams, [CHL+18]*)

The **delay** $: \mathcal{S}_{\mathbb{T}\setminus\{0\}} \times \mathcal{S}_{\mathbb{D}} \to \mathcal{S}_{\mathbb{U}}$ *operator over event streams is defined as* $\mathbf{delay}(d,r) := z$*, where* z *is defined at every timestamp* $t \in \mathbb{T}$ *as follows:*

$$z(t) = \begin{cases} \square & \exists_{t'<t} d(t') = t - t' \wedge setable(z,r,t') \wedge noreset(r,t',t) \\ \bot & defined(z,t) \wedge \forall_{t'<t} d(t') \neq t - t' \wedge d(t') \neq ? \vee unsetable(z,r,t') \vee reset(r,t',t) \\ ? & otherwise \end{cases}$$

where

$$\begin{aligned} setable(z,r,t') &\Leftrightarrow z(t') = \square \vee t' \in ticks(r) \\ reset(r,t,t') &\Leftrightarrow \exists_{t''|t<t''<t'} t'' \in ticks(r) \\ unsetable(z,r,t') &\Leftrightarrow z(t') = \bot \wedge r(t') = \bot \\ noreset(r,t,t') &\Leftrightarrow \forall_{t''|t<t''<t'} r(t'') = \bot \end{aligned}$$

More precisely for the $\bot$ case, it occurs at the current timestamp if for each previous timestamp either

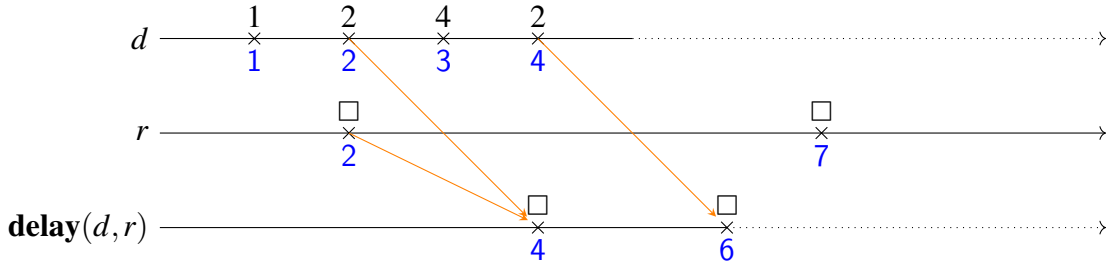
- the delay stream d does not have an event with a value pointing to the current timestamps and is not ?,
- *unsetable* holds, therefore, the reset stream r and the output stream z where $\bot$, or
- *reset* holds, therefore, in the future, there is an event on r before the current timestamp, cancelling every previous delay.

Additionally, *defined* has to hold such that no ? has been outputted on a previous timestamp already.

Example 3.33 (delay *Operator over Event Streams*)

For two given input streams $d \in \mathcal{S}_{\mathbb{N}}$ and $r \in \mathcal{S}_{\mathbb{U}}$, the **delay** operator over event streams looks as

follows:



The change to the semantics over completed streams occurs at timestamp 6. Because there is an output event and the progress of stream d has already ended, a new timeout could be set, but we do not know if there will be an event on d. Therefore, there could be an output event at any timestamp after timestamp 6, but we do not know yet. For this reason, the progress of the delay ends after timestamp 6.

All the operations we derived from the six core TeSSLa operators for the semantics over completed streams can be used in the same way for the prefix semantics, because the **lift** takes care of the stream endings and thus the functions used in the **lift** can stay the same. An example for how standard operations work in the prefix semantics is already given in Example 3.18 in the concrete case of an addition. The other derived operations like const, merge, filter and slift work in the same way as before, only potentially changing when one or more streams end. In the following four examples, we will recap those derived operations.

The first operation we take a look at is const for defining constant value streams, this time using the prefix semantics.

Example 3.34 (*Constant Values in Prefix Semantics*)

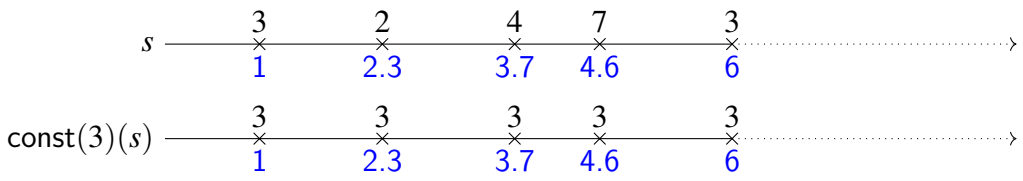
As in Example 3.17, a mapping to a constant value can be defined as:

$$\mathsf{const}(c)(s) := \mathbf{lift}(f_c)(s)$$

with

$$f_c(d) := \begin{cases} c & \text{if } c \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

The f_c does not need to be changed, because **lift** takes care of the ? values. The following stream picture shows the usage of const on a event stream s and a constant 3:



Because the **lift** only has one input stream, the progress ends on the output stream as soon as it ends on the input stream.

The second operation we consider over the prefix semantics is merge for merging two streams.

Example 3.35 (*Merging Streams in Prefix Semantics*)

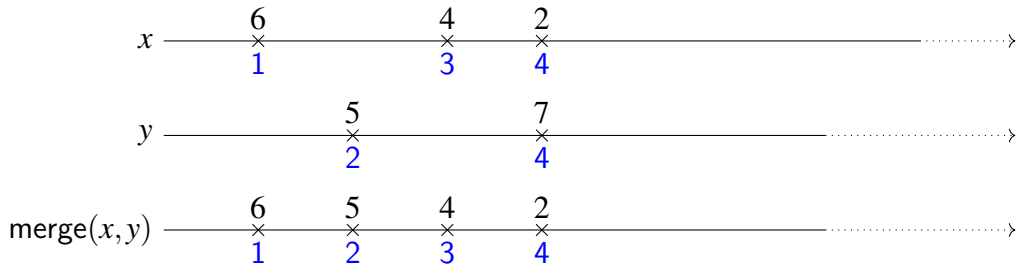
As in Example 3.19, a merging operation for event streams can also be defined in the same way:

$$\mathsf{merge}(x,y) := \textbf{lift}(f)(x,y)$$

with

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

With fixed input streams, the merge in the prefix semantics looks as follows:



The third operation we recall over the prefix semantics is filter for filtering a streams events depending on the truth value of a condition.

Example 3.36 (*Filtering in Prefix Semantics*)

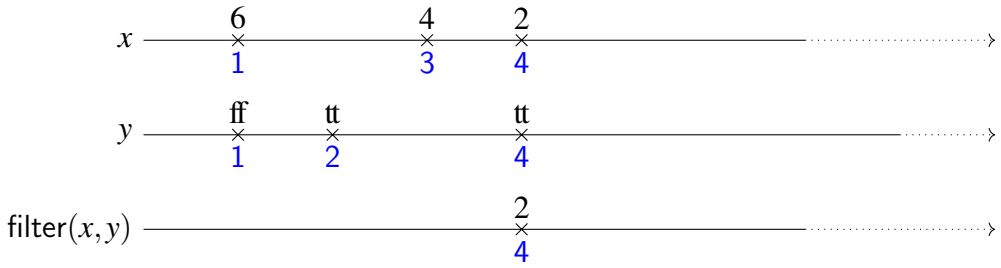
As in Example 3.20, a filtering operation for event streams can also be defined in the same way:

$$\mathsf{filter}(x,y) := \textbf{lift}(f)(x, \mathsf{merge}(y, \textbf{last}(y,x)))$$

with

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

With fixed input streams, the filter in the prefix semantics looks as follows:



The last operation we already considered is slift for implementing the signal semantics.

Example 3.37 (*Signal Semantics in Prefix Semantics*)

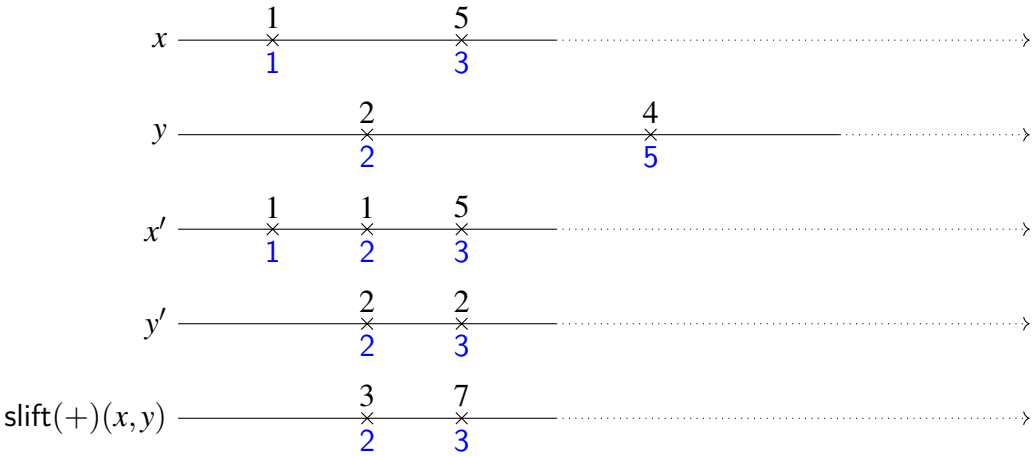
As in Example 3.21, we can define the **slift** for the signal semantics as before:

$$\mathsf{slift}(f)(x,y) := \mathbf{lift}(g_f)(x',y')$$

with $x' := \mathsf{merge}(x, \mathbf{last}(x,y))$ and $y' := \mathsf{merge}(y, \mathbf{last}(y,x))$ as well as

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

For two given streams x and y and the standard addition $+$, the slift works as follows:



We will now go on with another two examples, the first one being the running example from Example 3.22, updated to the prefix semantics.

Example 3.38 (*TeSSLa Specification with Prefix Semantics*)

We can now specify the stream transformations shown in Figure 3.2 with input streams value and resets and output streams cond and sum in TeSSLa with prefix semantics. Let $\mathsf{resets} \in \mathcal{S}_{\mathbb{U}}$ and $\mathsf{values} \in \mathcal{S}_{\mathbb{Z}}$ be two input event streams. We then derive $\mathsf{cond} \in \mathcal{S}_{\mathbb{B}}$ and $\mathsf{lst}, \mathsf{sum} \in \mathcal{S}_{\mathbb{Z}}$ as follows:

$$\begin{aligned} \mathsf{cond} &= \mathsf{slift}(\leq)(\mathbf{time}(\mathsf{resets}), \mathbf{time}(\mathsf{values})) \\ \mathsf{lst} &= \mathsf{merge}(\mathbf{last}(\mathsf{sum}, \mathsf{values}), \mathsf{zero}) \\ \mathsf{sum} &= \mathsf{slift}(f)(\mathsf{cond}, \mathsf{lst}, \mathsf{values}) \end{aligned}$$

where

$$f : \mathbb{B} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \text{ with}$$
$$f(c, l, v) = \begin{cases} 0 & \text{if } c = \text{true} \\ l + v & \text{otherwise} \end{cases}$$

Compared to Example 3.22, the specification only differs in the fact that it can now handle stream endings, and thus ? values. At the timestamp values ends, cond ends as well because, informally speaking, the slift is on one stream ?, which means it does not know the value there, and therefore also outputs ?. The same holds for sum. The stream lst also gets ?, because the **last** is ? as soon as a ? occurs on the second input stream and there was at least one event on the first input stream or ? before. The merge than merges the ? in.

The second example shows how the fixed-point calculation for a TeSSLa specification under the prefix semantics works in detail.

Example 3.39 (*Fixed-Point Calculation*)

Consider the equation

$$y = \mathsf{merge}\big(\mathbf{lift}\big(+1\big)\big(\mathbf{last}(y, x)\big), 0\big)$$

where $0 = \mathsf{zero} = \mathsf{const}(0, \mathsf{unit}) = \mathsf{const}(0, \Box\infty) = 0\ 0\ \infty$.

In this example, we use the representation of streams as sequences, therefore, every second

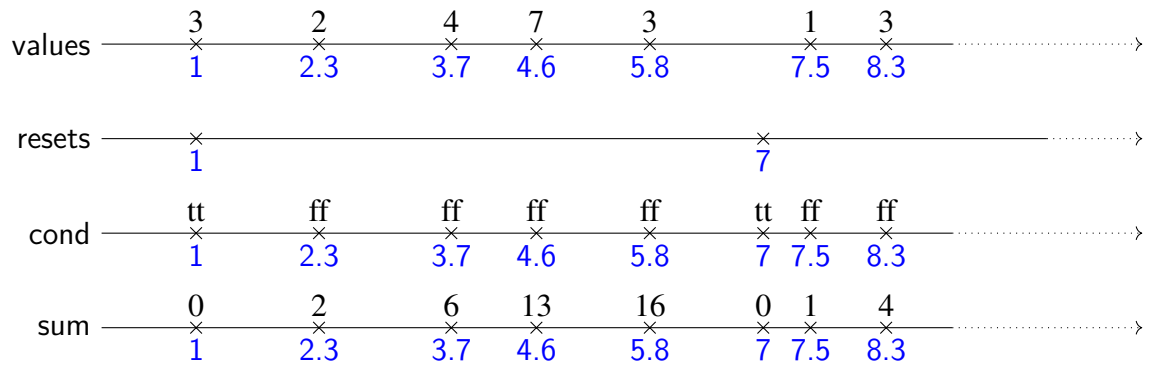


Figure 3.2: Figure 3.1 adjusted to the prefix semantics. The dotted lines indicate that the streams values, cond and sum end at the same time, hence are ?, while the progress of resets ends a bit later. As can be seen, the prefix semantics produces the same output as the semantics over completed streams as long as no stream ended.

symbol is a timestamp and the last timestamp of the stream indicates the end of its progress and therefore the point from which on the stream is ?.

Let us examine the equation for a fixed input stream $x = 2 \ \square \ 4 \ \square \ \infty$. To compute the least fixed-point of this function we start with the empty stream (the stream without any progress, therefore, ? only) $y_0 = 0$. We than have

$$\begin{aligned} \mathbf{last}(y_0, x) &= 2 \\ \mathbf{lift}(+1)\big(\mathbf{last}(y_0, x)\big) &= 2 \\ \mathsf{merge}\big(\mathbf{lift}(+1)\big(\mathbf{last}(y_0, x)\big), 0\big) &= 0\ 0\ 2 \end{aligned}$$

For the next iteration we start with $y_1 = 0\ 0\ 2$:

$$\begin{aligned} \mathbf{last}(y_1, x) &= 2\ 0 \\ \mathbf{lift}(+1)\big(\mathbf{last}(y_1, x)\big) &= 2\ 1 \\ \mathsf{merge}\big(\mathbf{lift}(+1)\big(\mathbf{last}(y_1, x)\big), 0\big) &= 0\ 0\ 2\ 1 \end{aligned}$$

For the next iteration we start with $y_2 = 0\ 0\ 2\ 1$:

$$\begin{aligned} \mathbf{last}(y_2, x) &= 2\ 0\ 4 \\ \mathbf{lift}(+1)\big(\mathbf{last}(y_2, x)\big) &= 2\ 1\ 4 \\ \mathsf{merge}\big(\mathbf{lift}(+1)\big(\mathbf{last}(y_2, x)\big), 0\big) &= 0\ 0\ 2\ 1\ 4 \end{aligned}$$

For the next iteration we start with $y_3 = 0\ 0\ 2\ 1\ 4$:

$$\begin{aligned}\mathbf{last}(y_3, x) &= 2\ 0\ 4\ 1\ \infty \\ \mathbf{lift}(+1)(\mathbf{last}(y_3, x)) &= 2\ 1\ 4\ 2\ \infty \\ \mathsf{merge}(\mathbf{lift}(+1)(\mathbf{last}(y_3, x)), 0) &= 0\ 0\ 2\ 1\ 4\ 2\ \infty\end{aligned}$$

For the next iteration we start with $y_4 = 0\ 0\ 2\ 1\ 4\ 2\ \infty$:

$$\begin{aligned}\mathbf{last}(y_4, x) &= 2\ 0\ 4\ 1\ \infty \\ \mathbf{lift}(+1)(\mathbf{last}(y_4, x)) &= 2\ 1\ 4\ 2\ \infty \\ \mathsf{merge}(\mathbf{lift}(+1)(\mathbf{last}(y_4, x)), 0) &= 0\ 0\ 2\ 1\ 4\ 2\ \infty\end{aligned}$$

So we have reached a fixed-point.

Note that $y_0 \sqsubseteq y_1 \sqsubseteq y_2 \sqsubseteq y_3 \sqsubseteq y_4$ regarding the prefix relation.

The next statement is important, since it states that the prefix semantics always does as much progress as possible, hence, outputs as much information as it can certainly derive from the input.

Proposition 3.40 (*Prefix Semantics Produces Maximum Prefixes*)

Let $x_1, \ldots, x_n \in \mathcal{S}_{\mathbb{D}}$ and let φ be a TeSSLa formula with $\varphi(x_1, \ldots, x_n) = y_1, \ldots, y_m$. Then $y_1, \ldots, y_m$ is a maximum prefix regarding φ and $x_1, \ldots, x_n$.

If you add additional knowledge to the end of the input streams when using the prefix semantics, the output gets additional knowledge as well, as the previous proposition states. If you proceed with this and the input converges into a completed stream, the output also does this and the completed streams the input and output streams converge to are also related in the semantics over completed streams. This is stated by the following proposition:

Proposition 3.41 (*Relation Between Semantics over Completed Streams and Prefix Semantics*)

Let $x_1, \ldots, x_n \in \mathcal{S}^{\infty}_{\mathbb{D}}$ with $\varphi(x_1, \ldots, x_n) =_{\infty} x'_1, \ldots, x'_m$ and $s_1, \ldots, s_n \in \mathcal{S}_{\mathbb{D}}$ with $\varphi(s_1, \ldots, s_n) = s'_1, \ldots, s'_m$. Then the following holds

$$(\forall 1 \leq i \leq n : s_i \sqsubseteq x_i) \Rightarrow \forall 1 \leq i \leq m : s'_i \sqsubseteq x'_i$$

Both propositions together state that the prefix semantics always outputs as much of the output from the semantics over completed streams as possible, taking the ending of the progress of the streams, marked by the ?, into account. In the end this also means that, if we consider a set of completed streams, then both semantics will provide the same output.

An important aspect we will use later in this thesis is compositionality. It means that a formula can be split arbitrarily into two syntactically correct formulas where the second formula takes the output of the first formula as input and the output of the second formula is the same as the output of the original formula. The other direction is supposed to hold as well, two formulas evaluated after another where one uses the output of the other as input can be combined into one big formula, while keeping the semantics. Therefore, in case of, for example, LOLA or TeSSLa, it means that the operators can be composed in such a way, as well as the equations (which is important, because for expressing certain stream transformations, multiple equations are needed).

We state in the following proposition that the operators and equations of TeSSLa and therefore TeSSLa specifications are closed under composition.

Proposition 3.42 (*Compositionality of TeSSLa Semantics*)

TeSSLa specifications are closed under composition.

In the following, we will only consider the prefix semantics and, if not stated otherwise, will only call it semantics.

3.3 Adding a Future Operator to TeSSLa

In the last section of this chapter, we add an operator for future references to TeSSLa. This extension, which we call TeSSLa^f, has an additional operator which complements the **last** operator, but instead of returning the previous value on the value parameter stream, it will return the following value on the value parameter stream. This operator will be called **next**. This is an extension to the version of TeSSLa known from [CHL$^+$18].

Before we get to the definition of TeSSLa^f, we extend our representation of streams. Until now, a stream had events with values over a data domain at certain timestamps, $\perp$ between those events and at some point ?, which represents that the progress of the stream ended at this point. Thus there was an invariant which stated that from the first point on where we had ?, there is ? forever on the stream. While we could keep this notion of streams, every use of the **next** operator would possibly

result in a very early start of ? on the output stream, if the input streams had any ?. Extending the notion of streams by removing this invariant lets us represent and keep more information by allowing point-wise ? values followed by other values or $\bot$. Therefore, even though the **next** operator could be added without lifting the invariant, it makes sense to combine the introduction of **next** with lifting the invariant.

Thus, at first, we will lift this invariant from our stream model and allow ? values at arbitrary places in the stream. We call this type of streams multi-progress streams. Multi-progress streams are an extension of event streams, allowing the use of ? without any invariants. In these streams, at each point where something happens on the stream, a triple (t,d,x) represents this, where t is the timestamp, d the value (may be $\bot$ or ?) and x states what follows after t, either $\bot$ or ? until the next triple occurs at a later timestamp. On each stream the first triple is forced at timestamp 0, which even if there is no event there, either states that the stream starts with $\bot$ or ?.

Definition 3.43 (*Multi-Progress Streams*)

A multi-progress stream *over a time domain* $\mathbb{T}$ *and a data domain* $\mathbb{D}$ *is a finite or infinite sequence* $s = a_0 a_1 \cdots \in \mathcal{M}_{\mathbb{D}} = \{0\} \times \mathbb{D} \cup \{\bot,?\} \times \{\bot,?\} \cdot X^{\omega} \cup X^*$ *where* $X = \mathbb{T} \times \mathbb{D} \cup \{\bot,?\} \times \{\bot,?\}$ *where* $a_{2i} < a_{2(i+1)}$ *for all* i *with* $0 < 2(i+1) < |s|$ *(*$|s|$ *is* ∞ *for an infinite number of events).*

Note that it is possible to add redundant triples in multi-progress streams, like a subsequence of two triples $(t,\bot,\bot)(t',\bot,\bot)$, where the first states that the current value is $\bot$ and the values after this triple as well, while the second triple is stating the same, which has no effect. But adding such redundant triples does not add anything to the streams, and the second triple in this case can just be ignored. But we do not forbid such sequences per definition because it would make the definition much less readable.

We can again represent a multi-progress stream s as a function $s : \mathbb{T} \to \mathbb{D} \cup \{\bot,?\}$ as follows

$$s(t) = \begin{cases} d & \text{if } s \text{ contains } (t,d,x) \\ x, \text{ where } s \text{ contains } (t',d,x) \text{ with } \nexists t' < t'' < t : s \text{ contains } (t,d',x') & \text{otherwise} \end{cases}$$

which states that at any timestamp t, there is either a triple (t,d,x) which means the stream has value d at timestamp t or, if there is no triple, the stream has still the value x of the previous triple, so is either $\bot$ or ? at timestamp t. We will use both representations of multi-progress streams, as a function and as a sequence of tuples, for the rest of the thesis, depending on what is appropriate.

The prefix relation for multi-progress streams is defined similar to the one for event streams. The difference is, that now at all positions a ? can occurs, independently of what happens afterwards. As ? is the value with the least knowledge (it is unknown what value will be there), a multi-progress stream is a prefix of another if it contains at least as many timestamps with ? and the values at all other timestamps are equivalent.

Definition 3.44 (*Prefixes of Multi-progress Streams*)

We say a multi-progress stream $u \in \mathcal{M}_{\mathbb{D}}$ is a prefix of another multi-progress stream $s \in \mathcal{M}_{\mathbb{D}}$, $u \sqsubseteq s$, iff the following holds:

$$\forall t \in \mathbb{T} : u(t) = s(t) \vee u(t) = ?$$

Therefore, according to this prefix relation, the lowest element of multi-progress streams would still be the stream with no progress, thus the stream $s = (0, ?, ?)$ and a stream $u = (0, \bot, \bot)(2, ?, \bot)$ would be a prefix of $u' = (0, \bot, \bot)(2, \bot, \bot)$ and $u'' = (0, \bot, \bot)(2, 5, \bot)$.

Now, we define the syntax of this TeSSLa extension, which is the syntax of TeSSLa with the new **next** operator being added.

Definition 3.45 (*TeSSLa^f Syntax*)

Let I be a set of input streams and f be a k-ary function. Then a TeSSLa^f specification φ is a system of equations each of the form $x := e$, where the syntax of each e is given through the following grammar, with s_{ref} being a constant reference to an input stream $s \in I$, which is interpreted as a stream and y the left hand side of an equation of φ:

$$e ::= \textbf{nil} \mid \textbf{unit} \mid s \mid y \mid \textbf{lift}(f)(e, \ldots, e) \mid \textbf{time}(e) \mid \textbf{last}(e, e) \mid \textbf{delay}(e, e) \mid \textbf{next}(e, e).$$

The only addition is the new **next** operator as mentioned before. In the following, we define the semantics of TeSSLa^f over multi-progress streams. Therefore, we also update the semantics of the TeSSLa operators, which updates the parts of the definitions that used the invariant of ? being final on streams and allows the operators to handle ? at arbitrary positions. This does not effect the semantics over completed streams, therefore streams without ?, at all, but does change different parts of the definitions for the operators in the prefix semantics.

Concretely for the definition of the general semantics, the only change is that the input and output streams are now multi-progress streams.

Definition 3.46 (*TeSSLa^f Semantics*)

Let φ be a TeSSLa^f specification. Then the semantics of the TeSSLa^f specification φ with equations $y_1 := e_1, \ldots, y_n := e_n$ and input streams I over multi-progress streams is the semantic function $[\![\varphi]\!]_M : \mathcal{M}_{\mathbb{D}_1} \times \cdots \times \mathcal{M}_{\mathbb{D}_n} \to \mathcal{M}_{\mathbb{D}'_1} \times \cdots \times \mathcal{M}_{\mathbb{D}'_m}$ and we write $[\![\varphi]\!]_M(I) = O$, where O is the set of output streams after the equations have been evaluated using the input streams in I. The semantics for the system of equations that is φ is given as the least fixed-point of the equations interpreted as a function of the stream variables and fixed input streams as follows:

$$[\![\varphi]\!]_M(I) = \mu([\![e_1]\!]_M(I), \ldots, [\![e_n]\!]_M(I))$$

The semantics for each single operator is given below.

The language a TeSSLa^f specification φ represents is given as $\mathcal{L}(\varphi) = \{(I,O) \mid [\![\varphi]\!]_M(I) = O\}$.

Compared to the prefix semantics, we have to relieve the assumption for the streams that, once a ? occurs on a stream at timestamp t, at all timestamps after t, the stream is also ?. This is what the multi-progress streams represent. The reason is that for the **next** operator, we also need to be able to get unknown values in the beginning or in between because the stream may end before we get the future values needed to evaluate the **next**, but values in between the positions where the **next** may output something can possibly be calculated still, even though the **next** may add some ? before. Without lifting the invariant, using the next would be quite useless, as the output streams would be ? at most of the timestamps.

Again, we abuse notation and interpret a TeSSLa^f formula φ directly as a function and write $\varphi(I) = O$ for $[\![\varphi]\!]_M(I) = O$, when it becomes clear from the context that the semantics over multi-progress streams are used.

To adjust the core functions, we only need to remove the parts which rely on the invariant used in the prefix semantics.

The only thing which changes for **nil** is the representation, because ∞ and $(0, \bot, \bot)$ both represent the stream which is always $\bot$, only in different types of stream models. As **nil** has no input stream, nothing has to be changed regarding the handling of ?.

Definition 3.47 (nil *Operator on Multi-Progress Streams*)

The **nil** $\in \mathcal{M}_{\emptyset}$ *operator on multi-progress streams is defined as follows:*

$$\textbf{nil} = (0, \bot, \bot)$$

unit $= (0, \Box, \bot) \in \mathcal{M}_{\mathbb{U}}$ is the stream with a single unit event at timestamp zero and no other events. As for **nil**, this operators also stays the same in all semantics, hence $\textbf{unit}(0) = \Box$ and $\forall 0 < t \in \mathbb{T} : \textbf{unit}(t) = \bot$.

Definition 3.48 (unit *Operator on Multi-Progress Streams*)

The **unit** $\in \mathcal{M}_{\mathbb{U}}$ *operator on multi-progress streams is defined as follows:*

$$\textbf{unit} = (0, \Box, \bot)$$

As for **nil**, the changes to the **unit** operator are only because of the new stream model, **unit** does still represent the same stream as before.

Because **time** was just copying values if no event occurred, it can still do so and the definition does not change at all. The possibly additional ? values in between on a stream do not interfere with the way **time** worked before.

Definition 3.49 (time *Operator on Multi-Progress Streams*)

The **time** $: \mathcal{M}_{\mathbb{D}} \to \mathcal{M}_{\mathbb{T}}$ *operator over multi-progress streams is defined as* $\textbf{time}(s) := z$*, where* z *is defined at every timestamp* $t \in \mathbb{T}$ *as follows:*

$$z(t) = \begin{cases} t & \text{if } t \in \mathit{ticks}(s) \\ s(t) & \text{otherwise} \end{cases}$$

lift changes quite a bit compared to the prefix semantics. In the prefix semantics, the premise has to be kept that once a stream is ?, it stays ? forever. This is not necessary anymore. Thus it can be decided by the function locally how it handles the ? because there is no premise that has to be fulfilled over time, which means the function f needs the signature $f : \mathbb{D}_1 \cup \{\bot, ?\} \times \cdots \times \mathbb{D}_n \cup$

$\{\bot,?\} \to \mathbb{D} \cup \{\bot,?\}$ now. This removes one of the cases the **lift** had in the prefix semantics and simplifies the definition of the output stream in the end.

As before, we need to restrict the functions f to those that do not create events, hence

$$f(\bot,\ldots,\bot) = \bot$$

but additionally, due to the new stream model, we need another criteria such that the function f is consistent and does not output more knowledge than it can have due to the inputs, which is done by the new constraint

$$\exists i : d_i = ? \wedge f(d_1,\ldots,d_i,\ldots,d_n) = x \wedge x \neq ? \Rightarrow \forall d_i' \in \mathbb{D}_i \cup \{\bot\} : f(d_1,\ldots,d_i',\ldots,d_n) = x$$

thus if one of the inputs of a function f is a ? and f outputs something else than a ?, is has to output the same value independently of the value of the input parameter which was ?, as long as the other parameters stay the same.

Definition 3.50 (lift *Operator on Multi-Progress Streams*)

Let $f : \mathbb{D}_1 \cup \{\bot,?\} \times \cdots \times \mathbb{D}_n \cup \{\bot,?\} \to \mathbb{D} \cup \{\bot,?\}$ *be a non-creating function on n data values, which fulfils*

$$\exists i : d_i = ? \wedge f(d_1,\ldots,d_i,\ldots,d_n) = x \wedge x \neq ? \Rightarrow \forall d_i' \in \mathbb{D}_i \cup \{\bot\} : f(d_1,\ldots,d_i',\ldots,d_n) = x$$

The **lift** $: (\mathbb{D}_1 \cup \{\bot,?\} \times \cdots \times \mathbb{D}_n \cup \{\bot,?\} \to \mathbb{D} \cup \{\bot,?\}) \to (\mathcal{M}_{\mathbb{D}_1} \times \cdots \times \mathcal{M}_{\mathbb{D}_n} \to \mathcal{M}_{\mathbb{D}})$ *operator over multi-progress streams is defined as* $\mathbf{lift}(f)(s_1,\ldots,s_n) := z$*, where z is defined at every timestamp* $t \in \mathbb{T}$ *as follows:*

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

As for the **lift**, again the parts can be taken out that were necessary to secure the premise of the prefix semantics in the definition of the **last** operator. Instead we have to check that no ? occurred after the last value on v. This is already done implicitly by the *isLast* predicate. Thus the only change to be made is in the $\bot$ case by simply removing the *defined* predicate, which exactly took care of the invariant.

Definition 3.51 (last *Operator on Multi-Progress Streams*)

The **last** $: \mathcal{M}_{\mathbb{D}} \times \mathcal{M}_{\mathbb{D}'} \to \mathcal{S}_{\mathbb{D}}$ *operator over multi-progress streams is defined as* $\mathbf{last}(v,r) := z$*,*

where z is defined at every timestamp $t \in \mathbb{T}$ as follows:

$$z(t) = \begin{cases} d & t \in \mathit{ticks}(r) \text{ and } \exists_{t'<t} \mathit{isLast}(t,t',v,d) \\ \bot & r(t) = \bot \text{ or } \forall_{t'<t} v(t') = \bot \\ ? & \mathit{otherwise} \end{cases}$$

where

$$\mathit{isLast}(t,t',v,d) \Leftrightarrow v(t') = d \wedge \forall_{t''|t'<t''<t} v(t'') = \bot$$

The definition of the **delay** operator has to be changed in the same way as for the **last**. When a $\square$ is outputted, the definition already takes care that no ? occurs on r in between by the *noreset* predicate. Only in the $\bot$ case, we again have to remove the *defined* predicate and leave the rest of the condition as it was.

Definition 3.52 (delay *Operator on Multi-Progress Streams, [CHL+18]*)

The **delay** $: \mathcal{M}_{\mathbb{T}\setminus\{0\}} \times \mathcal{M}_{\mathbb{D}} \to \mathcal{M}_{\mathbb{U}}$ *operator over Multi-Progress streams is defined as* $\mathbf{delay}(d,r) :=$ *z, where z is defined at every timestamp $t \in \mathbb{T}$ as follows:*

$$z(t) = \begin{cases} \square & \exists_{t'<t} d(t') = t - t' \wedge \mathit{setable}(z,r,t') \wedge \mathit{noreset}(r,t',t) \\ \bot & \forall_{t'<t} d(t') \neq t - t' \wedge d(t') \neq ? \vee \mathit{unsetable}(z,r,t') \vee \mathit{reset}(r,t',t) \\ ? & \mathit{otherwise} \end{cases}$$

where

$$\begin{aligned} \mathit{setable}(z,r,t') &\Leftrightarrow z(t') = \square \vee t' \in \mathit{ticks}(r) \\ \mathit{reset}(r,t,t') &\Leftrightarrow \exists_{t''|t<t''<t'} t'' \in \mathit{ticks}(r) \\ \mathit{unsetable}(z,r,t') &\Leftrightarrow z(t') = \bot \wedge r(t') = \bot \\ \mathit{noreset}(r,t,t') &\Leftrightarrow \forall_{t''|t<t''<t'} r(t'') = \bot \end{aligned}$$

By now, we updated all original TeSSLa operators semantics to the multi-progress streams. Lastly in this section, after changing the TeSSLa operators accordingly, we now define the new operator which allows us to specify future references in the same way as the **last** does for past references. Every time an event occurs on its second input stream, it outputs the value of the following future event on the first input stream.

Definition 3.53 (next *Operator*)

next : $\mathcal{M}_{\mathbb{D}} \times \mathcal{M}_{\mathbb{D}'} \to \mathcal{M}_{\mathbb{D}}$ *with* $\mathbf{next}(v,r) := z$ *takes a value stream* v *and a trigger stream* r*. Formally,* z *is defined for every* $t \in \mathbb{T}$ *as follows:*

$$z(t) = \begin{cases} d & t \in ticks(r) \text{ and } \exists t' > t : isNext(t,t',v,d) \\ \bot & r(t) = \bot \\ ? & otherwise \end{cases}$$

where $isNext(t,t',v,d) \Leftrightarrow v(t') = d \wedge \forall t' > t'' > t : v(t'') = \bot$ *holds if* $t'd$ *is the following event on* v *after* t*.*

As one can see, compared to the **last** operator, **next** is defined very similar, mostly the greater than and less than signs at the timestamp quantifications have turned into the other one.

Before moving to an example for the fixed-point calculations for TeSSLa^f, let us first give two examples of lifted functions, which now has to take care of the ? as well. The first one is the merge again, while the second one is an if-then-else.

Example 3.54 (*merge in TeSSLa^f*)

In TeSSLa^f, each lifted function has to consider the ? case as well. For the merge this is defined formally as

$$\mathsf{merge}(x,y) := \mathbf{lift}(f)(x,y)$$

with

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

In the case of merge, we do not have to change anything, because the only constraint regarding the ? is, that if one input stream is ? and the other is not ?, then the output should not change if the input stream with the ? has a different value in the same situation. And this always holds for the merge, if a is ?, the output is ? and if b is ? then it only changes something if a is $\bot$ and then the output is also ?.

Example 3.55 (*ifThenElse in TeSSLa^f*)

In TeSSLa^f we defined an if-then-else as

$$\mathsf{ifThenElse}(b,x,y) := \mathbf{lift}(f)(b,x,y)$$

with

$$f(b,x,y) = \begin{cases} x & \text{if } b = \mathrm{tt} \\ y & \text{if } b = \mathrm{ff} \\ b & \text{otherwise} \end{cases}$$

If there is an event on b, either the value of x or y is outputted depending on the truth value on b. If there is no event on b, its value is copied to the output stream, therefore either $\bot$ or ?.

The following example shows how the **next** operators works and how least fixed-points are calculated using it.

Example 3.56 (*Calculations with* next)

Consider the equation

$$y = \mathsf{merge}\big(\mathbf{lift}(+)\big(\mathbf{next}(y,x),x\big),d\big)$$

where $d = (0,\bot,\bot)(42,0,\bot)$.

The equation can be seen as function of y with fixed input stream $x = (0,\bot,\bot)(2,2,\bot)(4,4,\bot)$. To compute the least fixed-point of this function we start with the empty stream $y_0 = (0,?,?)$. We than have

$$\begin{aligned} \mathbf{next}(y,x) &= (0,\bot,\bot)(2,?,\bot)(4,?,\bot) \\ \mathbf{lift}(+)\big(\mathbf{next}(y,x),x\big) &= (0,\bot,\bot)(2,?,\bot)(4,?,\bot) \\ \mathbf{merge}\big(\mathbf{lift}(+)\big(\mathbf{next}(y,x),x\big),d\big) &= (0,\bot,\bot)(2,?,\bot)(4,?,\bot)(42,0,\bot) \end{aligned}$$

For the next iteration we start with $y_1 = (0,\bot,\bot)(2,?,\bot)(4,?,\bot)(42,0,\bot)$:

$$\begin{aligned} \mathbf{next}(y,x) &= (0,\bot,\bot)(2,?,\bot)(4,0,\bot) \\ \mathbf{lift}(+)\big(\mathbf{next}(y,x),x\big) &= (0,\bot,\bot)(2,?,\bot)(4,4,\bot) \\ \mathbf{merge}\big(\mathbf{lift}(+)\big(\mathbf{next}(y,x),x\big),d\big) &= (0,\bot,\bot)(2,?,\bot)(4,4,\bot)(42,0,\bot) \end{aligned}$$

For the next iteration we start with $y_2 = (0,\bot,\bot)(2,?,\bot)(4,4,\bot)(42,0,\bot)$:

$$\begin{aligned} \mathbf{next}(y,x) &= (0,\bot,\bot)(2,4,\bot)(4,0,\bot) \\ \mathbf{lift}(+)\big(\mathbf{next}(y,x),x\big) &= (0,\bot,\bot)(2,6,\bot)(4,4,\bot) \\ \mathbf{merge}\big(\mathbf{lift}(+)\big(\mathbf{next}(y,x),x\big),d\big) &= (0,\bot,\bot)(2,6,\bot)(4,4,\bot)(42,0,\bot) \end{aligned}$$

For the next iteration we start with $y_3 = (0,\bot,\bot)(2,6,\bot)(4,4,\bot)(42,0,\bot)$:

$$\begin{aligned}
\mathbf{next}(y,x) &= (0,\bot,\bot)(2,4,\bot)(4,0,\bot)\\
\mathbf{lift}\big(+\big)\big(\mathbf{next}(y,x),x\big) &= (0,\bot,\bot)(2,6,\bot)(4,4,\bot)\\
\mathbf{merge}\big(\mathbf{lift}\big(+\big)\big(\mathbf{next}(y,x),x\big),d\big) &= (0,\bot,\bot)(2,6,\bot)(4,4,\bot)(42,0,\bot)
\end{aligned}$$

So we have reached a fixed-point.

Note that $y_0 \sqsubseteq y_1 \sqsubseteq y_2 \sqsubseteq y_3$ regarding the prefix relation.

4 Language Theoretic Results

Contents

In this section we present results regarding the complete TeSSLa language, taking a look at the structure of formulas and what kind of formulas we consider in this thesis, as well as making statements about the expressiveness the different operators deliver, considering fragments with and without the **delay** operator. These results have been published in [CHL$^+$18]. Additionally, we show which additional formulas we can express when adding the **next** operator, thus considering TeSSLa^f formulas, also with and without **delay**.

4.1 General Properties and Computability

We start off stating that the least fixed point of a TeSSLa specification always exists. This follows from the fact that event and multi-progress streams form a dcpo $(L, \sqsubseteq)$ with $L = \mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_n}$ and a least element 0 or $L = \mathcal{M}_{\mathbb{D}_1} \times \cdots \times \mathcal{M}_{\mathbb{D}_n}$ with a least element $(0, ?, ?)$, for event or multi-progress streams, respectively. By applying the Kleene fixed point theorem (Theorem 2.4), we get:

Proposition 4.1 (*Least Fixed Point in TeSSLa and TeSSLa^f*)

The least fixed point for a TeSSLa and TeSSLa^f specification always exists.

We go on describing two properties of TeSSLa^f, and therefore also TeSSLa, in general. We can observe that all seven basic operators **nil**, **unit**, **time**, **lift**, **last**, **delay** and **next** are monotonic and continuous. From the fact that both properties are closed under composition and that the smallest fixed-point is determined by the Kleene chain, we can therefore conclude:

Lemma 4.2 (*Basic Properties of TeSSLa^f semantics, [CHL+18]*)

The semantics of TeSSLa^f is monotonic and continuous in the input streams.

This means that a TeSSLa and TeSSLa^f specification represents nothing else than a stream transformation. In other words, the semantics will provide an extended result for an extended input and is therefore suited, for example, for online analysis and monitoring.

Another important observation for TeSSLa is that the pre-fixed-points on the Kleene chain have the property that the progress only increases a finite number of times until a further event has to be appended. This means that after a finite computation time, a new event has to be outputted, the stream ends (? for the rest of the stream) or the progress is infinite ($\bot$ for the rest of the stream). This is due to the basic functions that do handle progress in this way.

We therefore obtain:

Theorem 4.3 (*Computability and Computation Time of TeSSLa Semantics, [CHL+18]*)

For a specification φ the following two statements hold:

1. *Every finite prefix of $\varphi(s_1, \ldots, s_k)$ can be computed assuming all lifted functions are computable.*
2. *Assuming the basic operators* **nil**, **unit**, **time**, **last** *and* **delay** *and all lifted functions are computable in $O(1)$ steps, a finite prefix can be computed in $O(k|\varphi|)$ steps where k is the number of events over all involved streams.*

Proof. The TeSSLa operators **nil**, **unit**, **time**, **last** and **delay** are computable on their own. The **lift** operator just applies a function on values to every value of some streams, which makes **lift** computable as long as the function it lifts is computable. Because TeSSLa specifications are just

a composition of TeSSLa operators and computability is compositional, TeSSLa specifications are also computable as long as the lifted functions are. This proves statement 1.

For the second statement, let us first view the **lift** operator in more detail. As this operator does nothing more than applying the lifted function to every event on the input streams and we assume that all lifted functions only need $O(1)$ steps to be computed, so does **lift**. For all other TeSSLa operators **nil**, **unit**, **time**, **last** and **delay**, we already assume that they process an event in $O(1)$ steps (or do not process events at all in case of **nil** and **unit**). So if k is the overall number of all events on all streams, it is the number of events which have to be processed by the TeSSLa operators in a specification φ, which all need $O(1)$ steps per event. Hence a finite prefix can be computed in $O(k|\varphi|)$. □

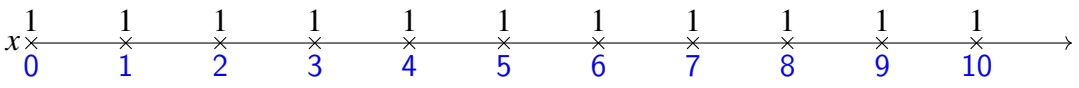
Note that in case the specification contains no **delay**, output streams cannot contain any such timestamps that did not occur already in the inputs. Further note, that fixed-points might contain infinitely many positions with data values (in case of **delay**) and we can thus only compute prefixes. A respective monitor would exhibit infinite outputs even for finite inputs. An example for this can be seen in Example 4.4.

Example 4.4 (*Creating events with* delay)

Consider the following TeSSLa specification:

$$x := \mathsf{const}(1, \mathsf{merge}(\mathbf{delay}(x, \mathbf{unit}), \mathbf{unit}))$$

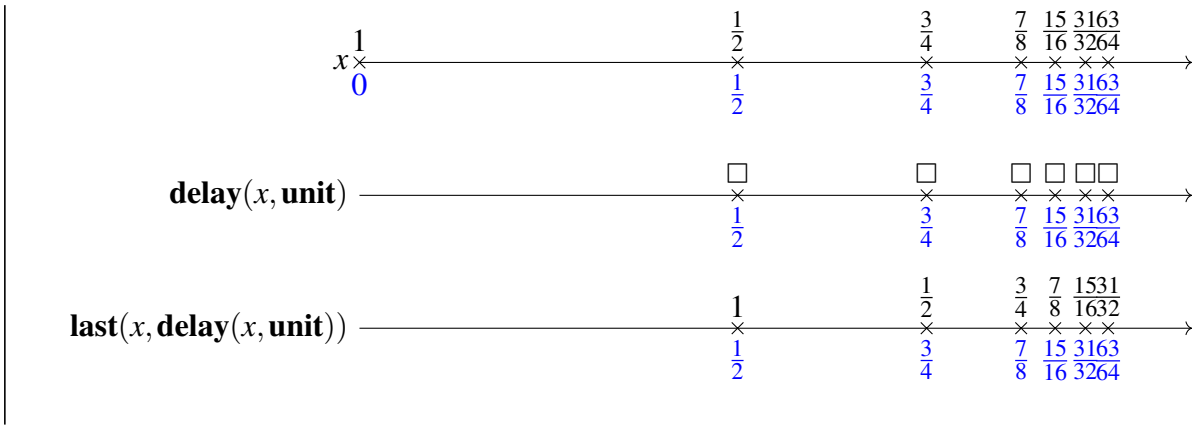
This specification does not depend on any input stream and still x is a stream which contains an event every 1 time unit beginning with timestamp 1. Thus x contains an infinite number of events.



Next, consider the following specification:

$$x := \mathsf{merge}\left(\frac{\mathbf{last}(x, \mathbf{delay}(x, \mathbf{unit}))}{2}, 1\right)$$

This specification always divides the last value on x by 2 when the **delay** emits an event, hence it always halves the next delay value. This results in a stream whose events timestamps converge to the timestamp 1 but never reach it.



Due to Lemma 4.2 we can reuse a previously computed fixed-point for a prefix if new input events occur and hence also compute the outputs incrementally by extending this prefix.

4.2 Well-formedness

Until now, we allowed any possible equation which can be build through using the six basic operators, constants and other equations names in a TeSSLa specification. But some and some combinations of these have multiple or even an infinite number of fixed-points. The simplest of those is the equation

$$x := x$$

which can be solved by assigning any possible stream to x. Also in such cases, the least fixed-point is often the stream without any progress or some other stream with too little progress and one would be interested in greater, for example the maximal, fixed-point, because otherwise, information may be ignored and, for the case of runtime verification, errors could not be detected as soon as possible.

In general, the user is interested in getting as much information as possible out of a monitoring system and therefore, about the software under scrutiny. Since maximal fixed-points would be more difficult to compute, especially in the setting of online monitoring, having only a single fixed point and being able to compute the least fixed point, may also be important for monitoring systems with limited computational power, like an FPGA. For these reasons, we define a fragment for which a unique fixed-point exists. At first, let us define what a dependency graph of a TeSSLa specification is.

Definition 4.5 (*Dependency Graph of TeSSLa Specifications, [CHL+18]*)

The dependency graph *of a flat TeSSLa specification φ of equations $y_i := e_i$ is the directed edge-labelled graph $G = (V, E)$ of nodes $V = \{y_1, \ldots, y_n\}$. For every $y_i := e_i$ the graph contains the edge $(y_i, y_j, l) \in E$ every time y_j is used in e_i and*

$$l = \begin{cases} \text{delayed} & \text{if } \exists k : e_i = \mathbf{last}(y_j, y_k) \vee e_i = \mathbf{delay}(y_j, y_k) \\ \varepsilon & \text{otherwise} \end{cases}$$

such that edges are labelled corresponding to the first argument of **last** *or* **delay** *with* delayed.

The following example shows the dependency graph for a given TeSSLa specification. In case of labelling an edge with the empty word ε, we will write no label on the corresponding edge.

Example 4.6 (*Dependency Graph of a TeSSLa Specification*)

Consider the specification

$$\begin{aligned} \mathsf{cond} &= \mathsf{slift}(\leq)(\mathbf{time}(\mathsf{resets}), \mathbf{time}(\mathsf{values})) \\ \mathsf{lst} &= \mathsf{merge}(\mathbf{last}(\mathsf{sum}, \mathsf{values}), \mathsf{zero}) \\ \mathsf{sum} &= \mathsf{slift}(f)(\mathsf{cond}, \mathsf{lst}, \mathsf{values}) \end{aligned}$$

from Examples 3.22 and 3.38 again. To build the dependency graph for this specification, we have to flatten it first. The flattened specification is the following, where we assume that merge, merge and zero are core operators, which means we will not split those into their definition using only core operators as we would have to do normally, to keep the size reasonable.

$$\begin{aligned} \mathsf{a} &= \mathbf{time}(\mathsf{resets}) \\ \mathsf{b} &= \mathbf{time}(\mathsf{values}) \\ \mathsf{cond} &= \mathsf{slift}(\leq)(\mathsf{a}, \mathsf{b}) \\ \mathsf{c} &= \mathbf{last}(\mathsf{sum}, \mathsf{values}) \\ \mathsf{lst} &= \mathsf{merge}(\mathsf{c}, \mathsf{zero}) \\ \mathsf{sum} &= \mathsf{slift}(f)(\mathsf{cond}, \mathsf{lst}, \mathsf{values}) \end{aligned}$$

The dependency graph for this specification can then be seen in Figure 4.1.

Using dependency graphs for TeSSLa specifications, we define now the property of well-formed

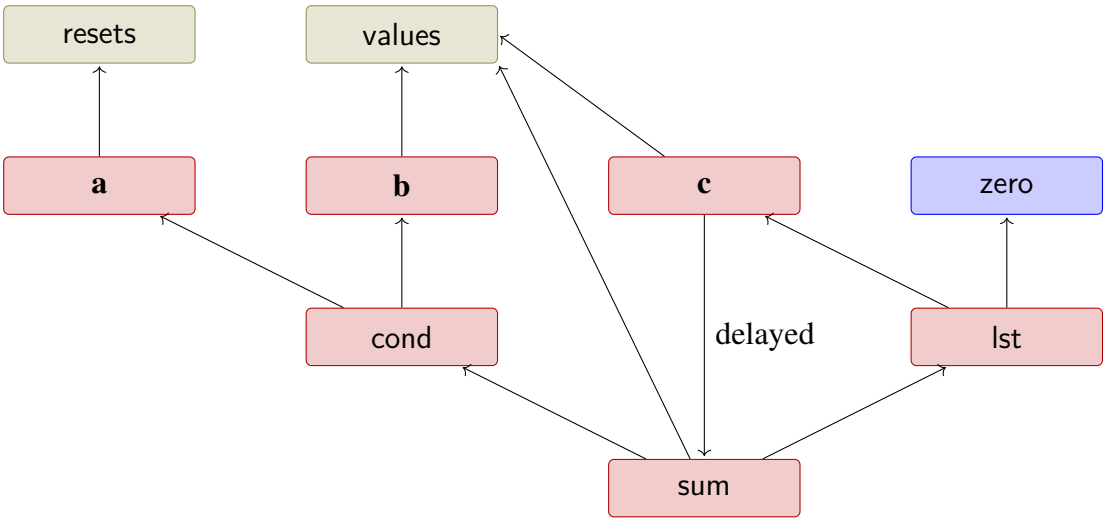


Figure 4.1: Shows the dependency graph for the TeSSLa specification given in Example 4.6. The yellow boxes denote input streams, the blue ones constants and the red ones TeSSLa operators. The red boxes with bold text in it are expressions with only core operators while the other ones represent functions which are build from core operators but are not considered into more detail in this graph.

TeSSLa specifications, which restricts TeSSLa specifications to those with unique fixed points.

Definition 4.7 (*Well-formedness of TeSSLa specifications, [CHL+18]*)

We call a TeSSLa specification φ well-formed *if every cycle* $(e_1, e_2, l_1), \ldots, (e_n, e_1, l_n)$ *of the dependency graph* $G = (V, E)$ *in the flattened specification contains at least one* delayed*-labelled edge such that* $\exists 1 \leq i \leq n : l_i =$ delayed*.*

Well-formedness forbids two types of specifications. First, every specification needs to have a **last** or **delay** in every cycle in the equations such that the cycle goes through the first parameter of those operators, so only older values can be referenced in a cycle. While the **last** just takes the previous value on the stream which is the first parameter, the **delay** works for such cycles because its first parameter sets a timeout in the future, so it has no effect on the current fixed-point calculated for the prefix. Second, it also forbids cycles which involve the second parameter of a **last** or **delay** in the specification, besides if the second parameter is again one of these two operators with the cycle going through its first parameter.

The following example shows four specifications and discussed the well-formedness of these four.

Example 4.8 (*Well-formed Specifications*)

Let a be an input stream. Consider the four TeSSLa specifications given next:

1.
$$x := y$$
$$y := x$$

2.
$$x := y$$
$$y := \mathbf{last}(x, x)$$

3.
$$x := y$$
$$y := \mathbf{last}(a, x)$$

4.
$$x := y$$
$$y := \mathbf{last}(x, a)$$

Specifications 1. to 3. are not well-formed, while 4. is. For 1. and 2., the reason for not being well-formed is that the value of x directly depends on itself for any given timestamp. The same holds for 3., the **last** only outputs an event when an event occurs on the second input stream, in this case, x. Therefore, x only has an event in this specification when it has an event, which means, events can be placed arbitrarily on x, they just need the previous value of a to fulfil the specification. Thus, more than one fixed point exists. In 4., a and x are swapped as parameters of the **last**. This means, that the timestamps at which x has an event now depend on the events on a and only the value of those depends on the previous value on x, not the current.

Additionally, the specification given in Example 4.6 is also well-formed, because it has a *delayed* labelled edge in each circle in the dependency graph.

Next we show that the well-formed constraint for TeSSLa solves the fixed-point problems we mentioned at the beginning of this section, as stated in the following theorem.

Theorem 4.9 (*Well-formed Specifications and Fixed-Points, [CHL+18]*)

Given a well-formed TeSSLa specification φ of equations $y_i := e_i$ and input streams $s_1, \ldots, s_k$, then the fixed point of $\varphi(s_1, \ldots, s_k)$ is unique.

Proof. Assume we have calculated the least fixed-point O and a second fixed-point P with $O \neq P$ for φ on input $s_1, \ldots, s_k$. Now P must be greater than O, because O is the least fixed-point. Additionally, assume φ is flat, because very TeSSLa specification can be transformed into a semantically equivalent one.

Each cycle on the dependency graph now corresponds to a certain subset of the expressions in φ. Because φ is well-formed, at least one of those entries must be either **last** or **delay**. We call this entry y_i. For both fixed-points, the value evaluated for y_i has to stay the same once the fixed-point is reached and in the case where y_i is the only equation with a **last** or **delay** in that cycle, it has to hold that the calculated fixed-point value for y_i for O has to be smaller than the one for P. But this is a contradiction to y_i being either an expression with **last** or **delay**, because both operators are defined such that they refine their input unless a fixed-point is reached: an output event at a timestamp t is calculated for both operators independently of their input streams at t regarding the input streams involved in the recursion, because the trigger streams can not be recursive (because φ is well-formed). Therefore, both operators output more progress than the input has. As mentioned before, both **last** and **delay** refine their outputs until the maximal progress is reached and also no other operator can cut the progress earlier, because all TeSSLa operators are monotonic. Overall this means that always the maximal progress of the whole recursion is calculated and therefore the maximum fixed-point of the Kleene-chain is always reached. □

4.3 Expressiveness of TeSSLa and the delay Operator

In this and the next section, we will take a look at the expressiveness of TeSSLa and state which types of functions TeSSLa can express. In this section, we take a closer look at the **delay** operator, while the following section will take a closer look at the **next** operator in TeSSLa^f. At the end of this chapter, an overview graphic is given in Figure Figure 4.3, representing all fragments considered and its relations to each other.

The **delay** operator is quite an interesting one in terms of TeSSLa because its only purpose is to schedule timeouts for events being outputted in the future.

At first, we consider TeSSLa without the **delay** operator and after that, we take a look at what the **delay** operators adds to TeSSLa.

4.3.1 TeSSLa without delay

TeSSLa without the **delay** operator, which we will call TeSSLa^{-d} for short, can do everything besides setting timeouts. The main ability this removes from TeSSLa is the one to output events with timestamps that did not occur on events in any of the input streams. The following theorem

states what kind of stream transformations can be expressed with TeSSLa that is missing the **delay** operator.

Before getting to the theorem stating the expressiveness, let us first state that continuity contains the fact that every prefix with a finite number of events of the result of a stream transformation can be calculated stepwise with only seeing a prefix of the input streams containing only a finite number of events. We call this property finitely progressive and then prove that it is included in continuity, hence, every stream transformation has it.

Definition 4.10 (*Finite Progressiveness*)

We call a stream transformation $f : \mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_n} \to \mathcal{S}_{\mathbb{D}'_1} \times \cdots \times \mathcal{S}_{\mathbb{D}'_m}$ finitely progressive *iff for all* $s_1, \ldots, s_n$ *with* $f(s_1, \ldots, s_n) = u_1, \ldots, u_m$ *and for all* $t_1, \ldots, t_m \in \mathbb{T}_\infty$ *with* $|u_i|_{t_i}| \in \mathbb{N}$*, there exists* $t'_1, \ldots, t'_n \in \mathbb{T}_\infty$ *with* $|s_i|_{t'_i}| \in \mathbb{N}$ *such that*

$$f(s_1|_{t'_1}, \ldots, s_n|_{t'_n}) = u'_1, \ldots, u'_m \Rightarrow \forall 1 \leq i \leq m : u_i|_{t_i} \sqsubseteq u'_i$$

To show the meaning of the previous definition, the following example explains how finite progressiveness works on a concrete set of input and output streams for the specification of the running example.

Example 4.11 (*Finite Progressiveness*)

Consider the streams specified in Figure 4.2 which show the stream transformation from the Examples 3.22 and 3.38 as well as the input and output streams cut at certain timestamps. In the formal definition of finite progressiveness, this stream transformation can be written as

$$f(\mathsf{values}, \mathsf{resets}) = \mathsf{cond}, \mathsf{sum}$$

The stream transformation is finitely progressive. To show that, we would need to cut the output at every possible pair of timestamps. As an example, we do it for one pair of timestamps here. We cut the output streams at timestamps 8 and 6, respectively, and the results are $\mathsf{cond}|_8$ and $\mathsf{sum}|_6$ as shown in the figure. The idea of finite progressiveness is now that all cuts of the output which result in a finite number of events, as the given cut is, are outputted by the stream transformation after seeing only a finite number of events on the input streams. Therefore, a cut has to exists on the input streams, outputting a suffix of $\mathsf{cond}|_8$ and $\mathsf{sum}|_6$. Such a cut is found when cutting the input streams at 9 and 10, respectively, therefore $\mathsf{values}|_9$ and $\mathsf{resets}|_{10}$, which

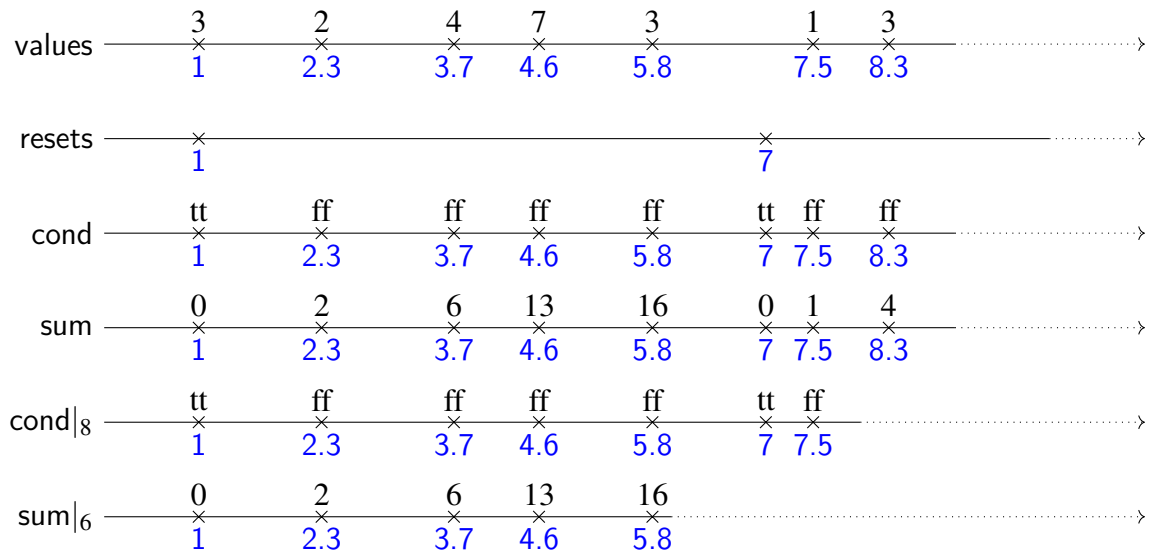


Figure 4.2: Figure 3.2 expanded by two cuts for cond and sum to show how finite progressiveness works. It can be seen that $\mathsf{cond}|_8$ and $\mathsf{sum}|_6$ are prefixes of cond and sum.

> results in the original input streams values and resets and output the original output streams cond and sum, which are a suffix of $\mathsf{cond}|_8$ and $\mathsf{sum}|_6$.

The following lemma states that every stream transformation is finitely progressive.

Lemma 4.12 (*Finite Progressiveness and Stream Transformations*)

> *Every stream transformation is finitely progressive.*

Proof. A function f is continuous iff $\bigvee f(D) = f(\bigvee' D)$, which means it is equivalent iff we take the supremum after calculating every output of f for a set of inputs D or calculating the supremum first and then apply f to it. Now consider streams $s_1, \ldots, s_n$ and all the prefixes of them. The supremum of the prefixes of one s_i is obviously s_i again. Because f is continuous, the outputs, when inputting all the prefixes in f, have to be the way such that the supremum of the outputs is the same as if we input $s_1, \ldots, s_n$ in f. This means especially, that for all prefixes of the output streams until a finite timestamp it has to hold that there have to be prefixes of the input streams until a finite timestamp such that, when inputting those prefixes, a suffix of the given output-prefixes has to be output. Otherwise, f would not be continuous, because $\bigvee f(D) = f(\bigvee' D)$ would not hold. But this is also the criteria for being finitely progressive, which means therefore it follows that, if f is continuous, it is also finitely progressive. Because every stream transformation is continuous, it is also finitely progressive. □

Now using the previous lemma, we can prove the following theorem, which states that TeSSLa^{-d} allows us to exactly express all stream transformations that are timestamp conservative and future independent.

To show that there is a behavioural equivalent stream transformation for all timestamp conservative and future independent stream transformations which can be expressed by TeSSLa, the proof uses a representation of the stream transformation as an iterative function, which is similar to how evaluation strategies work. Using this representation, the timestamp conservative and future independent stream transformation can be translated into a TeSSLa specification which first synchronizes the input events using a merge and applying the iterative function, while using a **last** to simulate the memory.

The other direction in the proof follows directly from the definitions of the two properties and the TeSSLa operators.

Theorem 4.13 (*Expressiveness of TeSSLa^{-d}, [CHL+18]*)

For every stream transformation $f : \mathcal{S}_{\mathbb{D}_1} \times \ldots \times \mathcal{S}_{\mathbb{D}_k} \to \mathcal{S}_{\mathbb{D}'_1} \times \ldots \times \mathcal{S}_{\mathbb{D}'_n}$ there is a behavioural equivalent stream transformation f' that can be represented as a TeSSLa^{-d} specification iff f is

- *timestamp conservative and*
- *future independent.*

Proof. First, we show that for every stream transformation f, a behavioural equivalent stream transformation f' exists, which can be represented as a TeSSLa specification, if f has the two mentioned properties.

It follows from Lemma 4.12 that the stream transformation f can be represented as the iterative function $\tilde{f} : \mathbb{M} \cup \{\bot\} \times (\mathbb{D}_1 \cup \{\bot\} \times \ldots \times \mathbb{D}_k \cup \{\bot\})^n \times \mathbb{T} \to \mathbb{M}$ with $\tilde{f}(m,d,t) = m'$, which takes a parameter m, called the memory state, the current input values d, and the corresponding current timestamp t and returns the new memory state m'. This holds because the lemma states that the output for a given set of input streams can be calculated step-wise, therefore, for every prefix created by cutting the output streams after a finite amount of time, there are also finite timestamps to cut the input streams such that a suffix of this prefix is outputted. This means that we will either reach the output for the set of input streams at some point or converge to it as being a supremum of the prefixes we calculated and that values we calculate step-wise will not be changed later.

Based on this memory state the function $\tilde{o} : \mathbb{M} \to \mathbb{D}'_1 \cup \{\bot\} \times \ldots \times \mathbb{D}'_n \cup \{\bot\}$ provides the output events for all output streams.

Because f is monotonic it is sufficient to compute the output events step by step and because f is future independent it is sufficient to allow $\tilde{f}$ to store arbitrary information about the past events. Additionally, because f is timestamp conservative it is sufficient to execute $\tilde{f}$ for every timestamp in the input events. Using this representation of f and the fact that we do not have to care about the progress because f' is behavioural equivalent to f, we can translate f' into the following equivalent TeSSLa specification with input variables $x_1, \ldots, x_k$ and output variables $y_1, \ldots, y_n$:

$$\begin{aligned} t &:= \mathbf{time}(\mathsf{merge}(x_1, \ldots, x_k, \mathbf{unit})) \\ m &:= \mathbf{lift}(\tilde{f})(\mathbf{last}(m,t), x_1, \ldots, x_k, t) \\ \forall_{i \leq n}\, y_i &:= \mathbf{lift}(\tilde{o}_i)(m) \end{aligned}$$

The **unit** is necessary to be able to have an event at timestamp 0. This shows the first direction.

Next, we show the other direction, therefore, that each TeSSLa^{-d} specification can be represented by a stream transformation f that is timestamp conservative and future independent.

This direction follows immediately from the fact that due to Lemma 4.2 TeSSLa is monotonic and continuous and from the fact that every TeSSLa^{-d} specification is timestamp conservative and future independent. This is true because besides the **delay**, no operator can create events at timestamps at which no input stream had an event, and thus timestamp conservatism follows, and because no operator at all in TeSSLa can access future information, from which future independence follows. □

In the following example we will consider a stream transformation which is timestamp conservative and future independent, but can not be expressed using TeSSLa^{-d}. This shows that the behavioural equivalence is really needed in the statement the previous theorem makes.

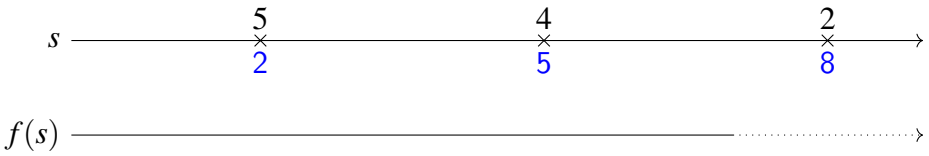
Example 4.14 (*Expressiveness of TeSSLa without* delay)

Consider the stream transformation $f : \mathcal{S}_{\mathbb{D}} \to \mathcal{S}_{\mathbb{D}'}$ with

$$f(s)(t) = \begin{cases} ? & \text{if } \exists t' < t : s(t') \in \mathbb{D} \wedge t \geq t' + s(t') \wedge \nexists t'' < t' : s(t'') \in \mathbb{D} \\ \bot & \text{otherwise} \end{cases}$$

This stream transformation never outputs an event, but instead only generates exclusive progress

until the timestamp which is the sum of the timestamp and the value of the first event. For a given input stream s it would generate the following output stream:



Because at timestamp 2 the first event occurs on s and its value is 5, the output stream has progress until timestamp 7.

This stream transformation can not be expressed using TeSSLa^{-d}, because there is no possibility to reference the timestamp 7 without having an event there. Still, f is future independent and timestamp conservative. But there is a behavioural equivalent stream transformation f' with

$$f'(s) = \textbf{nil}$$

which can obviously be expressed by TeSSLa.

The statement of the previous theorem is quite weak in terms of what kind of progress can be expressed using TeSSLa, because a behavioural equivalent function ignores every progress. But we can make a stronger statement regarding the progress of the behavioural equivalent stream transformations that TeSSLa can express, which is given in the following theorem.

Theorem 4.15 (*Progress of TeSSLa Without Delay*)

The progress of every output stream s for a stream transformation that can be expressed by TeSSLa^{-d} is at least the minimal progress of all input streams to which a path exist from s in the dependency graph of the given specification.

Proof. Because TeSSLa operators are compositional, we only need to show that the statement holds for every single operator, thus, every operator needs to output at least the minimal progress of its input streams. For **nil** and **unit** this does hold trivially, because these operators do not have input streams. For **time** it holds, because just the values of the events are changed, so the progress is just copied by the operator. The **lift** operator directly outputs ? if at least one of its input streams is ?, therefore it outputs exactly the minimum of the progress of its input streams. For the **last**,

according to the definition, the progress of the output stream can not end until the progress of at least one of the input streams ended. □

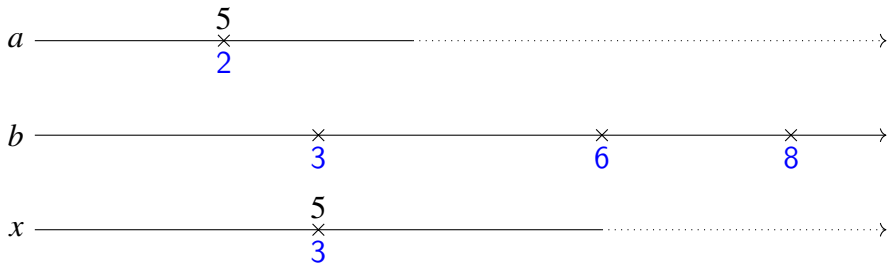
The previous theorem stated that every output stream of a TeSSLa^{-d} specification has at least as much progress as the minimum progress of the input streams it depends on is. The next example shows a stream transformation that has more progress than the minimal progress of the input streams without having complete progress like **nil** or **unit**.

Example 4.16 (*Progress of TeSSLa^{-d}*)

Consider a TeSSLa specification

$$x := \mathbf{last}(a, b)$$

where a and b are input streams. Depending on how the input streams behave, this specification may output more progress than the minimal progress of the input streams:



Even though a has only progress up to timestamp 4, if no trigger and therefore an event on b occurs, we know that we would not output anything. Thus, x stays $\perp$ until an event occurs on b at timestamp 6.

4.3.2 TeSSLa with delay

In the previous section we showed which kind of stream transformations TeSSLa without **delay** operator can express and how much progress it outputs at least. As next step, we now take a look at full TeSSLa (which is stated sometimes as TeSSLa with **delay** here to explicitly state that **delay** is now included). As stated before, without the **delay** only stream transformations which are timestamp conservative and future independent can be expressed by TeSSLa. The following theorem states that exactly the restriction of timestamp conservatism is lifted when the **delay** is included into TeSSLa.

The proof uses the same technique as the proof for Theorem 4.13 and adds a new timeout function inside the step function to allow for outputting new events at timestamps on the output streams which have not already been in the input streams. Using this step function and a **delay** operator, we can again represent a behavioural equivalent function f' for f as a generic TeSSLa specification.

Theorem 4.17 (*Expressiveness of TeSSLa With Delay, [CHL+18]*)

For every stream transformation $f : \mathcal{S}_{\mathbb{D}_1} \times \ldots \times \mathcal{S}_{\mathbb{D}_k} \to \mathcal{S}_{\mathbb{D}'_1} \times \ldots \times \mathcal{S}_{\mathbb{D}'_n}$ there is a behavioural equivalent stream transformation f' that can be represented as a TeSSLa specification iff f is

- *future independent.*

Proof. Again, because of Lemma 4.12, we can represent f as an iterative step-function. By removing the timestamp conservatism requirement we need to accompany the step-function $\tilde{f}$ and the output function $\tilde{o}$ from Theorem 4.13 with a timeout function $\tilde{u}$ which is evaluated on every new memory state and can return the timestamp of the next evaluation of $\tilde{f}$ if no input event happens before. One timeout function is sufficient because the step-function $\tilde{f}$ can perform arbitrary computations and store an arbitrary state in order to simulate multiple timeouts.

Using this representation and the fact that we do not have to care about the progress because f' is behavioural equivalent to f, we can translate f' into the following equivalent TeSSLa specification with input variables $x_1, \ldots, x_k$ and output variables $y_1, \ldots, y_n$:

$$
\begin{aligned}
t &:= \mathbf{time}(\mathsf{merge}(x_1, \ldots, x_k, \mathbf{unit}, \mathbf{delay}(d, \mathsf{merge}(x_1, \ldots, x_k, \mathbf{unit})))) \\
d &:= \mathbf{time}(t) - \mathbf{lift}(\tilde{u})(m) \\
m &:= \mathbf{lift}(\tilde{f})(\mathbf{last}(m,t), x_1, \ldots, x_k, t) \\
\forall_{i \leq n}\ y_i &:= \mathbf{lift}(\tilde{o}_i)(m)
\end{aligned}
$$

The **unit** operators are again needed to be able to have an event at timestamp 0 and to possibly start a timeout with the **delay** at timestamp 0.

The other direction follows again directly from Lemma 4.2 which states that TeSSLa is monotonic and continuous and the fact that also the **delay** is future independent. As every other TeSSLa operator, it is not able to access any future information. □

Informally, TeSSLa with **delay** can now also express stream transformations which are not timestamp conservative, but is still limited to stream transformations which are future independent, because

even the **delay** operator does not need any value from later inputs to output a value now and also is not able to consider future values.

In the following example we again consider the stream transformation from Example 4.14.

Example 4.18 (*Expressiveness of TeSSLa with* delay)

As an intuitive example, reconsider the stream transformation $f : \mathcal{S}_{\mathbb{D}} \to \mathcal{S}_{\mathbb{D}'}$ with

$$f(s)(t) = \begin{cases} ? & \text{if } \exists t' < t : s(t') \in \mathbb{D} \wedge t \geq t' + s(t') \wedge \nexists t'' < t' : s(t'') \in \mathbb{D} \\ \bot & \text{otherwise} \end{cases}$$

from Example 4.14. Even TeSSLa including the **delay** operator can not express f. Even though one could possibly try to set a timeout when the first event on s happens with the correct delay, we have no reset stream for the **delay** operator to trigger the event without modifying the progress of the output stream.

Next, we again make a stronger statement about the progress TeSSLa specifications output at least, this time including the **delay**.

Theorem 4.19 (*Progress of TeSSLa With Delay*)

The progress of every output stream s for a stream transformation that can be expressed by TeSSLa with **delay** *is at least the minimal progress of all input streams to which a path exist from s in the dependency graph of the given specification.*

Proof. As the **delay** is compositional as well, it remains to prove that it also outputs at least the minimum of the progress from its input streams. This is the case because, like the **last**, the **delay** is defined in a way such that it can not output a ? as long as no input stream had a ?. □

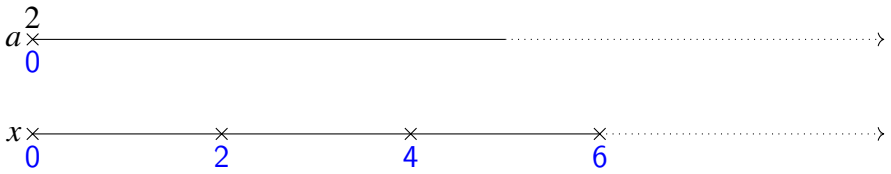
The previous theorem stated that also every output stream of a TeSSLa specification has at least as much progress as the minimum progress of the input streams it depends on is. The following example shows a stream transformation that has more progress than any of its input streams without having complete progress like **nil** or **unit**.

Example 4.20 (*Progress of TeSSLa With Delay*)

Consider a TeSSLa specification

$$x := \mathsf{merge}(\mathbf{delay}(\mathsf{const}(x, a), \mathbf{unit}), \mathbf{unit})$$

where a is an input stream. Now assume a to be the following stream:



Even though a has only progress up to timestamp 5, this specification creates progress up to timestamp 6, because without a reset happening, it does not matter for the **delay** if the first input stream still has progress or not if a timeout is already set.

Note that with the **delay** operator it is possible to construct Zeno streams because the timeout function is not restricted in any way. An example for this was already given with the second stream in Example 4.4. By Rice's theorem it is impossible to check for an arbitrary timeout function whether it only generates non-Zeno timestamp sequences, because this is a semantics property. Hence, one would need to restrict allowed timeout functions more drastically, which would restrict the possible event sequences generated by a TeSSLa specification further than necessary. For that reason we decided to include the capability to generate zeno streams with TeSSLa.

Next we state that it does not matter if we are allowed to use only one **delay** in a property or multiple. This follows as a consequence of Theorem 4.17 because the generic TeSSLa specification given there only needs one **delay** to express any given function.

Corollary 4.21 (*Specifications with Multiple* delay *Operators, [CHL+18]*)

A TeSSLa specification with multiple **delay** *operators can be translated into an equivalent specification with only one* **delay** *operator.*

TeSSLa with and without **delay** are closely related because TeSSLa without **delay** can verify the relation of given input/output streams with respect to a TeSSLa specification that uses the **delay** operator. Therefore, if we consider a TeSSLa specification φ, we can build a TeSSLa^{-d} specification ψ such that if we insert two sets of streams into ψ, it can output a boolean stream indicating if the second set would be the correct set of output streams of φ when inputting the streams from the first set. The **delay** is only needed to actively generate the events at specified times while TeSSLa without it can still check if the timeouts happened at the correct timestamps. In the following we want to state formally in a theorem that this can be done and show constructively in the proof how such a TeSSLa^{-d} specification ψ can be build for a given TeSSLa specification φ.

For this purpose, we denote with $\varphi|_y(x_1,\ldots,x_k) \in \mathbb{B}$ the boolean function indicating whether the boolean output stream $y \in \mathcal{S}_{\mathbb{B}}$ of the TeSSLa specification φ contains only events with value tt for the input streams $x_1,\ldots,x_k \in \mathcal{S}_{\mathbb{D}_1} \times \ldots \times \mathcal{S}_{\mathbb{D}_k}$.

Following Corollary 4.21, TeSSLa specifications with one or multiple **delay** operators are equally expressive, which we use in the proof of the following theorem. Furthermore, the following theorem is closely related to Theorem 4.13, which already proves that a specification φ' has to exist, but the following theorem also shows how such a specification can be build constructively.

Informally, the proof creates a specification which checks the correctness of the outputted events by the **delay** operator by keeping an eye on the reset and delay streams and checks the behaviour if the output stream of the **delay** has an event.

Theorem 4.22 (*Delay Elimination, [CHL+18]*)

For every TeSSLa specification φ with $\varphi \in \mathcal{S}_{\mathbb{D}_1} \times \ldots \times \mathcal{S}_{\mathbb{D}_k} \to \mathcal{S}_{\mathbb{U}} \times \mathcal{S}_{\mathbb{D}'_1} \times \ldots \times \mathcal{S}_{\mathbb{D}'_n}$ with **delay** *operators, there exists a TeSSLa specification φ' without a* **delay** *operator, which derives a boolean stream $z \in \mathcal{S}_{\mathbb{B}}$, such that for any given input streams $x_1,\ldots,x_k \in \mathcal{S}_{\mathbb{D}_1} \times \ldots \times \mathcal{S}_{\mathbb{D}_k}$ and any given output streams $y_1,\ldots,y_n \in \mathcal{S}_{\mathbb{D}'_1} \times \ldots \times \mathcal{S}_{\mathbb{D}'_n}$ and any stream $d \in \mathcal{S}_{\mathbb{U}}$ we have $\varphi(x_1,\ldots,x_k) = d,y_1,\ldots,y_n$ iff $\varphi'|_z(x_1,\ldots,x_k,d,y_1,\ldots,y_n)$ where d is the stream derived immediately through the* **delay** *operator.*

Proof. We know from Corollary 4.21 that TeSSLa specifications with multiple **delay** operators can be translated into specifications with only one **delay** operator. Thus we assume that φ has only one **delay** operator, otherwise we would transform it into one.

We construct the validating specification φ' from the generating specification φ. We show how to convert the TeSSLa specification φ, which derives new output streams, into the TeSSLa specification φ', which validates that the output streams were derived according to the specification φ. The specification φ' derives all the streams except d in the same way as φ does. The stream d was derived in φ using the **delay** operator and is provided as input to φ'. The specification φ' now validates if the derived streams are equivalent to the input streams $y_1,\ldots,y_n$. In order to check two streams for equivalence one has to assert the equivalence of the timestamps and the values separately, i.e. for every two streams $y,y' \in \mathcal{S}_{\mathbb{D}}$ we assert $\text{time}(y) = \text{time}(y') \wedge y = y'$, where $=$ denotes the lifted equal function on $\mathbb{D}$.

Next we have to assert the correct derivation of d. Assuming that φ contains $d = \textbf{delay}(a,r)$ we derive the effective delays in absolute time by distinguishing three cases:

- We accept the new delay on a when the old delay is due,
- we accept the new delay on a together with an event on r and
- events on r without a new delay on a can reset the effective delay to ∞.

These three cases are combined in the following TeSSLa specification a':

$$\begin{aligned} a' := \mathsf{merge}(\ & \mathsf{filter}(\mathbf{last}(a',a) = \mathbf{time}(a), \mathbf{time}(a)+a), \\ & \mathsf{filter}(\mathbf{time}(r) = \mathbf{time}(a), \mathbf{time}(a)+a), \\ & \mathsf{const}(\infty)(\mathsf{merge}(r, \mathbf{unit}))\ ) \end{aligned}$$

Now we assert at all events $t := \mathsf{merge}(a,r,d)$ that either the delay was fulfilled by an event on d, i.e. $\mathsf{merge}(\mathbf{time}(d), \mathsf{zero}) = \mathbf{last}(a',t)$ or the delay is not due and there was no event on d, i.e. $\mathbf{time}(t) > \mathsf{merge}(\mathbf{time}(d), \mathsf{zero}) \wedge \mathbf{last}(a',t) > \mathbf{time}(t)$. The boolean stream z is derived as conjunction of all these assertions. □

4.4 Expressiveness of TeSSLa with next

Compared to how TeSSLa is defined in [CHL+18], we added an extension of TeSSLa, called TeSSLa^f, in Section 3.3, which has one additional operator, a future version of **last**, the **next** operator. In this section, we will take a closer look at the power of this new operator which now allows future references, give some examples and show that the **next** adds additional expressive power, but does not replace the **delay**. Instead, it complements it, allowing to express even future independent stream transformations.

Informally, the **next** operator is not really increasing the number of properties that can be specified which also practically make sense. In an evaluation strategy, using the **next** operator only leads to the problem of waiting until the future event arrives and then print out the result with an older timestamp. This was not possible without the **next** operator, but because we now output old timestamps, we can instead just output the value at the timestamp it occurs by rewriting the formula, getting mainly the same result for every practical application.

Because TeSSLa was originally created with operators to have good and easy to check memory consumption, it does not naturally contain the **next** operator. Even though it increases expressiveness in general, it does break the property that every operator just needs to save at most one value

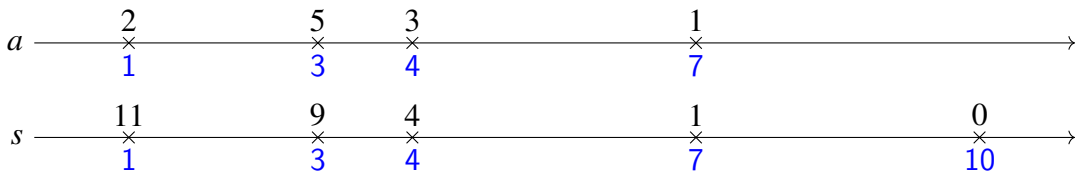
at any point in time, even if the data domain is bounded or the **next** operator is somehow restricted. This can be seen in the following example:

Example 4.23 (*Recursion with* next *Operator*)

Assume a TeSSLa^f formula

$$s := \textbf{next}(\mathsf{merge}(\mathsf{const}(\textbf{delay}(10,\textbf{unit}),0),s),a)+a$$

where a is an input stream. The **delay** takes care of setting a point up to which the **next** sums up the values on a. Which means, informally, every time an event occurs on a, its value is summed up until timestamp 10 is reached. At the first event on a, s will contain the sum of all the events on a until timestamp 10. For a given input stream a, this looks as follows:



When we now try to evaluate this formula incrementally, we have to remember every value arriving on a to compute the output after the stream has reached timestamp 10, because for every timestamp, we do not know the current value of the **next** yet. Only if the stream reaches the timestamp 10 we can infer the values for this subexpression. So the amount of values that needed to be saved in between does not only depend on the formula but on the length of the trace and the number of events arriving on the input streams.

Also, including the **next** operator into TeSSLa leads to other problems. While for TeSSLa well-formed specifications only depend on the fact that every cycle in the specification needs at least one **last** or **delay** operator included such that the cycle goes through their first parameter, this changes< when we add the **next**. Let us first define the dependency graph for a TeSSLa^f specification. Compared to the one for TeSSLa, the dependency graph for TeSSLa^f adds a labelling for edges involving the **next** operator.

Definition 4.24 (*Dependency Graph of TeSSLa^f Specifications, [CHL+18]*)

The dependency graph *of a flat TeSSLa^f specification φ of equations $y_i := e_i$ is the directed edge-labelled graph $G = (V,E)$ of nodes $V = \{y_1,\ldots,y_n\}$. For every $y_i := e_i$ the graph contains*

the edge $(y_i, y_j, l) \in E$ *every time* y_j *is used in* e_i *and*

$$l = \begin{cases} \text{delayed} & \text{if } \exists k : e_i = \textbf{last}(y_j, y_k) \vee e_i = \textbf{delay}(y_j, y_k) \\ \text{next} & \text{if } e_i = \textbf{next}(y_j, y_k) \\ \varepsilon & \text{otherwise} \end{cases}$$

such that edges are labelled corresponding to the first argument of **last** *or* **delay** *with* delayed*, or with* next *in case a* **next** *is involved.*

Using the dependency graph, we now define the notion of well-formedness for TeSSLa^f specifications.

Definition 4.25 (*Well-formedness of TeSSLa^f Specifications*)

We call a TeSSLa^f specification φ well-formed *if every cycle of the dependency graph contains*

- *at least one edge labelled with* delayed *and*
- *no edge labelled with* next.

As one can see, we forbid any cyclic use of the first parameter of the **next** operator in well-formed TeSSLa^f specifications. This is because the **next** is not completely the future counterpart of **last**, as we have no definitive ending in our streams. For the **last**, the point with timestamp 0 is the definitive beginning of a stream and with this, gives us knowledge we do not have for the **next**. Because of this missing definitive ending, nearly every cyclic use of the first parameter of a **next** can have multiple fixed-points depending on the input streams as long as it does not have any mechanism of breaking the recursion. Hence, the well-formed fragment can not be enforced with a weaker syntactic restriction.

Note that, if we would add a definitive ending point in our streams like the timestamp 0 is for the beginning, the **next** would be an exact counterpart of the **last** and cycles with **next** could be allowed in well-formed specification for the most cases, in the same way a cyclic usage of **last** is allowed. This is exactly why the well-formedness criteria for LOLA [DSS$^+$05], which also includes future references, is less restrictive, because there only streams with a finite ending timestamp are considered.

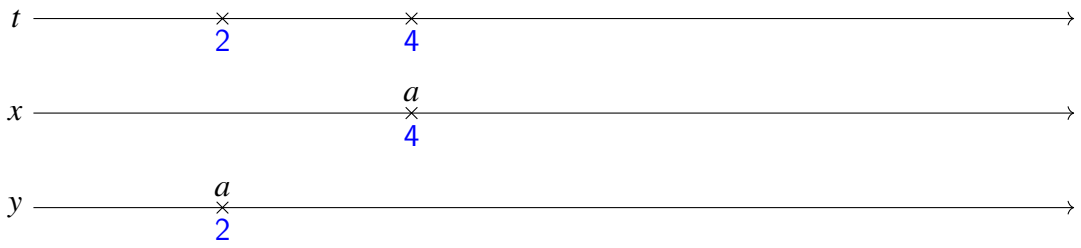
We depict some cases for not well-formed TeSSLa^f specifications in the following example.

Example 4.26 (*Well-formedness of TeSSLa^f*)

Let us consider at first the following TeSSLa^f specification with an input stream t:

$$x = \mathbf{last}(y,t)$$
$$y = \mathbf{next}(x,t)$$

This specification represents a cyclic usage of **last** and **next**, triggered by the input stream t and is not well-formed. Additionally, it has possibly multiple fixed-points, for example is has always the fixed-point where x and y are always $\bot$ and it can have, for example, the following fixed-points:



where a can now be any value which leads to a possibly infinite number of different fixed-points. The same can happen to a specification

$$x = \mathbf{last}(z,t)$$
$$y = \mathbf{next}(x,s)$$
$$z = \mathbf{next}(y,s)$$

with input streams s and t, where, even though the number of **last** and **next** operators is different, multiple fixed-points can occur if the number of events on the two input streams matches a certain pattern.

The following theorem states that well-formed TeSSLa^f specifications only have one fixed-point. It follows the same idea as the proof for Theorem 4.9, where it was already shown that well-formed TeSSLa specifications only have one single fixed-point. Therefore, it remains to extend the proof to include the newly in TeSSLa^f introduced **next** operator. Because the **next** operator is not allowed in any recursive usage in a well-formed specification, only non-cycles paths through **next**s have to be considered.

Theorem 4.27 (*Well-formed Specifications and Fixed-Points, [CHL+18]*)

Given a well-formed TeSSLa^f specification φ of equations $y_i := e_i$ and input streams $s_1, \ldots, s_k$, then the fixed point of $\varphi(s_1, \ldots, s_k)$ is unique.

Proof. Assume we have calculated the least fixed-point O and a second fixed-point P with $O \neq P$ for φ on input $s_1, \ldots, s_k$. Now P must be greater than O, because O is the least fixed-point. Additionally, assume φ is flat, because every TeSSLa^f specification can be transformed into a semantically equivalent one.

From Theorem 4.9 we know the fixed-point is unique for well-formed TeSSLa specifications. The addition in TeSSLa^f is the **next**, but in this case it is not allowed in any recursive way, because φ is well-formed. As **last** and **delay**, **next** also calculates as much progress as possible. Because it is not allowed to occur recursively, it always calculates the maximal fixed-point for any input. Therefore, still the maximal progress of every recursion is calculated and the greatest fixed-point of the Kleene-chain is always reached. □

In the following sections, we will take a look at the expressiveness the **next** operator adds to TeSSLa.

4.4.1 TeSSLa^f without delay

In this section, we will consider the fragment of TeSSLa^f without the **delay** operator, or TeSSLa^{f-d} for short. Removing the **delay** operator removes the property of expressing non-timestamp conservative stream transformations in TeSSLa. We now show how the addition of the **next** operator in TeSSLa^f impacts the resulting expressiveness.

We show that TeSSLa^{f-d} is more expressive than TeSSLa^{-d} first. Afterwards, we show that TeSSLa^{f-d} and TeSSLa are incomparable. The proof of the following theorem uses a concrete stream transformation which can be expressed by TeSSLa^{f-d} but not by TeSSLa^{-d} to show that TeSSLa^{f-d} is more expressive. The other direction follows trivially, as TeSSLa^{f-d} contains every operator from TeSSLa^{-d} plus the **next**.

Theorem 4.28 (*Expressiveness of TeSSLa^f Without* delay)

It holds that TeSSLa$^{f-d} \supsetneq$ TeSSLa^{-d}.

Proof. Obviously, TeSSLa$^{f-d} \supseteq$ TeSSLa^{-d} holds because TeSSLa^{f-d} has all the operators from TeSSLa^{-d}. It remains to show that the **next** allows us the expression of additional stream transformations.

Consider the stream transformation $f : \mathcal{S}_{\mathbb{D}} \to \mathcal{S}_{\mathbb{D}}$ with

$$f(s)(t) = \begin{cases} d' & \text{if } s(t) \in \mathbb{D} \wedge \exists t' > t : s(t') \in \mathbb{D} \wedge s(t') = d' \wedge \nexists t < t'' < t' : s(t'') \in \mathbb{D} \cup \{?\} \\ ? & \text{if } s(t) \in \mathbb{D} \wedge \exists t' > t : s(t') = ? \wedge \nexists t < t'' < t' : s(t'') \in \mathbb{D} \\ s(t) & \text{otherwise} \end{cases}$$

while TeSSLa^{f-d} can express this stream transformation with the following specification as stream s', where s is an input stream

$$s' = \mathbf{next}(s, s)$$

TeSSLa^{-d} can not express this stream transformation, because it is not able to consider values from future events for the calculation of the value for the current timestamp. Therefore, TeSSLa$^{f-d} \supsetneq$ TeSSLa^{-d} holds. □

The theorem shows that the **next** is adding the possibility to express more stream transformations if we have no **delay** operator. As we have shown in Theorem 4.13 and Theorem 4.17, the **delay** adds the possibility to express more stream transformations. In the following theorem, we show that both operators add other stream transformations than the **next** to the ones already expressible by TeSSLa^{-d} and therefore, that TeSSLa^{f-d} and TeSSLa are incomparable. Again, the proof uses two concrete stream transformations both of which can only be express by one of the TeSSLa^{f} fragments to show that TeSSLa^{f-d} and TeSSLa are incomparable. One of them is again the stream transformation from the previous proof.

Theorem 4.29 (*TeSSLa^{f} Without* delay *versus TeSSLa*)

It holds that TeSSLa^{f-d} and TeSSLa are incomparable.

Proof. To show that TeSSLa^{f-d} and TeSSLa are incomparable, we have to show that TeSSLa^{f-d} $\not\supseteq$ TeSSLa and TeSSLa$^{f-d} \not\subseteq$ TeSSLa hold.

To show that TeSSLa$^{f-d} \not\subseteq$ TeSSLa holds, we can use the example stream transformation from Theorem 4.28 again. Even with **delay**, TeSSLa can not express this stream transformation.

To show that TeSSLa$^{f-d} \not\supseteq$ TeSSLa does hold on the other hand, consider the following stream transformation $f : \mathcal{S}_{\mathbb{N}} \to \mathcal{S}_{\mathbb{N}}$ with

$$f(s)(t) = \begin{cases} 3 & \text{if } t = 5 \\ \bot & \text{otherwise} \end{cases}$$

TeSSLa can express this stream transformation as stream s', where s is an input stream, as follows:

$$s' = \mathsf{merge}(\mathbf{nil}, \mathbf{delay}(\mathsf{const}_5(\mathbf{unit}), \mathbf{unit}))$$

TeSSLa^{f-d} can not express this stream transformation, as it is not able to set a timeout to timestamp 5 and therefore can not output an event there, if no input event exists at that timestamp.

Hence it holds that TeSSLa^{f-d} and TeSSLa are incomparable. □

Even though we were able to make statements about the expressiveness of TeSSLa^{f-d} in the previous two Theorems 4.28 and 4.29, it was also possible to extend the result of Theorem 4.13 for the progress made by TeSSLa in Theorem 4.15. This statement does not hold anymore in TeSSLa^{f}. The **next** operator allows us to look in the future and if there is no further event, the progress of the stream defined by the **next** ends earlier than the input stream. Thus, the progress of the output streams does not depend anymore on the progress of the input streams.

4.4.2 TeSSLa^{f} with delay

In the previous sections, we considered TeSSLa without **delay**, with **next** and without **delay** but with **next**. In this section, we add back the **delay** operator and now consider full TeSSLa^{f} and its expressiveness. While it is obvious that the union of TeSSLa and TeSSLa^{f-d} is more expressive than any of its single parts due to the fact that both languages are incomparable as shown in Theorem 4.29, we will show in the following theorem that having both operators in one language adds even more to the expressiveness, therefore, that TeSSLa^{f} is more expressive than TeSSLa and TeSSLa^{f-d}. This means that TeSSLa^{f} can also express stream transformations which are neither timestamp conservative nor future independent.

The proof for the following theorem again uses a concrete stream transformation to show the results. This one is exactly build the way such that it creates events at timestamps where none have been before are well as it is considering future values at a given timestamp.

Theorem 4.30 (*Expressiveness of TeSSLa^f*)

The following two statements hold:

- *TeSSLa^f $\supsetneq$ TeSSLa and*
- *TeSSLa^f $\supsetneq$ TeSSLa^{f-d}.*

Proof. First, stating that TeSSLa^f $\supseteq$ TeSSLa and TeSSLa^f $\supseteq$ TeSSLa^{f-d} directly follows from the fact that TeSSLa^f contains both operators, **delay** and **next**. It remains to show that both inclusions are strict.

Consider the stream transformation $f : \mathcal{S}_{\mathbb{D}} \to \mathcal{S}_{\mathbb{D}}$ over time domain $\mathbb{R}$ with

$$f(s)(t) = \begin{cases} d & \text{if } t \in \mathbb{N} \wedge \exists t' > t : s(t') \in \mathbb{D} \wedge s(t') = d \wedge \nexists t < t'' < t' : s(t'') \in \mathbb{D} \cup \{?\} \\ ? & \text{if } t \in \mathbb{N} \wedge \exists t' > t : s(t') = ? \wedge \nexists t < t'' < t' : s(t'') \in \mathbb{D} \vee \forall t' > t : s(t') = \bot \\ \bot & \text{otherwise} \end{cases}$$

TeSSLa^f can express this stream transformation with the following specification as stream s', where s is an input stream:

$$x := \mathbf{delay}(\mathsf{const}_1(\mathsf{merge}(x, \mathbf{unit})), \mathbf{unit})$$
$$s' := \mathbf{next}(s, x)$$

Neither TeSSLa nor TeSSLa^{f-d} is able to express f. First, a **next** is necessary to look for future events on s and get the value of the next event, because otherwise, f would per definition not output an event. Also, the **delay** is needed to create the periodic events at every integer timestamp, as no other TeSSLa^f operator is able to output events at timestamps that have not been in any of the input streams. Therefore, an input stream for the **next** is needed which is coming from a **delay**, thus both operators need to be nested to express f. And because TeSSLa and TeSSLa^{f-d} only contain **next** or **delay**, but not both, and can therefore not mix them in one specification, TeSSLa^f $\supsetneq$ TeSSLa and TeSSLa^f $\supsetneq$ TeSSLa^{f-d} holds. □

The proof follows the same idea as the ones for Theorem 4.28 and Theorem 4.29, giving a stream transformation that the one fragment can express, while the other can not. Because the given stream transformation needs to create events at every integer timestamps, independently of the events in the input streams, as well as outputting the next value on the input stream s on those

events, **next** and **delay** have to be combined to express the stream transformation and can not do this independently. This means that a union of TeSSLa and TeSSLa^f without **delay** would not be enough to express it.

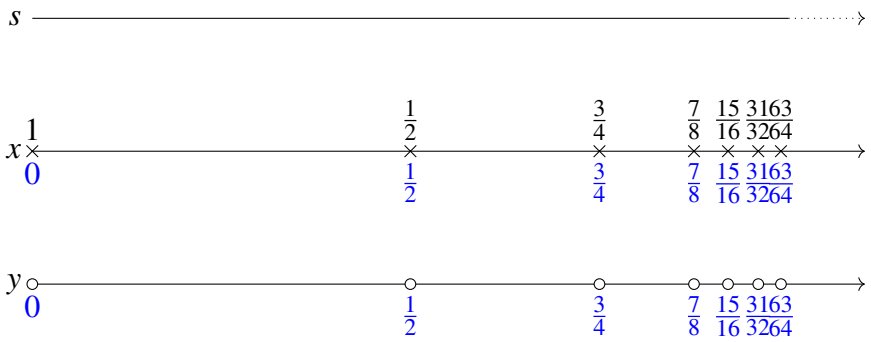
As for Theorem 4.28, we will not make any statement about the progress, because the addition of **next** removes the property we had before for TeSSLa. Additionally, using the **delay** and **next** in combination, some stream transformations with strange behaviour can be specified, for example one with no progress only at some timestamps, as seen in the following example.

Example 4.31 (*Progress in TeSSLa^f*)

Reconsider the specification from Example 4.4. We will extend this specification in this example with another equation y using **next** as follows:

$$x := \mathsf{merge}\left(\frac{\mathbf{last}(x, \mathbf{delay}(x, \mathbf{unit}))}{2}, 1\right)$$
$$y := \mathbf{next}(s, x)$$

where s is an input stream. x still always divides the last value on itself by 2 when the **delay** emits an event, hence it always halves the next delay value. The new stream y now always outputs the next value on s when an event occurs on x. If s has no events but ends at some point, y is a stream which has exactly no progress where x has an event.



4.5 Conclusion

Besides the results on well-formedness and computability, we considered four fragments and extensions of TeSSLa and their expressiveness in this chapter. These were obtained by removing

the **delay** operator or adding the **next** operator for the extension TeSSLa^f of TeSSLa. While the removal of other operators is not delivering interesting results or does not remove core functionality from TeSSLa, like the **lift** operator, these two operators contain some special functionality within TeSSLa which adds distinct features, as we saw in the previous two sections. This section concludes the results on these versions of TeSSLa.

The graph given in Figure 4.3 shows an overview of the four versions of TeSSLa that have been considered in this chapter. While adding either the **delay** or the **next** adds something to the expressiveness, either the ability to express non-timestamp conservative or non-future independent stream transformations, the two resulting versions are incomparable. Adding both operators leads to the highest expressiveness, even more expressiveness than the union of the fragments without **next** or without **delay**. The diagram also states how we were able to make statements about the amount of progress a specification outputs at least if it does not contain a **next** operator, as these statements can not be made anymore if a **next** is present in the specification.

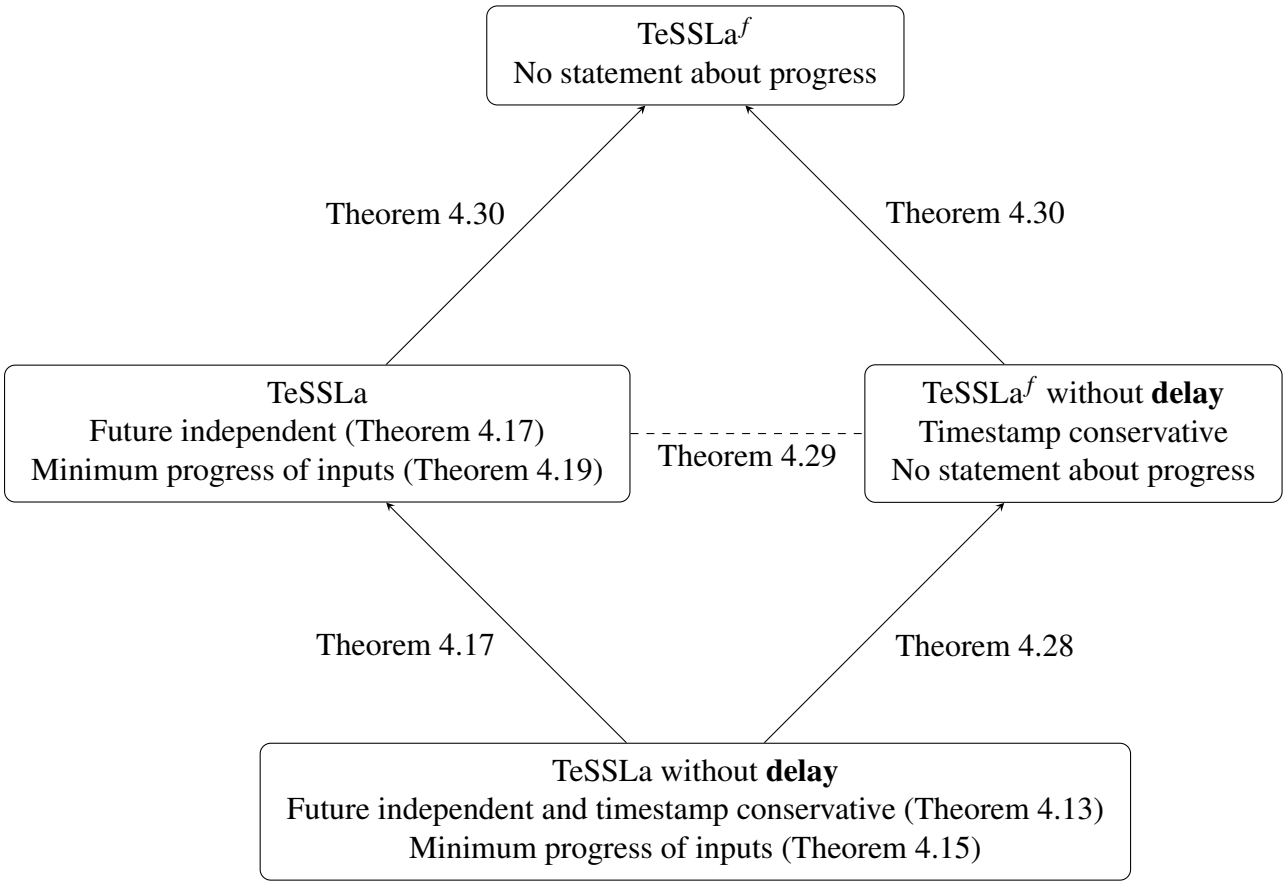


Figure 4.3: Shows the results of Chapter 4 regarding TeSSLa fragments and extensions. The arrows indicate that the TeSSLa version where the arrow ends is more expressive than the version where the arrow starts. The dashed line indicates that the two versions are incomparable.

5 TeSSLa Fragments and Relation to Transducers

Contents

The comparison of formalisms to existing ones is important. Besides getting an overview of the advantages and disadvantages of different languages, one gets complexity results on decision problems like equivalence as well as the memory related decision properties mentioned earlier in this thesis. Another point of interest is to see what a formalism can express and how the previously mentioned results are for different parts, or fragments, of a formalism.

In this chapter, we get results on the previously mentioned aspects regarding TeSSLa. We will take a look at the relationship to different languages and automata, properties of different fragments and an extension for future references within TeSSLa on continuous time domains.

In the following we will look at different results on streams with a continuous time domain. We will compare TeSSLa and TeSSLa^f and different fragments of it with various other formalisms, mostly transducers which are related to different classes of well known automata or logics. Besides results regarding expressiveness we will obtain results regarding the complexity and decidability of different decision problems. Besides emptiness and equivalence, we also take a look at various properties regarding memory requirements when evaluating a formula.

Before we get to the different sections about the different TeSSLa fragments and extensions, we have to first make statements about transducers in general and the streams we consider in this section. To compare the different types of transducers to TeSSLa, we will present transformations of streams into finite or infinite (timed) words and vice versa by synchronizing the input streams of the TeSSLa specification. While therefore it can be stated that transducers do represent stream transformations, they can in general also express functions that are not stream transformations, thus are not monotonic and or not continuous. This can only happen if we consider streams with an possibly infinite number of events, as it is then possible that a non-deterministic transducer than decides where a non-deterministic choice takes him depending on what happens in the infinity, for example, it guesses that it will always see an a in the future.

In the following proposition, we will state this and show an example of such a transducer afterwards.

Proposition 5.1 (*Transducers over Infinite Words and Continuity*)

(Functional) NFSTs and NTFSTs are not continuous over words with a possibly infinite number of symbols.

A transducer that is not continuous over a word with an infinite number of symbols is depicted in the following example.

Example 5.2 (*Non-continuous Transducer*)

Consider the NFST given in Figure 5.1. When reading an infinite word, in this NFST, the decision if an x or a y is outputted on the first a depends on if all other symbols are either bs or cs, respectively. This can only be decided by knowing the complete, infinite word, which means that this NFST is not continuous.

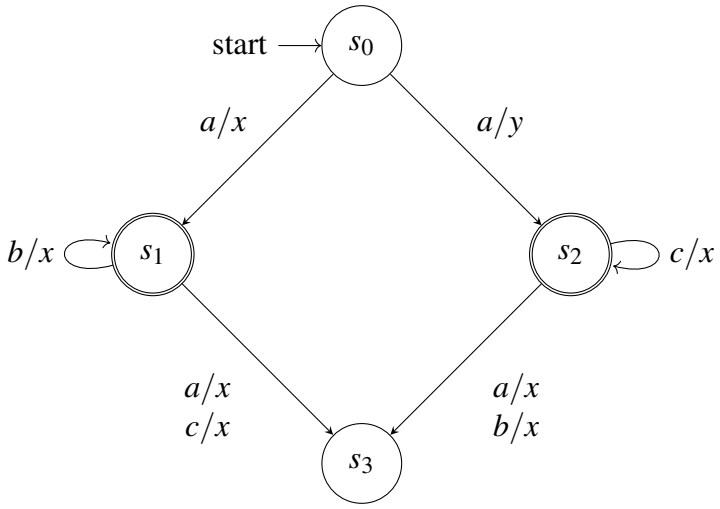


Figure 5.1: An NFST with four states. It accepts every word that contains an a first and forever b or forever c afterwards. Depending if it reads forever b or forever c, it outputs an x or a y when reading the first a.

The following proposition states that over finite words, all transducers considered in this thesis are continuous.

Proposition 5.3 (*Transducers over Finite Words, Monotonicity and Continuity*)

DFSTs, DTFSTs, DPTs, NFTSs and NTFSTs are monotonic and continuous over finite input words.

Because of Propositions 5.1 and 5.3, we will only consider streams with a finite number of events but still arbitrary, possibly infinite, progress in this chapter, unless noted otherwise. Normally, the question arises what we do with **delay** when we only consider streams with a finite number of events, because **delay** can potentially still create streams with an infinite number. But because the **delay** operator is not part of the fragments we consider in this chapter, this problem does not arise. Additionally, we only consider well-formed specifications in this chapter. Even though most of the transducer constructions also work with specifications that are not well-formed, it leads to some edge cases where transducers would have no states or no transitions, even though it should be able to output anything.

Before we get to the interesting fragments of TeSSLa and TeSSLa^f, we define an evaluation strategy for a specification first.

5.1 An Evaluation Strategy for TeSSLa

Besides equivalence, we want to consider three other properties of TeSSLa specifications in this chapter, namely if every specification of a fragment is finite memory under a given evaluation strategy and if not, how hard it is to decide for a given specification if it is finite memory under a given evaluation strategy (finite memory problem, FMP) as well as if there exists an evaluation strategy such that the given specification can evaluated using only finite memory (rewrite to finite memory problem, RFM). For the question if a fragment is finite memory as a whole and for FMP, a concrete evaluation strategy is necessary. In this thesis, we want to focus on an evaluation strategy which reflects an intuitive and straight forward way to evaluate a TeSSLa specification. This means an evaluation of a specification in a compositional way by evaluating every operator by its semantics without considering the semantics of its surroundings and forwarding its output events to the following operators according to the dependency graph.

We now define an evaluation strategy E_{TeSSLa} for TeSSLa by using the prefix semantics. In short, the evaluation strategy for TeSSLa works in a compositional way, thus it just evaluates every TeSSLa operator locally for itself, without considering the semantics of the operators it depends on or that are depending on the operator. Therefore, it directly applies the prefix semantics for each operator when evaluating an operator for given input streams. When an operator produces an output for an input, the output is forwarded to the following operators. Furthermore, E_{TeSSLa} synchronizes the input in the way already described before the definition of evaluation strategies: All inputs with the same timestamp are forwarded to the specification at once and the inputs with the following timestamp are not forwarded to the specification until the calculation for the given inputs with the prior timestamp is completely done. Note that we do not consider the **delay** operator for the evaluation strategy as the fragments considered in this chapter do not contain the **delay**.

Definition 5.4 (*Compositional Evaluation Strategy for TeSSLa*)

The evaluation strategy $E_{TeSSLa} = (m, o)$ with a memory function m and an output function o evaluates a TeSSLa formula by flattening it and then evaluating every single equation, and therefore every single operator, according to the prefix semantics as if the equation would be a TeSSLa formula on its own, hence in a compositional way. After evaluating each equation for arriving input events at a given timestamp, the outputted events are forwarded as events on the input streams to the following equations corresponding to the dependencies in the dependency graph of the formula.

This is done by the memory function m for each operator in the dependency graph according to

the prefix semantics by taking the current input values and the current memory state M. After finishing the calculation for the input for an operator in the dependency graph the function m outputs a new memory state M' to store values for later inputs, which are the values that have to be remembered for each **last** *operator.*

The function o generates the corresponding output of the specification based on the current input values and the current memory state M.

Because no **delay** operators exist in a specification evaluated by E_{TeSSLa} and the inputs come in synchronized by their timestamps (the inputs for a given timestamp all are available at once), the cyclic dependencies in the specification are resolved by writing the values remembered for a **last** to the memory. No special handling of a cycle is necessary, as it is automatically evaluated correctly because the necessary values are remembered and used again when the next input values arrive. Therefore, E_{TeSSLa} calculates the correct output for a given TeSSLa specification.

E_{TeSSLa} is obviously not optimal in time as there are other, more time efficient evaluation strategies, but because we only consider the memory usage in this thesis, it fits well for our purposes. Furthermore, it describes the typical, intuitive and straight-forward way of evaluating a TeSSLa specification, by just evaluating the operators locally one by one, corresponding to their definition.

Further, note that E_{TeSSLa} is also not optimal in its memory usage. As described at the end of Chapter 2, we are considering two decision problems regarding memory usage: FMP and RFM. While FMP asks whether the evaluation of a property is finite memory under a certain evaluation strategy, we will use E_{TeSSLa} to state whether the formula can be evaluated with only finite memory in a compositional and somehow direct way, without any preprocessing or minimization. RFM on the other hand asks, whether there is a possibility to evaluate it using only finite memory, which corresponds to creating an memory-wise optimal, semantically equivalent, STM and ask whether this STM only needs a finite number of memory cells. Thus this would be an optimal evaluation strategy regarding memory usage.

Example 5.5 (*Compositional Evaluation Strategy, FMP and RFM*)

Consider the flat TeSSLa specification φ which is given as follows for an input stream $s \in \mathcal{S}_{\mathbb{N}}$:

$$a := \textbf{last}(s, s)$$
$$b := \mathsf{const}(a, 1)$$
$$c := \textbf{lift}(+)(b, b)$$

> E_{TeSSLa} would now evaluate the stream a first for every input occurring on s, because b and c depend on a. Therefore, it evaluates $\textbf{last}(s,s)$. This is done by writing the current value on s into memory M and outputting the previous value saved. Then it forwards the output generated into $\mathsf{const}(a,1)$, to evaluate b, mapping the value from a to 1 and forwarding it to c. By then evaluating $\textbf{lift}(+)(b,b)$, therefore adding 1 to 1, it would generate 2 as the output of the specification for the current timestamp. Then, it repeats this evaluation mechanism for every incoming event.
>
> E_{TeSSLa} would need an unbounded, therefore not finite amount of memory, because when evaluating a it would write the current value of s, which may be arbitrarily large, into its memory, even though the value is never used, because it is mapped to 1 in b. Therefore, FMP would not be fulfilled for this formula under E_{TeSSLa}.
>
> Still, RFM is fulfilled, because there exists an evaluation strategy which is finite memory. This can be done by, for example, erasing the equation a and directly putting s into b in place of a, ignoring its first event to keep the semantics of the **last**. Then, the value on s would be ignored, while still outputting 2 every time an event arrives on s after the first.

As evaluation strategy, we also use E_{TeSSLa} for TeSSLa^f. We just extend it with the usage of **next**, which is also evaluated locally like every other operator according to its semantics. Compared to the other operators, **next** may remember timestamps and output multiple events later as soon as it knows the value.

Definition 5.6 (*Compositional Evaluation Strategy for TeSSLa^f*)

> *For TeSSLa^f, we extend E_{TeSSLa} by the* **next** *operator. An expression* $\textbf{next}(a,b)$ *is evaluated as follows: Every time an event occurs on b, the memory function m writes its timestamp to the memory M. Every time an event occurs on a, the timestamps remembered in M are deleted from M and output by the output function o in order with the value of the event on a.*
>
> *Additionally, the operators* **lift** *and* **last** *obtain the possibility of writing events into the memory, if an input depends on a* **next** *and the* **next** *writes the current timestamp into the memory, as well as being able to output multiple events in the order of the memorized timestamps at once when the* **next** *outputs an event.*

In the following, we always write E_{TeSSLa}, independently from the fact if we consider TeSSLa or TeSSLa^f, as the extension of E_{TeSSLa} for TeSSLa^f just adds a strategy for the **next** operator which is simply ignored when evaluating a TeSSLa formula.

5.2 Boolean Fragment

In this section we show a fragment of TeSSLa, called TeSSLa$_{\text{bool}}$, as defined in [CHL$^+$18], which fits to deterministic finite-state transducers. It is of interest because it resembles the regular languages and shows the expressive power of the TeSSLa operators when restricting the data domain. While timestamps are technically still in the input streams, they can not be used by this fragment and besides the order of arrival, time plays no role.

The fragment TeSSLa$_{\text{bool}}$ restricts TeSSLa to boolean streams and the operators **nil**, **unit**, **last** and **lift** with boolean functions and therefore removes **time** and **delay**.

Definition 5.7 (*TeSSLa$_{bool}$, [CHL$^+$18]*)

A TeSSLa formula φ is called a TeSSLa$_{bool}$ formula if $\varphi : \mathcal{S}_{\mathbb{B}} \times \ldots \times \mathcal{S}_{\mathbb{B}} \to \mathcal{S}_{\mathbb{B}} \times \ldots \times \mathcal{S}_{\mathbb{B}}$ and the syntax of every equation e is restricted as follows, where $f : \mathbb{B}^n_\perp \to \mathbb{B}_\perp$:

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

The semantics for every single operator stays the same as in TeSSLa.

First, let us add an observation concerning the relation of input and output streams regarding the order of their events. Since one can not access timestamps in this fragment, for a TeSSLa$_{\text{bool}}$-formula φ and two tuples of input streams $S, S' \in \mathcal{S}_{\mathbb{B}} \times \ldots \times \mathcal{S}_{\mathbb{B}}$ we have that $\varphi(S)$ and $\varphi(S')$ have events with the same values in the same global order iff all events in S' carry the same values in the same global order as those in S, independent from the exact timestamps of the events.

Note that in [CHL$^+$18], this TeSSLa fragment was defined with an additional operator which consists of an slift with $\geq$ on timestamps, which in the syntax was an expression of the following form:

$$\mathsf{slift}(\geq)(\textbf{time}(e), \textbf{time}(e))$$

It was added to the fragment in [CHL$^+$18] to show that even using **time** in this restricted way can also be used in TeSSLa$_{\text{bool}}$. In this thesis, we will not add $\mathsf{slift}(\geq)(\textbf{time}(e), \textbf{time}(e))$, but instead show that is does not change the expressiveness at all and that it can be expressed with the operators already existing in TeSSLa$_{\text{bool}}$.

Lemma 5.8 (*slift*$(\geq)(\textbf{time}(e), \textbf{time}(e))$ *and TeSSLa$_{bool}$*)

slift$(\geq)(\textbf{time}(e), \textbf{time}(e))$ *can be expressed with the TeSSLa$_{bool}$ operators.*

Proof. Consider a function $f : \mathbb{B}_\bot \times \mathbb{B}_\bot \to \mathbb{B}_\bot$ with

$$f(a,b) = \begin{cases} \mathrm{tt} & \text{if } a \in \mathbb{B} \\ \mathrm{ff} & \text{if } a = \bot \wedge b \in \mathbb{B} \\ \bot & \text{otherwise} \end{cases}$$

Then it holds that

$$\mathsf{slift}(f)(a,b) = \mathsf{slift}(\geq)(\mathbf{time}(a), \mathbf{time}(b))$$

Because slift can be expressed using merge, **lift** and **last** as defined in Chapter 3, subformulas of the form $\mathsf{slift}(\geq)(\mathbf{time}(a), \mathbf{time}(b))$ can be replaced with operators from $\text{TeSSLa}_{\text{bool}}$ such that the formula stays semantically equivalent. □

Note that even if $\mathsf{slift}(\geq)(\mathbf{time}(a), \mathbf{time}(b))$ would be added, for a $\text{TeSSLa}_{\text{bool}}$-formula φ and two tuples of input streams $S, S' \in \mathcal{S}_\mathbb{B} \times \ldots \times \mathcal{S}_\mathbb{B}$ we still have that $\varphi(S)$ and $\varphi(S')$ have events with the same values in the same global order iff all events in S' carry the same values in the same global order as those in S, independent from the exact timestamps of the events.

At first we want to show that $\text{TeSSLa}_{\text{bool}}$ resembles a certain type of transducers, that it is finite memory and what the complexity of the equivalence problem is. To show these properties, we make a statement about the relationship of DFSTs and $\text{TeSSLa}_{\text{bool}}$.

To show that $\text{TeSSLa}_{\text{bool}}$ and DFSTs have the same expressiveness, we encode DFST words as $\text{TeSSLa}_{\text{bool}}$ streams and vice versa. To do this, we are using a one-hot encoding such that for every symbol of the alphabet, a stream exists and for every given position of the word, only the stream representing the corresponding proposition is true and the others are false.

Formally, we use two functions α_Σ and β_Σ. The function $\alpha_\Sigma(w) = S$ encodes a DFST word $w = w_0 w_1 \ldots w_n \in \Sigma^*$ as a corresponding set of $\text{TeSSLa}_{\text{bool}}$ streams. For every $p \in \Sigma$ a stream $s_p \in S$ exists with

$$s_p = 0 d_0 1 d_1 \ldots n-1 d_{n-1} n d_n \infty \Leftrightarrow \forall i : (d_i \Leftrightarrow w_i = p)$$

Thus for every proposition a stream exists in S and for every symbol w_i of w an event exists on every stream from S with the value true if $w_i = p$ and false otherwise.

The function $\beta_\Sigma(s_1, \ldots, s_k) = w = w_0 w_1 \ldots w_n \in \Sigma^*$ represents the other way round, it encodes $\text{TeSSLa}_{\text{bool}}$ streams as a synchronized DFST word w. Let $\text{Val} = \{\bot, \mathrm{tt}, \mathrm{ff}, <', \bot', \mathrm{tt}', \mathrm{ff}'\}$ be the set of possible values a stream can have, where the primed values represent an ending of the

stream after the current value. Then β_Σ encodes a set of streams as a word over the alphabet $\Sigma = \{z_1, \ldots, z_k\} \to \text{Val}$, where $z_1, \ldots, z_k$ are variables which relate to the corresponding streams $s_1, \ldots, s_k$. Let $T = \{t_0, t_1, t_2, \ldots, t_n\} \setminus \{\infty\}$ with $t_0 = 0$ be the set of all timestamps present in the streams including 0 excluding ∞ with $t_i < t_{i+1}$. Then w_i is defined as follows:

$$w_i(s) = \begin{cases} <' & \text{if } s = vt_i \\ s(t_i)' & \text{if } s = vt_id \\ s(t_i) & \text{if } \exists t \in T : t > t_i \vee s = v\infty \\ \bot & \text{otherwise} \end{cases}$$

Informally, $w_i(s) = <'$ if s has finite, exclusive progress of t_i, hence ends with a timestamp that is not ∞. $w_i(s) = s(t_i)'$ if the progress ends with that timestamp inclusively. In this case, we will add a prime symbol to the value to mark the streams end. Furthermore, $w_i(s) = s(t_i)$ if s has not ended already at t_i, so is either a data value or $\bot$ but the progress still goes on and $w_i = \bot$ if the progress of the stream already ended (in this case, the exact symbol does not matter since the end is already encoded in a primed symbol). This $\bot$ after the end of the stream is necessary because other streams could still have progress.

Hence β_Σ encodes the streams as a word by creating symbols which are functions from stream names to values. For every timestamp where at least one of the streams has an event, or where its progress ends, a symbol exists in the resulting word.

Note that since the boolean transducers produce one output symbol per input symbol one could reattach the timestamps of the input streams to the output streams to preserve the exact timestamps, too.

The following Theorem states the relation between $\text{TeSSLa}_{\text{bool}}$ and DFSTs.

Theorem 5.9 (*Relation Between TeSSLa$_{bool}$ and DFSTs*)

For a DFST $R = (\Sigma, \Gamma, Q, q_0, \delta)$ there is a TeSSLa$_{bool}$ formula φ_R and for a TeSSLa$_{bool}$ formula φ there is a DFST $R_\varphi = (\Sigma, \Gamma, Q, q_0, \delta)$ s.t.

$$\alpha_\Gamma \circ [\![R]\!] = [\![\varphi_R]\!] \circ \alpha_\Sigma \quad \textit{and} \quad \beta_\Gamma \circ [\![\varphi]\!] = [\![R_\varphi]\!] \circ \beta_\Sigma.$$

We will prove the previous theorem constructively for both directions in the next two subsections.

5.2.1 Translating DFST to TeSSLa$_{\text{bool}}$

Given a DFST $R = (\Sigma, \Gamma, Q, q_0, \delta)$ we will now show how to create the corresponding TeSSLa$_{\text{bool}}$ formula φ_R from Theorem 5.9. The main part of this translation is a set of streams x_q which represent for each state $q \in Q$ if the DFST is currently in q, then $x_q = \text{tt}$, or if it is not in q, then $x_q = \text{ff}$. Each x_q is a disjunction of streams representing when a transition is taken that ends in q.

$$x_{q'} := \bigvee_{(q,\sigma,q',\gamma)\in\delta} d_{q,\sigma,q',\gamma}.$$

Each of them is a disjunction of all transitions which have the state q' as next state and if one of these transitions is taken, the corresponding $x_{q'}$ is true which means that the state q' is now active. Because a DFST is deterministic, only one of the streams $d_{q,\sigma,q',\gamma}$ representing transitions can be true at any time.

Before we get to the formal representation of each transition, we add one layer to the streams representing the states first, which is used to initialize them with their starting value. We represent the states $q \in Q \setminus \{q_0\}$ without the start state as streams which are true iff the transducer is in the corresponding state and because these states are not starting states, x_q is merged with false as initial value.

$$a_q := \mathsf{merge}(x_q, \text{ff})$$

On the other hand, the initial state is represented as

$$a_{q_0} := \mathsf{merge}(x_{q_0}, \text{tt})$$

which means that it is true in the beginning.

Every single transition $(q, \sigma, q', \gamma) \in \delta$ on the other hand is transformed by adding two streams, one checking if this transition is active now, $d_{q,\sigma,q',\gamma}$, and one logging the output of this transition if it is active, o_i, as follows:

$$d_{q,\sigma,q',\gamma} := \mathbf{last}(a_q, s_\sigma) \wedge s_\sigma$$

and

$$o_{q,\sigma,q',\gamma} := \mathsf{filter}(d_{q,\sigma,q',\gamma}, \mathsf{const}(\gamma, d_{q,\sigma,q',\gamma})).$$

Each $d_{q,\sigma,q',\gamma}$ is a conjunction of the stream which represents if the input symbol for the transition is in the current symbol of the word and a **last** which checks if the corresponding start state of

the transition was active before. The $o_{q,\sigma,q',\gamma}$ represent a stream of each transitions outputs. Every time the corresponding transition is taken a new event is emitted where the value is set to γ.

As mentioned before, because a DFST is deterministic, only one transition can be taken at any point in time. Hence for every timestamp, only one output stream has an event and to get the stream with all the outputs, we just need to merge the output streams for every single transition as follows:

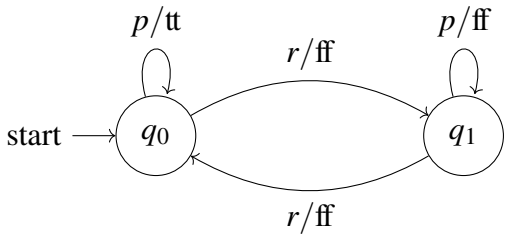
$$output := \mathsf{merge}\{o_{q,\sigma,q',\gamma} \mid (q,\sigma,q',\gamma) \in \delta\}$$

where this merge over all $o_{q,\sigma,q',\gamma}$ is just a shorthand for the multi application of the binary merge. Because never two or more of the $o_{q,\sigma,q',\gamma}$ have an event at the same point in time, the order in which these streams are merged does not matter.

Next, we give an example of how the transformation from a DFST into a TeSSLa$_{\text{bool}}$ formula works.

Example 5.10 (*Transforming a DFST into TeSSLa$_{bool}$*)

Consider the DFST $R = (\{p,r\},\{\text{tt},\text{ff}\},\{q_0,q_1\},q_0,\delta)$ where the transition function δ is given by the following graph:



Using the input streams s_p and s_r for the two input symbols p and r, the corresponding TeSSLa$_{\text{bool}}$ formula is the one given in Figure 5.2.

In the following section we will show the other direction, therefore, how to transform a TeSSLa$_{\text{bool}}$ into a DFST.

5.2.2 Translating TeSSLa$_{\text{bool}}$ to DFST

For the other direction, we translate every equation of the flattened specification of φ into individual DFSTs, which are then composed into one DFST R_φ. For every DFST the input symbols

$$
\begin{aligned}
a_{q_0} &:= \mathsf{merge}(x_{q_0}, \mathrm{tt}) \\
a_{q_1} &:= \mathsf{merge}(x_{q_1}, \mathrm{ff}) \\
x_{q_0} &:= d_{q_0,p,q_0,\mathrm{tt}} \vee d_{q_1,r,q_0,\mathrm{ff}} \\
x_{q_1} &:= d_{q_1,p,q_1,\mathrm{ff}} \vee d_{q_0,r,q_1,\mathrm{ff}} \\
d_{q_0,p,q_0,\mathrm{tt}} &:= \mathbf{last}(a_{q_0}, s_p) \wedge s_p \\
d_{q_0,r,q_1,\mathrm{ff}} &:= \mathbf{last}(a_{q_0}, s_r) \wedge s_r \\
d_{q_1,p,q_1,\mathrm{ff}} &:= \mathbf{last}(a_{q_1}, s_p) \wedge s_p \\
d_{q_1,r,q_0,\mathrm{ff}} &:= \mathbf{last}(a_{q_1}, s_r) \wedge s_r \\
o_{q_0,p,q_0,\mathrm{tt}} &:= \mathsf{filter}(d_{q_0,p,q_0,\mathrm{tt}}, \mathsf{const}(\mathrm{tt})(d_{q_0,p,q_0,\mathrm{tt}})) \\
o_{q_0,r,q_1,\mathrm{ff}} &:= \mathsf{filter}(d_{q_0,r,q_1,\mathrm{ff}}, \mathsf{const}(\mathrm{ff})(d_{q_0,r,q_1,\mathrm{ff}})) \\
o_{q_1,p,q_1,\mathrm{ff}} &:= \mathsf{filter}(d_{q_1,p,q_1,\mathrm{ff}}, \mathsf{const}(\mathrm{ff})(d_{q_1,p,q_1,\mathrm{ff}})) \\
o_{q_1,r,q_0,\mathrm{ff}} &:= \mathsf{filter}(d_{q_1,r,q_0,\mathrm{ff}}, \mathsf{const}(\mathrm{ff})(d_{q_1,r,q_0,\mathrm{ff}})) \\
\mathit{output} &:= \mathsf{merge}(o_{q_0,p,q_0,\mathrm{tt}}, o_{q_0,r,q_1,\mathrm{ff}}, o_{q_1,p,q_1,\mathrm{ff}}, o_{q_1,r,q_0,\mathrm{ff}})
\end{aligned}
$$

Figure 5.2: The TeSSLa$_{\text{bool}}$ formula created from the DFST R given in Example 5.10. The streams s_p and s_r are the input streams resulting from the input symbols of R.

are functions from the names of the input streams to Val and the output symbols are functions from the name of the equation to Val. As discussed in the previous section, for this finite data domain we only need to consider finitely many different internal states for every equation. The transition function realizes the state changes and the current output based on the current state.

The function toDFST builds the transducers for a single equation of a flattened TeSSLa$_{\text{bool}}$ formula φ as explained next. The first types of equations we look at are equations of the form $z := \mathbf{nil}$.

$$\mathsf{toDFST}(z := \mathbf{nil}) = (\emptyset, \{z\} \to \mathrm{Val}, \{s\}, s, \delta)$$

with $\delta(s, \emptyset) = (s, \{z \mapsto \bot\})$. Because **nil** is an operator without any parameters, the DFST has no input symbols, only one state and can only output $\bot$ which represents the fact that **nil** produces the stream without any events.

Compared to **nil**, the operator **unit** is quite similar with the difference that an event is output as first action. So while the DFST for **unit** still has no input symbols, it has two states where the only transition from the first one goes to the second one and outputs an event with value true. From the

second state only a loop exists which always outputs $\bot$. This results in the following DFST:

$$\mathsf{toDFST}(z := \textbf{unit}) = (\emptyset, \{z\} \to \mathrm{Val}, \{s_0, s_1\}, s_0, \delta)$$

with $\delta(s_0, \emptyset) = (s_1, \{z \mapsto \mathrm{tt}\})$ and $\delta(s_1, \emptyset) = (s_1, \{z \mapsto \bot\})$.

While **nil** and **unit** are operators without parameters, they result in constant streams. Compared to them, **lift** is more complex because it has to react to the input of its parameters.

The DFST for **lift** consists of two states: s where the transducer loops for every input and applies f to the inputs to compute the output and the sink s_e which is reached when one of the input streams ends. Formally, an equation $z := \textbf{lift}(f)(a^0, \ldots, a^n)$ can be translated to the following DFST:

$$\mathsf{toDFST}(z := \textbf{lift}(f)(a^0, \ldots, a^n)) = (\{a^{i \leq n}\} \to \mathrm{Val}, \{z\} \to \mathrm{Val}, \{s, s_e\}, s, \delta)$$

with

$$\delta(s,h) = \begin{cases} (s, \{z \mapsto g(b^0, \ldots, b^n)\}) & \text{if } \forall i : h(a^i) \in \{\mathrm{tt}, \mathrm{ff}, \bot\} \\ (s_e, \{z \mapsto <'\}) & \text{if } \exists i : h(a_i) = <' \\ (s_e, \{z \mapsto g(b^0, \ldots, b^n)'\}) & \text{otherwise,} \end{cases}$$

where

$$b^i = \begin{cases} \mathrm{tt} & \text{if } h(a^i) \in \{\mathrm{tt}, \mathrm{tt}'\} \\ \mathrm{ff} & \text{if } h(a^i) \in \{\mathrm{ff}, \mathrm{ff}'\} \\ \bot & \text{otherwise} \end{cases}$$

and

$$g(b^0, \ldots, b^n) = \begin{cases} \bot & \text{if } \forall i : b^i = \bot \\ f(b^0, \ldots, b^n) & \text{otherwise} \end{cases}$$

While looping in s it is the function f taking care of the output as g is essentially only applying f and the b^i are just for converting stream-ending input values into normal values.

The last type of equations is $z := \textbf{last}(a,b)$. As **last** has to remember the last value and we have only the boolean data domain, two states are needed to solve this problem. Furthermore there are some special cases for initialization (state s_0) and endings of streams (states s_w, s_v and s_e) in the **last** operator. A DFST for an equation of the form $z := \textbf{last}(a,b)$ can be build as follows:

$$\mathsf{toDFST}(z := \textbf{last}(a,b)) = (\{a,b\} \to \mathrm{Val}, \{z\} \to \mathrm{Val}, \{s_0, s_{\mathrm{tt}}, s_{\mathrm{ff}}, s_w, s_v, s_e\}, s_0, \delta)$$

$$
\begin{aligned}
\delta(s_0, a_\bot \cup b_y) &= (s_0, z_\bot) \\
\delta(s_0, a_{x\in\mathrm{tf}} \cup b_y) &= (s_x, z_\bot) \\
\delta(s_0, a'_x \cup b'_y) &= (s_e, z'_\bot) \\
\delta(s_0, a_x \cup b'_y) &= (s_w, z_\bot) \\
\delta(s_0, a'_x \cup b_y) &= (s_v, z_\bot) \\
\delta(s_{d\in\mathrm{tf}}, a_\bot \cup b_\bot) &= (s_d, z_\bot) \\
\delta(s_{d\in\mathrm{tf}}, a_{x\in\mathrm{tf}} \cup b_\bot) &= (s_x, z_\bot) \\
\delta(s_{d\in\mathrm{tf}}, a_\bot \cup b_{y\in\mathrm{tf}}) &= (s_d, z_d) \\
\delta(s_{d\in\mathrm{tf}}, a_{x\in\mathrm{tf}} \cup b_{y\in\mathrm{tf}}) &= (s_x, z_d) \\
\delta(s_{d\in\mathrm{tf}}, a'_x \cup b_y) &= (s_v, z_\bot) \\
\delta(s_{d\in\mathrm{tf}}, a^?_x \cup b'_{y\in\{<,\bot\}}) &= (s_e, z'_y) \\
\delta(s_{d\in\mathrm{tf}}, a^?_x \cup b'_{y\in\mathrm{tf}}) &= (s_e, z'_d)
\end{aligned}
\qquad
\begin{aligned}
\delta(s_w, a_\bot \cup b^?_y) &= (s_w, z_\bot) \\
\delta(s_w, a_{x\in\mathrm{tf}} \cup b^?_y) &= (s_e, z'_\bot) \\
\delta(s_w, a'_x \cup b^?_y) &= (s_e, z'_\bot) \\
\delta(s_v, a^?_x \cup b_\bot) &= (s_v, z_\bot) \\
\delta(s_v, a^?_x \cup b'_\bot) &= (s_e, z'_\bot) \\
\delta(s_v, a^?_x \cup b^?_{y\in\{<,\mathrm{tt},\mathrm{ff}\}}) &= (s_e, z'_<) \\
\delta(s_e, a^?_x \cup b^?_y) &= (s_e, z_\bot)
\end{aligned}
$$

Figure 5.3: The transition function of the DFST for a $\text{TeSSLa}_{\text{bool}}$ formula $z := \mathbf{last}(a,b)$. It is depicted here in two parts. The left hand side represents the transitions from the initial state as well as the transitions from the two states which remember the last value. The right hand side represents the handling at and after the ending of progress.

where we use the following abbreviations for the definition of δ:

- $\mathrm{tf} = \{\mathrm{tt}, \mathrm{ff}\}$,
- $a_x = \{a \mapsto x\}$ for $x \in \{\bot, \mathrm{tt}, \mathrm{ff}\}$,
- $a'_x = \{a \mapsto x'\}$ for $x \in \{<, \bot, \mathrm{tt}, \mathrm{ff}\}$ and
- $a^?_x = \{a \mapsto x\}$ for $x \in \mathrm{Val}$.

Then the definition of δ is given in Figure 5.3.

$\mathsf{toDFST}(z := \mathbf{last}(a,b))$ has essentially three states: s_0 is taking care of the initialization and as long as no event occurred on a the DFST stays in s_0. After an event occurred on a the DFST moves to s_{tt} or s_{ff} depending on the value on the event and after that the transducer always moves between these two states when an event occurs on a depending on its value. If the transducer is in one of the two last mentioned states it outputs the index of the state when an event occurs on b. The three additional states are there to handle different cases when the stream ends because different amount of progress can be output depending on the order in which the streams end.

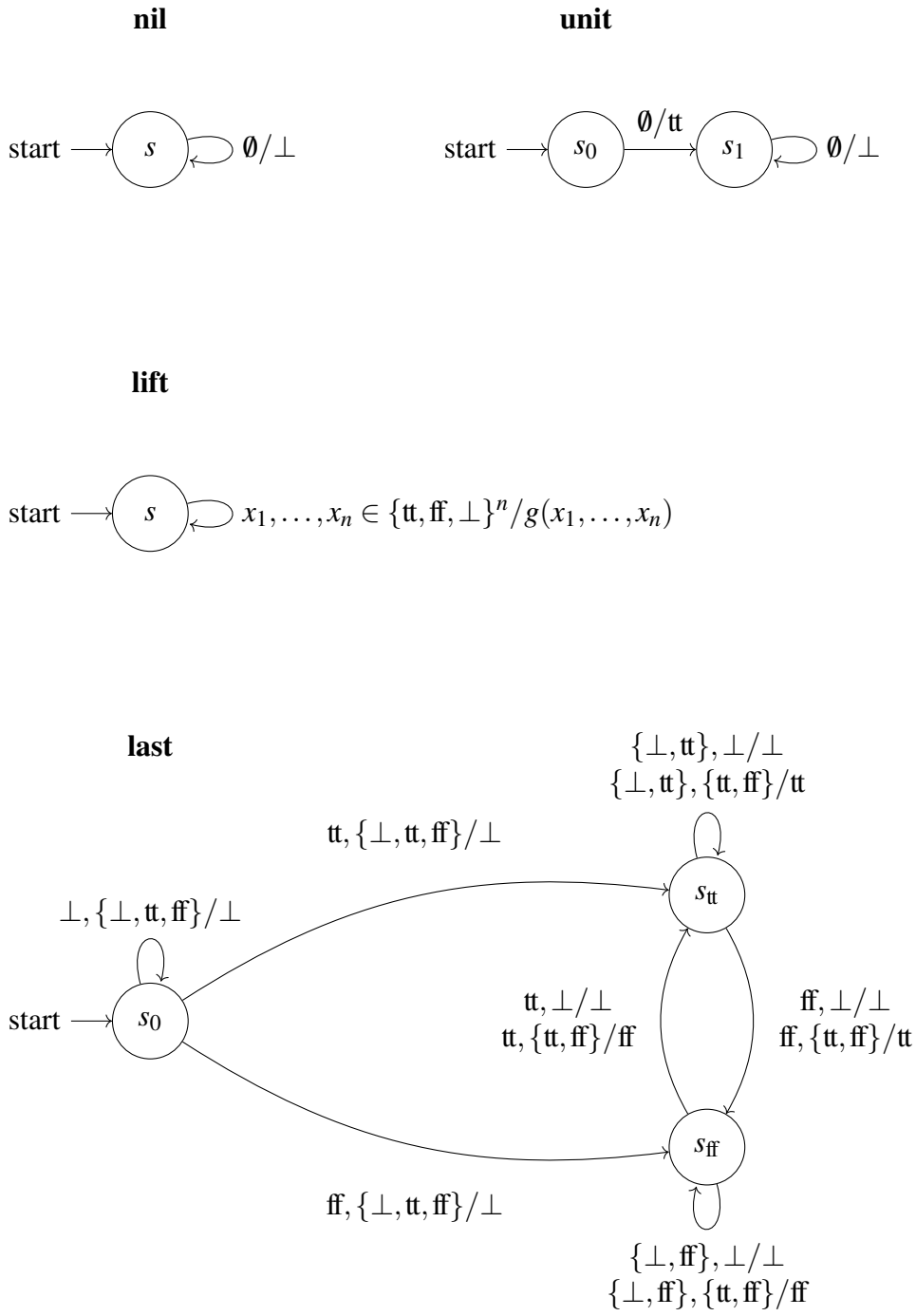


Figure 5.4: Shows the transducers for every TeSSLa operator in the TeSSLa$_{\text{bool}}$ fragment, which are **nil**, **unit**, **lift** and **last**.

A visual representation of the single DFSTs for every operator can be seen in Figure 5.4. The special cases for the endings of the stream have been left out for giving a better overview on how the operators and DFSTs work. The general structure of the transducers also shows the functionality of the operators in TeSSLa.

Until now, we have a set of transducers for every equation in the flattened version of the given $\text{TeSSLa}_{\text{bool}}$ specification. To get the final transducer representing the whole formula, we have to compose the individual DFSTs by combining them at the correspondingly named inputs and outputs. How this works is explained in the following.

At first, we always compose two existing transducers into a new one until only one transducer is left. This is done via a standard parallel composition algorithm that composes the transducers such that they are executed in parallel by a new transducer. Therefore, a state in the new transducer is a tuple of states from the original transducers, one from each, and the transition function has an entry every time both transducers have an entry in the transition function with the same input symbol. The output of each transition is the union of the outputs from the original transitions. Formally, this is done in the following way:

Let $R = (I \to \text{Val}, O \to \text{Val}, Q, q_0, \delta)$ and $R' = (I' \to \text{Val}, O' \to \text{Val}, Q', q'_0, \delta')$ be two DFSTs. The parallel composition of R and R' is then $R'' = (I \cup I' \to \text{Val}, O \cup O' \to \text{Val}, Q \times Q', (q_0, q'_0), \delta'')$ with

$$\begin{aligned}\delta''((s_1,s_2),g'') = ((s'_1,s'_2),h'') \Longleftrightarrow & \delta(s_1,g) = (s'_1,h) \wedge \delta'(s_2,g') = (s'_2,h') \wedge \\ & g'' = g \cup g' \wedge \forall \sigma \in I \cap I' : g(\sigma) = g'(\sigma) \wedge h'' = h \cup h'\end{aligned}$$

In the end, we have one transducer $R_A = (I_A \to \text{Val}, O_A \to \text{Val}, Q_A, q_{0A}, \delta_A)$ which represents all equations.

In the resulting transducer R_A, it is still possible that it contains transitions with the same in- and output values for certain propositions which represents dependencies between the original equations, like a cycle in the specification. To fix this problem, we now build the closure of this transducer which roughly resembles substituting the variables and computing the fixed-point of the equations, which results in $R_\varphi = (I_A \backslash O_A \to \text{Val}, O_A \to \text{Val}, Q_A, q_{0A}, \delta_\varphi)$, where

$$\delta_\varphi(s,g) = (s',h) \Longleftrightarrow \delta_A(s,g') = (s',h) \wedge g = g'|_{I_A \backslash O_A} \wedge (\forall a \in I_A \cap O_A : g'(a) = h(a))$$

for $g|_I := g \cap (I \times \text{Val})$.

The following example shows how a given TeSSLa$_{\text{bool}}$ specification is transformed into a DFST.

Example 5.11 (*Transforming a TeSSLa$_{bool}$ formula into a DFST*)

Consider the following TeSSLa$_{\text{bool}}$ specification:

$$x := \mathsf{mergeAnd}(\mathbf{last}(x,b),b)$$

where $\mathsf{mergeAnd}(a,b) = \mathbf{lift}(mergeAnd)(a,b)$ is a lifted function with $mergeAnd : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ defined as follows:

$$\begin{aligned} mergeAnd(\bot,b) &= b \\ mergeAnd(a,\bot) &= a \\ mergeAnd(a,\mathrm{ff}) &= \mathrm{ff} \\ mergeAnd(\mathrm{ff},b) &= \mathrm{ff} \\ mergeAnd(\mathrm{tt},\mathrm{tt}) &= \mathrm{tt} \end{aligned}$$

At first, we create a flattened specification:

$$\begin{aligned} x &:= \mathsf{mergeAnd}(a,b) \\ a &:= \mathbf{last}(x,b) \end{aligned}$$

We can now create the two single transducers, one for **lift** which is $R_x = (\{a,b\} \to \text{Val}, \{x\} \to \text{Val}, \{s\}, s, \delta_x)$ and one for **last** which is $R_a = (\{x,b\} \to \text{Val}, \{a\} \to \text{Val}, \{s_1,s_2,s_3\}, s_1, \delta_a)$. The transition functions are as depicted in Figure 5.4. After doing so, we can build the parallel composition which results in $R' = (\{a,b,x\} \to \text{Val}, \{a,x\} \to \text{Val}, \{(s,s_1),(s,s_2),(s,s_3)\}, (s,s_1), \delta')$ where δ' is given through the graph in Figure 5.5.

Because the specification was recursive, the DFST R' contains transitions with the same in- and output variables. We now have to remove all the transitions where these do not have the same value and just leave those as output if they have the same value. This results in the transducer $R_\varphi = (\{b\} \to \text{Val}, \{a,x\} \to \text{Val}, \{(s,s_1),(s,s_2),(s,s_3)\}, (s,s_1), \delta_\varphi)$ where δ_φ is given through the graph in Figure 5.6. As one can see, the resulting DFST R_φ is now working exactly how the specification would.

In the next section, we will provide results on equivalence and finite memory for the boolean TeSSLa fragment.

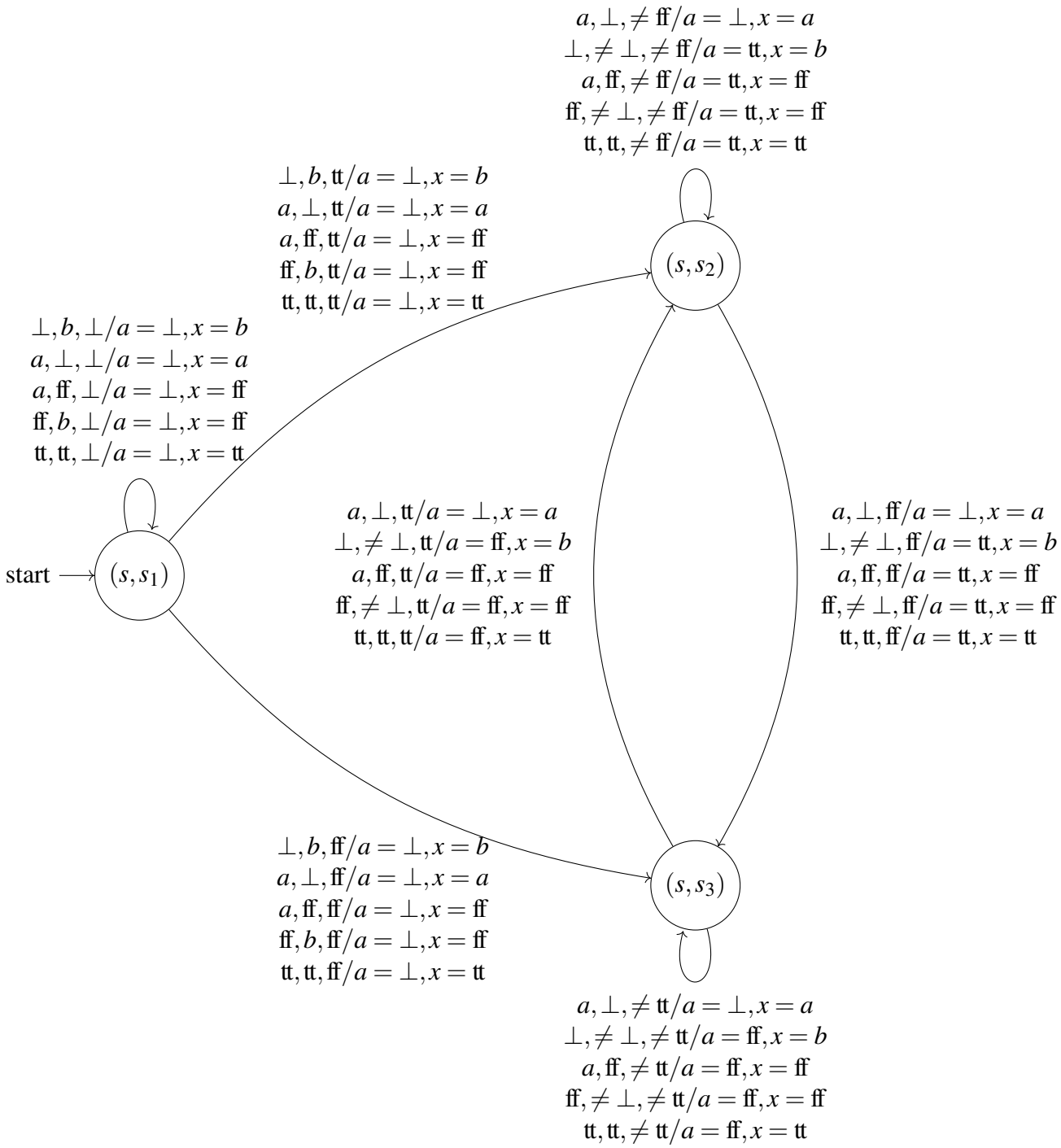


Figure 5.5: The transducer resulting from the parallel composition of the transducers for the equations $x := \mathsf{mergeAnd}(a,b)$ and $a := \mathbf{last}(x,b)$. If an input is denoted as $\neq x$, then the transition can be taken if the other inputs fit and this one has a value which is not x.

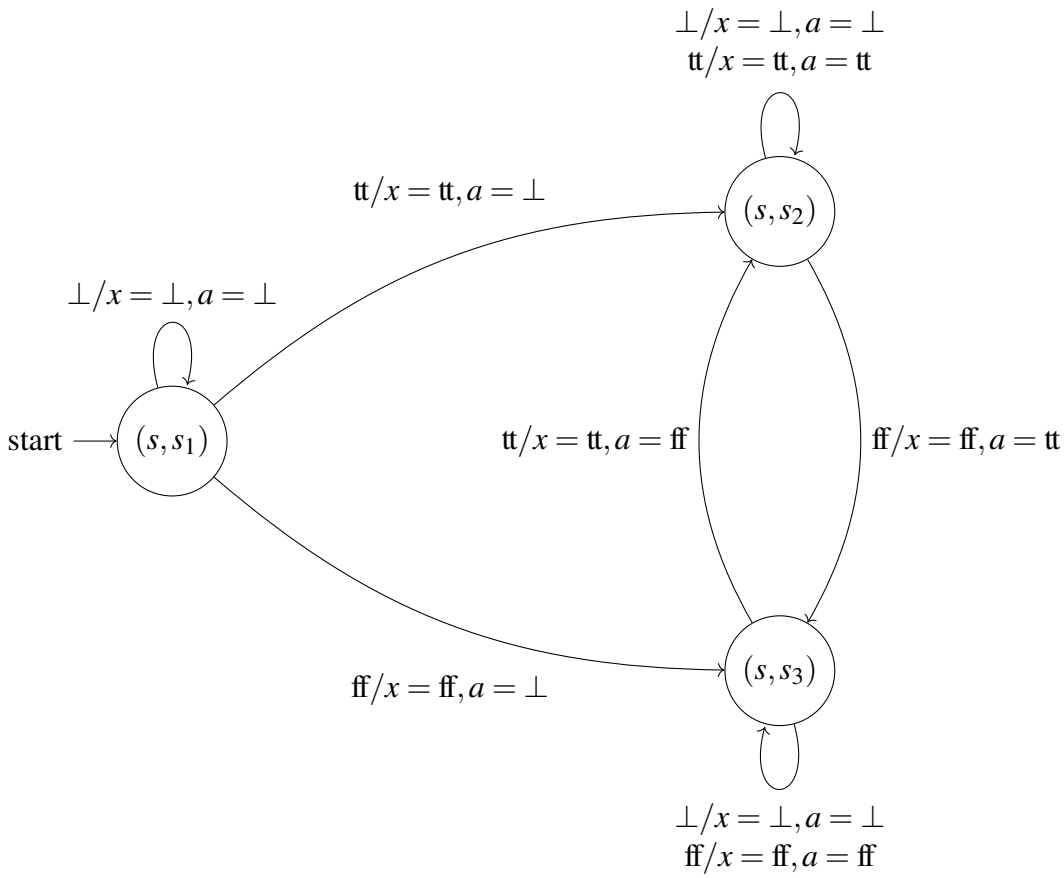


Figure 5.6: The transducer resulting from the one in Figure 5.5 after building the closure of it.

5.2.3 Results for TeSSLa$_{\text{bool}}$

By using the previous result for the relation of TeSSLa$_{\text{bool}}$ and DFSTs, we can now give results on equivalence of TeSSLa$_{\text{bool}}$ formulas. The proof is mainly based on the fact that equivalence for deterministic automata is in PTIME and that the constructed DFSTs can be represented as those. Note that we only show that equivalence of TeSSLa$_{\text{bool}}$ is in EXPTIME, we make no statement regarding completeness.

Theorem 5.12 (*Equivalence for TeSSLa$_{bool}$*)

Equivalence of TeSSLa$_{bool}$ formulas is in EXPTIME.

Proof. Above, we have shown how TeSSLa$_{\text{bool}}$ relates to DFSTs. To show the complexity of the equivalence problem for TeSSLa$_{\text{bool}}$ we use this relation, thus, we show the complexity of the equivalence problem for DFSTs.

We do this by reducing the equivalence problem for DFSTs to the equivalence problem for deterministic automata. We can encode a DFST $R = (\Sigma, \Gamma, Q, q_0, \delta)$ as a DFA $A = (\Sigma', Q', q_0', F', \delta')$ as follows:

- $\Sigma' = \Sigma \times \Gamma$,
- $Q' = Q \cup \{f\}$,
- $q_0' = q_0$,
- $F' = Q$
- $\delta'(q,(\sigma,\gamma)) = \begin{cases} q' & \text{if } \delta(q,\sigma) = (q',\gamma) \\ f & \text{otherwise} \end{cases}$

By using this construction, we can now transform two DFSTs into the corresponding DFAs and check equivalence there. And because equivalence of DFAs is in PTIME [HMU06], so it is of DFSTs.

In the end we have to take a look at the transformation from TeSSLa$_{\text{bool}}$ to DFSTs. This transformation is exponential since in the worst case, the formula consists of nested **last** operators. As shown before, the DFST for a **last** has always two states for remembering the last value of its subexpressions which results in an exponential blow up when using the composition algorithm on nested **last** operators. Therefore, it follows that equivalence of TeSSLa$_{\text{bool}}$ is in EXPTIME. □

Furthermore, we can make a statement about the memory usage for evaluating a $\text{TeSSLa}_{\text{bool}}$ formula. Recall that E_{TeSSLa} is the evaluation strategy which evaluates a TeSSLa formula compositionally by evaluating every operator on its own and forwarding the output to the operators following according to the dependency graph.

Theorem 5.13 (*Finite Memory and TeSSLa_{bool}*)

TeSSLa_{bool} under E_{TeSSLa} is finite memory.

Proof. The only operator that needs to store values in $\text{TeSSLa}_{\text{bool}}$ under E_{TeSSLa} is the **last**, which stores exactly one value at any time. This is the value it has to output when a trigger event occurs. Because only boolean values exist in this fragment, the size of each value is bounded. Hence, the maximum number of data any $\text{TeSSLa}_{\text{bool}}$ formula has to store is finite because there can be only finitely many **last**s. □

Because $\text{TeSSLa}_{\text{bool}}$ specifications are always finite memory, the two properties FMP (the question, if a given formula is finite memory under a certain evaluation strategy) and RFM (the question, if for a given formula an evaluation strategy exists under which the formula is finite memory) are always fulfilled, therefore, it is not necessary to make further statements about those.

5.3 Pushdown Fragment

The pushdown fragment of TeSSLa, which adds stacks to $\text{TeSSLa}_{\text{bool}}$, is interesting because it covers the context free languages and has interesting properties which are even relevant in practice, because it adds a possibility to have an infinite data structure while preserving good complexity results for important properties. It somehow represents the middle ground of fragments of TeSSLa that are finite memory and those which are not finite memory and have undecidable properties.

The fragment $\text{TeSSLa}_{\text{stack}}$ extends $\text{TeSSLa}_{\text{bool}}$ with streams of stacks, where a stack can contain an arbitrary amount of data, thus, can potentially grow unbounded. A stack is given over a certain data domain, in this case the boolean data domain $\mathbb{B}$. Informally, the way the stack works in $\text{TeSSLa}_{\text{stack}}$ resembles very much the way a stack is used in pushdown transducers, which we also compare $\text{TeSSLa}_{\text{stack}}$ later to. Such a transducer uses exactly one stack only for internal calculations and the same does $\text{TeSSLa}_{\text{stack}}$. Therefore, stacks are used as data values of events, but only one single

equation is allowed to represent a stream with stacks as values, which also means no input stream is allowed to have stacks.

A stack $\zeta \in \mathbb{B}^*$ of the boolean domain $\mathbb{B}$ is a sequence of data values. In TeSSLa$_{\text{stack}}$ there are four functions to operate on stacks. While *isEmpty* and *top* only return boolean values and can be used in any equation, *pop* and *push* return stacks and are only allowed in one equation in the formula. It is important to note that a TeSSLa$_{\text{stack}}$-formula does not take a stream of stacks as input, all streams with stacks are only internal. Thus it can solely be used to store previous data values or decisions. This restriction is later necessary to keep the power of this fragment down to the one of a pushdown transducer with one stack. Otherwise, its expressiveness would get to the one of a TM, because a pushdown automaton with two stacks is as expressive as a TM.

Formally, TeSSLa$_{\text{stack}}$ restricts TeSSLa to boolean streams, streams of stacks over $\mathbb{B}$ and the operators **last** and **lift**.

Definition 5.14 (*TeSSLa$_{stack}$*)

A TeSSLa formula φ is called a TeSSLa$_{stack}$ formula if $\varphi : \mathcal{S}_{\mathbb{B}} \times \ldots \times S_{\mathbb{B}} \to S_{\mathbb{B}} \times \ldots \times \mathcal{S}_{\mathbb{B}}$ and the syntax of every equation e is restricted as follows, where f is a function $f : \mathbb{B}^n_\perp \to \mathbb{B}_\perp$ or $f \in \{pop, push, isEmpty, top\}$:

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

Additionally, there is only one equation allowed which contains pop and / or push. The semantics for every single operator stays the same as in TeSSLa and the functions $isEmpty : \mathbb{B}^* \to \mathbb{B}, top : \mathbb{B}^* \to \mathbb{B}, pop : \mathbb{B}^* \to \mathbb{B}^*$ and $push : \mathbb{B}^* \times \mathbb{B} \to \mathbb{B}^*$ are defined as follows:

$$\begin{aligned} isEmpty(\langle b \rangle \& \zeta) &= ff \\ isEmpty(\langle\rangle) &= tt \\ pop(\langle b \rangle \& \zeta) &= \zeta \\ pop(\langle\rangle) &= \langle\rangle \\ top(\langle b \rangle \& \zeta) &= b \\ push(\zeta, b) &= \langle b \rangle \& \zeta \end{aligned}$$

Note at this point, that because of the definition above, flattened TeSSLa$_{\text{stack}}$ specifications may not be TeSSLa$_{\text{stack}}$ specifications anymore because they may contain multiple expressions containing

operations that return a stack. Nevertheless, we do not define a new notion of flatness, but still stay with the one for TeSSLa and use it. For the transformations into pushdown transducers later, it has no effect if the flattened specifications are not $\text{TeSSLa}_{\text{stack}}$ specifications in general.

As one can easily see, because of the potentially infinitely growing stack, $\text{TeSSLa}_{\text{stack}}$ is not finite memory and the following Theorem states this.

Theorem 5.15 (*Finite Memory and $\text{TeSSLa}_{\text{stack}}$*)

$\text{TeSSLa}_{\text{stack}}$ under E_{TeSSLa} is not finite memory.

Proof. Since the *push* function can be used on the stack stream in a cyclic way, an arbitrary amount of values can be pushed on the stack, as in the following formula, where s is an input stream:

$$x := \textbf{lift}(push)(\textbf{last}(x,s),s)$$

This leads to an unbounded amount of data that has to be remembered, because for every event on s, one element is pushed on the stack. Thus, $\text{TeSSLa}_{\text{stack}}$ formulas exist which are not finite memory. □

Still, it can be possible, even if a $\text{TeSSLa}_{\text{stack}}$ formula is not finite memory under E_{TeSSLa}, that we can rewrite it such that the new formula has the same semantics and is finite memory. The next theorem tells us that it is at least decidable for a given $\text{TeSSLa}_{\text{stack}}$ formula if it can be rewritten such that it is finite memory. Additionally, this theorem also states a result on equivalence of $\text{TeSSLa}_{\text{stack}}$ formulas.

The proof shows how to convert a $\text{TeSSLa}_{\text{stack}}$ formula into a DPT and vice versa. From this translation, we can apply results for DPTs to $\text{TeSSLa}_{\text{stack}}$ and by doing so, show the previous statement. For the first direction, to transform a DPT into a $\text{TeSSLa}_{\text{stack}}$ formula, we use the same translation scheme as for $\text{TeSSLa}_{\text{bool}}$ and DFSTs but need to change two parts: first, add a check on the first element of the stack to every equation representing if a transition is active and second, to add an equation representing the stack, which pops elements from the stack or pushes elements onto the stack depending on which transition is active. As the transducer is deterministic, only one transition can be active at any time. For the other direction, to build a DPT for a given $\text{TeSSLa}_{\text{stack}}$ formula, we add three types of transducers to the translation scheme for translating $\text{TeSSLa}_{\text{bool}}$ formulas into DFSTs: If we have to convert a lifted *isEmpty* or *top*, we create a transducer with a

corresponding check on emptiness or the top element of the stack and than add this element back, as the transducer also automatically removes the top element, but the function *top* does not. If we have to translate a lifted *pop*, we add a transducer always removing the top element of the stack when a transition is taken. If we have to translate a lifted *push*, we add a transducer where every transition adds the corresponding element to the stack. All those transducers only have two states, as the general transducer for a **lift** has. The composition algorithm then works in the same way as before, just additionally taking care of the stack.

Theorem 5.16 (*Equivalence and RFM for TeSSLa$_{stack}$*)

The following two statements about TeSSLa$_{stack}$ hold:

1. *Equivalence of TeSSLa$_{stack}$ formulas is decidable.*
2. *RFM for TeSSLa$_{stack}$ under E_{TeSSLa} is decidable.*

Proof. We will show both statements by transforming arbitrary TeSSLa$_{\text{stack}}$ formulas into DPTs and vice versa.

To transform a DPT into a TeSSLa$_{\text{stack}}$-formula, we use the same transformation as for the DFST with the following adjustments:

For every transition $\delta(q,\sigma,\lambda) = (q',(\lambda_1,\ldots,\lambda_n),\gamma)$ which maps a state, an input symbol and the top element of the stack to a new state, a sequence of elements to be pushed on the stack and an output symbol, we add a check to the equation used for DFSTs, if the top element of the stack fits, therefore, is λ:

$$d_{q,\sigma,\lambda,q',\Lambda,\gamma} := \mathbf{last}(a_q, s_\sigma) \wedge s_\sigma \wedge \mathsf{top}(\zeta) = \lambda$$

We replace $\mathsf{top}(\zeta) = \lambda$ with $\mathsf{isEmpty}(\zeta)$ if the check is on emptiness of the stack, hence $\lambda = \#$. Thereby, ζ is the stream of stacks, which is defined as a nested if-then-else clause. It checks for every $d_{q,\sigma,\lambda,q',\Lambda,\gamma}$ if it is true, thus, if the transition is active and then pops an element from the stack with $\mathsf{pop}(\mathbf{last}(\zeta, d_{q,\sigma,\lambda,q',\Lambda,\gamma}))$ if Λ does not contain any elements or pops and pushes elements with $\mathsf{push}(\ldots\mathsf{push}(\mathsf{pop}(\mathbf{last}(\zeta, d_{q,\sigma,\lambda,q',\Lambda,\gamma})),\lambda_n)\ldots,\lambda_1)$ if $\Lambda = \lambda_1,\ldots,\lambda_n$, therefore depending on if the transition pushes something or only pops (reads) an element, ζ is defined accordingly. Thereby, $\mathsf{pop}, \mathsf{push}, \mathsf{isEmpty}$ and top are shortcuts for $\mathbf{lift}(x)$ with $x \in \{pop, push, isEmpty, top\}$.

The other direction, to transform a TeSSLa$_{\text{stack}}$ formula into a DPT, is done using the same algorithm as used before, but now DPTs are used instead of DFSTs. Correspondingly, the four functions isEmpty, top, pop and push use a stack by checking for emptiness, looking at the top

element or changing the stack accordingly while every other function or operator just copies the existing stack and does neither change nor access it. Accordingly, checks and additions to the stack are added to the transitions. The composition then aligns the stack usage perfectly. Because $\text{TeSSLa}_{\text{stack}}$ is deterministic, this results in a DPT.

Then 1. follows from the fact that equivalence for DPTs is decidable, which has been shown in [Sén99]. Because $\text{TeSSLa}_{\text{stack}}$ can be transformed into a DPT, equivalence is also decidable for this TeSSLa fragment.

For 2., [Ste67] showed that the question if the language accepted by a DPT is regular is decidable. Then [Ser99] showed that the question if a semantically equivalent TM which needs only a bounded number of memory cells exists for a given DPT is equivalent to the question if the language accepted by the DPT is regular. Therefore, RFM for $\text{TeSSLa}_{\text{stack}}$ is decidable as well. □

Lastly, we show that one can decide for a given $\text{TeSSLa}_{\text{stack}}$ specification if it is finite memory under E_{TeSSLa} without rewriting it, even if the specification is using a stack. Consider the following example.

Example 5.17 (*Finite Memory $\text{TeSSLa}_{\text{stack}}$ Formula*)

Consider the following $\text{TeSSLa}_{\text{stack}}$ formula:

$$
\begin{aligned}
\mathit{one} &:= \mathsf{merge}(\mathsf{const}(s, \mathrm{tt}), \mathrm{ff}) \\
\mathit{two} &:= \mathsf{merge}(\mathbf{last}(\mathit{one}, s), \mathrm{ff}) \\
x &:= \mathsf{filter}(\mathsf{not}(\mathit{two}), \mathsf{push}(\mathbf{last}(x, s), s))
\end{aligned}
$$

This formula counts the number of values on the stack with the streams *one* and *two* and stops pushing values on the stack when two values are already stored there. Hence it never remembers more than two values and is finite memory.

Therefore, even if $\text{TeSSLa}_{\text{stack}}$ is not finite memory under E_{TeSSLa}, some $\text{TeSSLa}_{\text{stack}}$ formulas still are, as shown above, and the fragment has the nice property that, for a given $\text{TeSSLa}_{\text{stack}}$ formula, it is efficiently decidable if the formula is finite memory under E_{TeSSLa} or not. This is stated in the following theorem.

The proof is based on the DPT created for a $\text{TeSSLa}_{\text{stack}}$ specification as shown in Theorem 5.16. The DPT resembles very much the control flow graph of how E_{TeSSLa} would evaluate a specification, because it is also build compositionally from a single DPT for each operator in the given

specification. Thus, it also resembles the usage of the stack. Therefore, to check if a given specification is finite memory, we need to find out if the stack can reach an unbounded height for a given DPT created from a $\text{TeSSLa}_{\text{stack}}$ specification. This can be done by creating the non-deterministic finite automaton (NFA) which accepts the language of the stack, therefore, it accepts an input word if the stack can have the single symbols of the word as contents in the run of a given DPT. Afterwards, we only have to check if the language accepted by the NFA is of finite size, because then the stack is bounded in height, otherwise it is not.

Theorem 5.18 (*FMP for $\text{TeSSLa}_{\text{stack}}$*)

FMP for $\text{TeSSLa}_{\text{stack}}$ under E_{TeSSLa} is decidable in EXPTIME.

Proof. For a $\text{TeSSLa}_{\text{stack}}$ formula to be not finite memory, the expression which pushes elements to the stack needs to be used in a cyclic way and push more elements on the stack than it pops from it. Hence, it needs to follow the following recursive pattern:

$$x := \ldots \mathbf{push}(\mathbf{last}(x,s),t)\ldots$$

where s and t can be arbitrary other streams. But even if it follows this pattern, there can still be conditions surrounding it which make the formula finite memory.

To solve this problem, we build the DPT from the given formula. Because it uses the stack in the same way as the formula does, we can now according to [MMMP12] build the NFA which accepts all words that can be on the stack of the DPT for any given input word in PTIME. If the language accepted of the NFA is of finite size, then there is some finite maximal height of the stack in the DPT which means that the given formula is finite memory. Otherwise, the stack can be of unbounded height and the formula is not finite memory. This procedure exactly answers FMP for $\text{TeSSLa}_{\text{stack}}$ under E_{TeSSLa}.

As the DPT is exponential in the size of the $\text{TeSSLa}_{\text{stack}}$ formula and building the NFA is in PTIME, FMP for $\text{TeSSLa}_{\text{stack}}$ is in EXPTIME. □

Also note that, with the technique used in the proof for the previous theorem, we can not only decide if a given specification is finite memory, but also, how much memory it exactly needs on the stack. This can be done by looking at the language of the constructed NFA which accepts all words that can be on the stack and find the longest word in this language. The length of this word is exactly how big the stack must be in the worst case, independent of the input.

5.4 Functional Non-deterministic Fragment

We will now expand the boolean fragment from Section 5.2, TeSSLa$_{\text{bool}}$, and add non-determinism. Classical non-determinism, as known from non-deterministic automata, is not something which is present in TeSSLa$_{\text{bool}}$. For one there is no possibility to make choices but there is also no statement on what is a *good* execution, i.e. no meaning of acceptance exists for TeSSLa in general. But TeSSLa is able to mimic such non-deterministic behaviour, by for example remembering multiple values in a set and choosing on of them in the end or by anticipating the future, which can be done with the **next** operator introduced in TeSSLa^f. In the end, for simulating non-determinism in TeSSLa$_{\text{bool}}$, we need two additional mechanisms:

1. A possibility to represent acceptance and only output the correct output if the transducer would accept the run for the given input and
2. a possibility to represent the non-deterministic choices made.

As we are operating on boolean values, using a set or similar data structure to remember an arbitrary amount of values is not feasible, as we would loose an easy way to get complexity results for the different decision problems. But to be able to solve the two issues, we can add the **next** operator to TeSSLa$_{\text{bool}}$, which enables us to look for future choices of a non-deterministic transducer and in the end know at the beginning of the run already, if there is an accepting run for the given input on the transducer. We will call this new fragment TeSSLa$^f_{\text{bool}}$, for being TeSSLa$_{\text{bool}}$ with **next** or TeSSLa^f being restricted to boolean streams.

Definition 5.19 (*TeSSLa$^f_{bool}$*)

A TeSSLa formula φ is called a TeSSLa$^f_{bool}$ formula if $\varphi : S_{\mathbb{B}} \times \ldots \times S_{\mathbb{B}} \to S_{\mathbb{B}} \times \ldots \times S_{\mathbb{B}}$ and the syntax of every equation e is restricted as follows, where $f : (\mathbb{B} \cup \{\bot, ?\})^n \rightarrowtail \mathbb{B} \cup \{\bot, ?\}$:

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

The semantics for every single operator stays the same as in TeSSLa^f.

We again use similar functions α_Σ and β_Σ as before in Section 5.2, where we encoded TeSSLa streams as words for DFSTs and vice versa using these functions. As we consider NFSTs, therefore non-deterministic DFSTs, in this section, the input words are quite similar. The main difference is the encoding, as TeSSLa^f is able to produce ? values on streams anywhere, instead of just at

the end of the stream's progress. While this changes the style of the output of α_Σ and β_Σ, the functionality of the two functions is very similar to the one in Section 5.2.

We again use a one-hot encoding for the transformation of NFST words into TeSSLa^f streams. For α_Σ, we also make a small, additional change compared to the one for DFSTs. We add ending markers to the resulting streams at the last event, which are denoted by a prime added to the value. This is necessary to find the ending when looking into the future to see how the non-determinism plays out. Formally, the function $\alpha_\Sigma(w) = S$ encodes a functional NFST word $w = w_0 w_1 \dots w_n \in \Sigma^*$ as a corresponding set of TeSSLa$^f_{\text{bool}}$ streams for every $p \in \Sigma$ as a stream $s_p \in S$ as follows:

$$s_p = (0, d_0, \bot)(1, d_1, \bot)\dots(n-1, d_{n-1}, \bot)(n, d'_n, \bot)(\infty, \bot, \bot) \Leftrightarrow \forall i : (d_i \Leftrightarrow w_i = p)$$

Thus again for every proposition a stream exists in S and for every symbol w_i of w an event exists on every stream from S with the value true if $w_i = p$ and false otherwise.

The function $\beta_\Sigma(s_1, \dots, s_k) = w = w_0 w_1 \dots w_n \in \Sigma^*$ changes because it now encodes TeSSLa$^f_{\text{bool}}$ streams as a synchronized NFST word w over the alphabet $\Sigma = \{z_1, \dots, z_k\} \to \text{Val}$, where $z_1, \dots, z_k$ are variables which relate to the corresponding streams $s_1, \dots, s_k$, with $\text{Val} = \{x_y \mid x \in \{\bot, ?, \text{tt}, \text{ff}\} \wedge y \in \{\bot, ?\}\}$ being now adjusted such that it contains ? and every value has either a $\bot$ or a ? added as index additionally. Compared to the β_Σ for DFSTs, it now has to represent in the word that between two events, there can either be $\bot$ or ?. We do this by marking each symbol of the word with the value the stream has after this symbol, such that the automaton gets both informations, the current and the following value, at once. Additionally, we can now remove the primed ending symbols, because it is enough for our purpose to add a ? as index to the last symbol of the word.

Let $T = \{t_0, t_1, t_2, \dots, t_n\} \setminus \{\infty\}$ with $t_0 = 0$ be the set of all timestamps present in the streams including 0 excluding ∞ with $t_i < t_{i+1}$. Then w_i is build as follows:

$$w_i(s) = \begin{cases} d_x & \text{if } s = u(t_i, d, x)v \\ x_x & \text{otherwise, where } s = u(t, d, x)v \wedge t < t_i \wedge \nexists t < t' < t_i : s = u'(t', d', x')v' \end{cases}$$

Because we now encode the information about the values after an event directly at the data value in the word, we only have to consider two cases for $w_i(s)$. First, the case where an event exists at t_i and second the case where no event exists at t_i. Because everywhere can be ? now, the ending is not as special anymore as it was before.

The following theorem states the relationship between $\text{TeSSLa}^{f}_{\text{bool}}$ and functional NFSTs, i.e. that both are equally expressive. $\text{TeSSLa}^{f}_{\text{bool}}$ only relates to functional NFSTs and not to full NFSTs, therefore only to NFSTs which represent a function in regards to their input and output and not a relation, which means that for a given input, the output for every accepting run is the same. With the restriction to boolean streams only, it is not possible to represent multiple different outputs, because the number of branches for the output depends on the length of the word and is thus not representable with a finite number of boolean streams. Therefore, $\text{TeSSLa}^{f}_{\text{bool}}$ only relates to functional NFSTs and not NFSTs in general.

Theorem 5.20 (*Relation Between TeSSLa^{f}_{bool} and NFSTs*)

For a functional NFST $R = (\Sigma, \Gamma, Q, q_0, \delta)$ there is a TeSSLa^{f}_{bool} formula φ_R and for a TeSSLa^{f}_{bool} formula φ there is a functional NFST $R_\varphi = (\Sigma, \Gamma, Q, q_0, \delta)$ s.t.

$$\alpha_\Gamma \circ [\![R]\!] = [\![\varphi_R]\!] \circ \alpha_\Sigma \quad \textit{and} \quad \beta_\Gamma \circ [\![\varphi]\!] = [\![R_\varphi]\!] \circ \beta_\Sigma.$$

The proof to this theorem is given in the following two sections.

5.4.1 Transforming functional NFST to $\text{TeSSLa}^{f}_{\text{bool}}$

In this section we show how an arbitrary functional NFST can be transformed into a semantically equivalent $\text{TeSSLa}^{f}_{\text{bool}}$ formula. We will use a similar scheme as for DFSTs, but we need some adjustments for handling the non-determinism.

Generally, the **next** is used to handle the non-determinism. But because it can only simulate it by allowing us to see into the future, we still need to remember in which states we are at any given point in the transducer, which is done via having streams for combinations of states the transducer can be in at any point in time instead of just one stream for every state. Informally, the idea for translating functional NFSTs into $\text{TeSSLa}^{f}_{\text{bool}}$ is to use **next** recursively to simulate the run of the transducer while ignoring the outputs. After doing so, we know if the word would be accepted or not and can give either an output or a marker to indicate the rejection of the input.

Given an functional NFST $R = (\Sigma, \Gamma, Q, q_0, F, \delta)$, we will now show how to create the corresponding $\text{TeSSLa}^{f}_{\text{bool}}$ formula. Note that, different from the TeSSLa code for a DFST, in this

case multiple a_q can be true at once, because there may be multiple states the transducer is (non-deterministically) in. Other than that, in general, the specification is quite similar to the one for DFSTs and TeSSLa$_{\text{bool}}$:

$$
\begin{aligned}
a_q &:= \mathsf{merge}(x_q \wedge \mathit{isAccepting}_q, \mathrm{ff}) \\
a_{q_0} &:= \mathsf{merge}(x_{q_0} \wedge \mathit{isAccepting}_{q_0}, \mathrm{tt}) \\
x_{q'} &:= \left(\bigvee_{(q,\sigma,q',\gamma)\in\delta} d_{q,\sigma,q',\gamma} \right) \\
d_{q,\sigma,q',\gamma} &:= \mathbf{last}(a_q, s_\sigma) \wedge s_\sigma \\
o_i &:= \mathsf{filter}(d_{q,\sigma,q',\gamma} \wedge \mathit{isAccepting}_{q'}, \mathsf{const}(\mathsf{if}\ \mathit{isAccepting}\ \mathsf{then}\ \gamma'\ \mathsf{else}\ \gamma)(d_{q,\sigma,q',\gamma})) \\
\mathit{output} &:= \mathsf{merge}\{o_i \mid \eta_i \in \delta\}
\end{aligned}
$$

Most of it we already know from Subsection 5.2.1. The only change is the addition of the streams $\mathit{isAccepting}_q$ for every state q and the *isEnd* stream. While the *isEnd* stream always checks if the last event is reached (one stream is enough for this purpose, because the input streams all end at the same timestamp), the $\mathit{isAccepting}_q$ streams state at every point in time if, when we start at state q with the remaining events on the streams, we will accept in the end. This reflects the use of non-determinism by calculating the acceptance a priori, which we must do, to only follow the paths that lead to acceptance and to know which symbol we have to output. Formally, it is defined as follows:

$$
\begin{aligned}
\mathit{isAccepting}_q &:= \mathsf{if}\ \mathit{isAccepting}\ \mathsf{then}\ a_q^f\ \mathsf{else}\ \exists q' : d_{q,\sigma,q',\gamma} \wedge \mathbf{next}(a_{q'}, x_q) \\
a_q^f &:= \mathsf{const}(q \in F)(\mathbf{unit}) \\
\mathit{isAccepting} &:= \forall p : s_p \in \{\bot', \mathrm{tt}', \mathrm{ff}'\} \vee \mathbf{last}(s_p, s_p) \in \{\bot', \mathrm{tt}', \mathrm{ff}'\}
\end{aligned}
$$

More precisely, the **next** is used to recursively calculate the future and make the choice now depending on what is the outcome of the **next**, hence, if we will accept in the future following the path or not. The usage of $\mathit{accepting}_q$ in a_q allows us to only go into the states which lead to acceptance with the given input streams and the usage of $\mathit{accepting}_q$ in the o_i streams allows us to decide on the correct output, hence the one which is outputted when accepting the input. This works because the transducer is functional, which means that, given an input, it has to output the same output word on every accepting path.

5.4.2 Transforming TeSSLa$^{f}_{\text{bool}}$ to NFST

For the other direction, we again use mostly the construction given in Subsection 5.2.2 with some minor changes. At first, we need an additional transducer for the **next** operator in TeSSLa$^{f}_{\text{bool}}$. Thus, we will extend the function toDFST from Subsection 5.2.2 to toNFST by the following entry.

$\mathsf{toNFST}(z := \mathbf{next}(a,b))$ has essentially five states. It stays in its initial state s_0, which is accepting, until an event or a ? occurs on the trigger stream b. If that happens, the transducer makes a non-deterministic choice and moves to s_{tt} while outputting tt, s_{ff} while outputting ff or $s_\bot$ while outputting $\bot$, which means that the transducer guesses if the next event on a will be one with value tt, ff or there will be no event at all, respectively. Additionally, the transducer may go to $s_?$ if an event occurs and output ?. Also, if it sees a ? on b, it may move to any state besides $s_\bot$ and output the same value, which means it guesses that there will be some value corresponding to the state it went to justify the guess of output.

If it is in one of the three states s_{tt}, s_{ff} or $s_?$, the transducer waits for the confirmation of the non-deterministic choice it made earlier, it has an obligation to fulfil, which means that a needs as next event either one with tt or ff in case of the two first mentioned states which matches the output it chose, or a ? in case of $s_?$. If it moves to $s_\bot$, it guesses that neither than event nor a ? will occur any more on a. Then, only s_0 is accepting because if the transducer is there, any obligation is fulfilled and $s_\bot$ is accepting because it only stays there if there is neither an event nor a ? any more on a.

Formally, the transducer for an equation $z := \mathbf{next}(a,b)$ can be build as follows:

$$\mathsf{toNFST}(z := \mathbf{next}(a,b)) = (\{a,b\} \to \text{Val}, \{z\} \to \text{Val}, \{s_0, s_{\text{tt}}, s_{\text{ff}}, s_\bot, s_?\}, \{s_0\}, \{s_0, s_\bot\}, \delta)$$

For the definition of δ we use the abbreviations again, as follows:

- $\text{tf} = \{\text{tt}, \text{ff}\}$ and
- $a^y_x = \{a \mapsto x_y\}$ for $x_y \in \text{Val}$.

Then, the transition relation δ is given in Figure 5.7, where we assume that not specified inputs for δ lead to rejection of the input.

A depiction of the transducer for **next** can be seen in Figure 5.8.

$$\delta(s_0, a_x^y \cup b_\perp^\perp) = (s_0, z_\perp^\perp)$$
$$\delta(s_0, a_x^? \cup b_{v\neq\perp}^w) = (s_0, z_?^w)$$
$$\delta(s_0, a_x^? \cup b_\perp^?) = (s_0, z_\perp^?)$$
$$\delta(s_0, a_x^\perp \cup b_{v\in\mathrm{tf}}^w) = (s_\mathrm{tt}, z_\mathrm{tt}^w)$$
$$\delta(s_0, a_x^\perp \cup b_?^w) = (s_\mathrm{tt}, z_?^w)$$
$$\delta(s_0, a_x^\perp \cup b_\perp^?) = (s_\mathrm{tt}, z_\perp^?)$$
$$\delta(s_0, a_x^\perp \cup b_{v\in\mathrm{tf}}^w) = (s_\mathrm{ff}, z_\mathrm{ff}^w)$$
$$\delta(s_0, a_x^\perp \cup b_?^w) = (s_\mathrm{ff}, z_?^w)$$
$$\delta(s_0, a_x^\perp \cup b_\perp^?) = (s_\mathrm{ff}, z_\perp^?)$$
$$\delta(s_0, a_x^\perp \cup b_{v\neq\perp}^w) = (s_\perp, z_\perp^\perp)$$
$$\delta(s_0, a_x^\perp \cup b_\perp^?) = (s_\perp, z_\perp^\perp)$$
$$\delta(s_0, a_x^\perp \cup b_{v\neq\perp}^w) = (s_?, z_?^w)$$
$$\delta(s_0, a_x^\perp \cup b_\perp^?) = (s_?, z_\perp^?)$$
$$\delta(s_\mathrm{tt}, a_\perp^\perp \cup b_\perp^w) = (s_\mathrm{tt}, z_\perp^w)$$
$$\delta(s_\mathrm{tt}, a_\perp^\perp \cup b_{v\in\mathrm{tf}}^w) = (s_\mathrm{tt}, z_\mathrm{tt}^w)$$
$$\delta(s_\mathrm{tt}, a_\perp^\perp \cup b_?^w) = (s_\mathrm{tt}, z_?^w)$$
$$\delta(s_\mathrm{tt}, a_\mathrm{tt}^\perp \cup b_{v\in\mathrm{tf}}^w) = (s_\mathrm{tt}, z_\mathrm{tt}^w)$$
$$\delta(s_\mathrm{tt}, a_\mathrm{tt}^\perp \cup b_?^w) = (s_\mathrm{tt}, z_?^w)$$
$$\delta(s_\mathrm{tt}, a_\mathrm{tt}^\perp \cup b_\perp^?) = (s_\mathrm{tt}, z_\perp^?)$$
$$\delta(s_\mathrm{tt}, a_\mathrm{tt}^y \cup b_\perp^\perp) = (s_0, z_\perp^\perp)$$
$$\delta(s_\mathrm{tt}, a_\mathrm{tt}^? \cup b_{v\neq\perp}^w) = (s_0, z_?^w)$$
$$\delta(s_\mathrm{tt}, a_\mathrm{tt}^? \cup b_\perp^?) = (s_0, z_\perp^?)$$
$$\delta(s_\mathrm{tt}, a_\mathrm{tt}^\perp \cup b_{v\in\mathrm{tf}}^w) = (s_\mathrm{ff}, z_\mathrm{ff}^w)$$
$$\delta(s_\mathrm{tt}, a_\mathrm{tt}^\perp \cup b_?^w) = (s_\mathrm{ff}, z_?^w)$$
$$\delta(s_\mathrm{tt}, a_\mathrm{tt}^\perp \cup b_\perp^?) = (s_\mathrm{ff}, z_\perp^?)$$
$$\delta(s_\mathrm{tt}, a_\mathrm{tt}^\perp \cup b_{v\neq\perp}^w) = (s_\perp, z_\perp^\perp)$$
$$\delta(s_\mathrm{tt}, a_\mathrm{tt}^\perp \cup b_\perp^?) = (s_\perp, z_\perp^\perp)$$
$$\delta(s_\mathrm{tt}, a_\mathrm{tt}^\perp \cup b_{v\neq\perp}^w) = (s_?, z_?^w)$$
$$\delta(s_\mathrm{tt}, a_\mathrm{tt}^\perp \cup b_\perp^?) = (s_?, z_\perp^?)$$
$$\delta(s_\mathrm{ff}, a_\perp^\perp \cup b_\perp^w) = (s_\mathrm{ff}, z_\perp^w)$$
$$\delta(s_\mathrm{ff}, a_\perp^\perp \cup b_{v\in\mathrm{tf}}^w) = (s_\mathrm{ff}, z_\mathrm{ff}^w)$$
$$\delta(s_\mathrm{ff}, a_\perp^\perp \cup b_?^w) = (s_\mathrm{ff}, z_?^w)$$
$$\delta(s_\mathrm{ff}, a_\mathrm{ff}^\perp \cup b_{v\in\mathrm{tf}}^w) = (s_\mathrm{ff}, z_\mathrm{ff}^w)$$
$$\delta(s_\mathrm{ff}, a_\mathrm{ff}^\perp \cup b_?^w) = (s_\mathrm{ff}, z_?^w)$$
$$\delta(s_\mathrm{ff}, a_\mathrm{ff}^\perp \cup b_\perp^?) = (s_\mathrm{ff}, z_\perp^?)$$
$$\delta(s_\mathrm{ff}, a_\mathrm{ff}^y \cup b_\perp^\perp) = (s_0, z_\perp^\perp)$$
$$\delta(s_\mathrm{ff}, a_\mathrm{ff}^? \cup b_{v\neq\perp}^w) = (s_0, z_?^w)$$
$$\delta(s_\mathrm{ff}, a_\mathrm{ff}^? \cup b_\perp^?) = (s_0, z_\perp^?)$$
$$\delta(s_\mathrm{ff}, a_\mathrm{ff}^\perp \cup b_{v\in\mathrm{tf}}^w) = (s_\mathrm{tt}, z_\mathrm{tt}^w)$$
$$\delta(s_\mathrm{ff}, a_\mathrm{ff}^\perp \cup b_?^w) = (s_\mathrm{tt}, z_?^w)$$
$$\delta(s_\mathrm{ff}, a_\mathrm{ff}^\perp \cup b_\perp^?) = (s_\mathrm{tt}, z_\perp^?)$$
$$\delta(s_\mathrm{ff}, a_\mathrm{ff}^\perp \cup b_{v\neq\perp}^w) = (s_\perp, z_\perp^\perp)$$
$$\delta(s_\mathrm{ff}, a_\mathrm{ff}^\perp \cup b_\perp^?) = (s_\perp, z_\perp^\perp)$$
$$\delta(s_\mathrm{ff}, a_\mathrm{ff}^\perp \cup b_{v\neq\perp}^w) = (s_?, z_?^w)$$
$$\delta(s_\mathrm{ff}, a_\mathrm{ff}^\perp \cup b_\perp^?) = (s_?, z_\perp^?)$$
$$\delta(s_\perp, a_\perp^\perp \cup b_v^w) = (s_\perp, z_\perp)$$
$$\delta(s_?, a_\perp^\perp \cup b_\perp^w) = (s_?, z_\perp^w)$$
$$\delta(s_?, a_\perp^\perp \cup b_{v\neq\perp}^w) = (s_?, z_?^w)$$
$$\delta(s_?, a_?^\perp \cup b_{v\neq\perp}^w) = (s_?, z_?^w)$$
$$\delta(s_?, a_?^\perp \cup b_\perp^?) = (s_?, z_\perp^?)$$
$$\delta(s_?, a_?^y \cup b_\perp^\perp) = (s_0, z_\perp^\perp)$$
$$\delta(s_?, a_?^? \cup b_{v\neq\perp}^w) = (s_0, z_?^w)$$
$$\delta(s_?, a_\perp^? \cup b_\perp^w) = (s_0, z_\perp^w)$$
$$\delta(s_?, a_\perp^? \cup b_{v\neq\perp}^w) = (s_0, z_?^w)$$
$$\delta(s_?, a_?^\perp \cup b_{v\in\mathrm{tf}}^w) = (s_\mathrm{tt}, z_\mathrm{tt}^w)$$
$$\delta(s_?, a_?^\perp \cup b_?^w) = (s_\mathrm{tt}, z_?^w)$$
$$\delta(s_?, a_?^\perp \cup b_\perp^?) = (s_\mathrm{tt}, z_\perp^?)$$
$$\delta(s_?, a_?^\perp \cup b_{v\in\mathrm{tf}}^w) = (s_\mathrm{ff}, z_\mathrm{ff}^w)$$
$$\delta(s_?, a_?^\perp \cup b_?^w) = (s_\mathrm{ff}, z_?^w)$$
$$\delta(s_?, a_?^\perp \cup b_\perp^?) = (s_\mathrm{ff}, z_\perp^?)$$
$$\delta(s_?, a_?^\perp \cup b_{v\neq\perp}^w) = (s_\perp, z_\perp^\perp)$$
$$\delta(s_?, a_?^\perp \cup b_\perp^?) = (s_\perp, z_\perp^\perp)$$

Figure 5.7: The transition relation of the NFST created for an equation $z := \mathbf{next}(a, b)$.

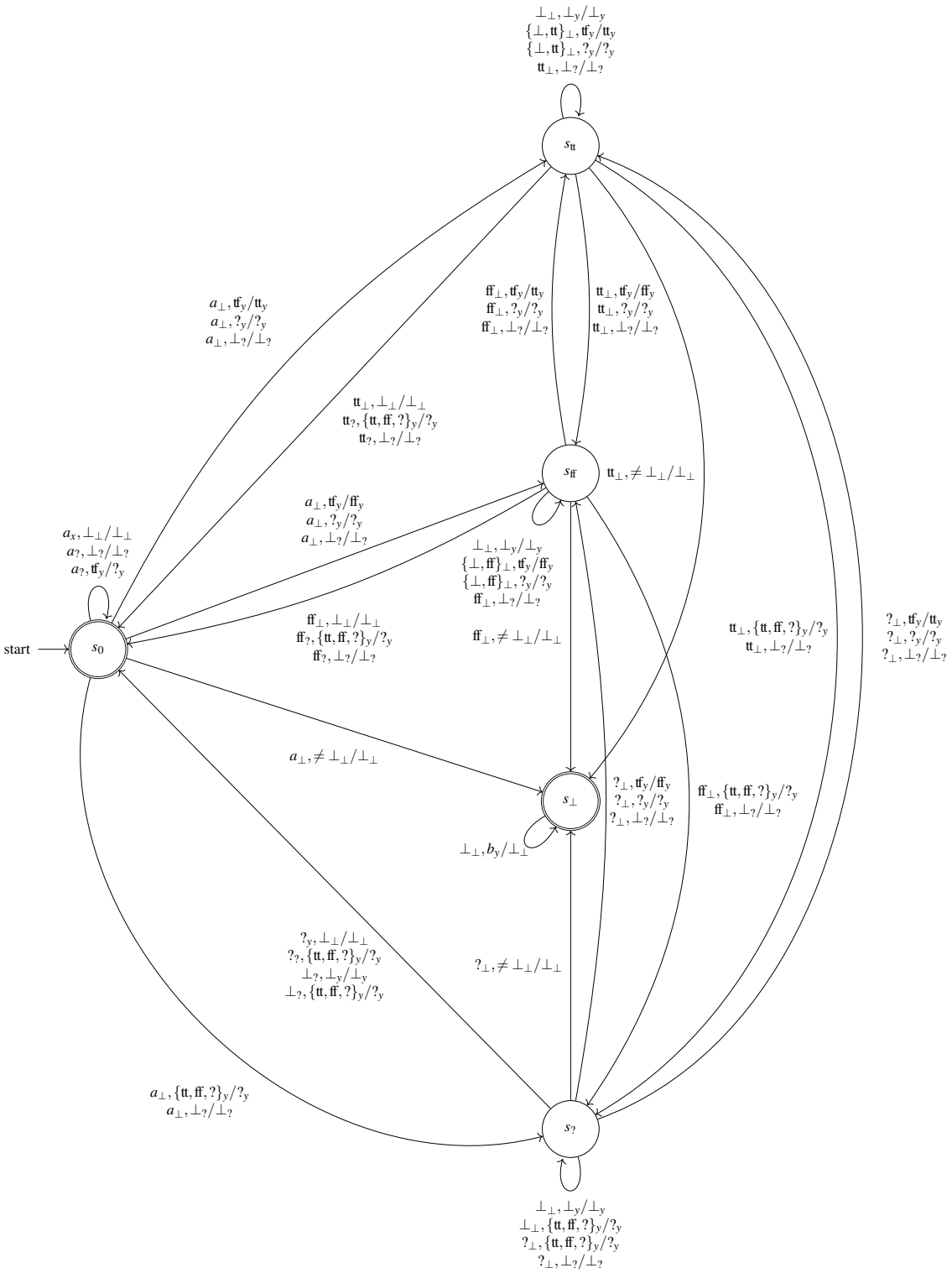


Figure 5.8: The transducer for the **next** operator in the TeSSLa$^f_{\text{bool}}$ fragment. For a **next**(a, b), the first input of every transition represents the input on a while the second represents the input on b.

Furthermore, for the other operators, the function toNFST is very similar as toDFST with slight adjustments, because the underlying stream model differs a bit from the one which was used for toDFST. Hence, we add additional transitions in the transducers for **last** and **lift** to handle ? which arise possibly between other values. Additionally, all states of the transducers for **nil**, **unit**, **lift** and **last** are now accepting.

Finally, we need to slightly adjust the composition algorithm for functional NFSTs, because of the new, non-deterministic transducer for the **next** operator which also needs accepting states. In the end, it does not differ much from the one for DFSTs.

Only minor changes were made for handling the non-determinism. First, we are also building the accepting states as all those where both states in the original transducers were accepting. Furthermore, we handle the transition function slightly different, because there can now be more next states than only one.

At first, we compose two existing transducers into a new one until only one transducer is left. This is done in the following way: Let $R = (I \to \text{Val}, O \to \text{Val}, Q, q_0, F, \delta)$ and $R' = (I' \to \text{Val}, O' \to \text{Val}, Q', q'_0, F', \delta')$ be two NFSTs. The parallel composition of R and R' is then $R'' = (I \cup I' \to \text{Val}, O \cup O' \to \text{Val}, Q \times Q', (q_0, q'_0), F \times F', \delta'')$ with

$$((s'_1, s'_2), h'') \in \delta''((s_1, s_2), g'') \Longleftrightarrow (s'_1, h) \in \delta(s_1, g) \wedge (s'_2, h') \in \delta'(s_2, g') \wedge \\ g'' = g \cup g' \wedge \forall \sigma \in I \cap I' : g(\sigma) = g'(\sigma) \wedge h'' = h \cup h'$$

In the end, again, we have one transducer $R_A = (I_A \to \text{Val}, O_A \to \text{Val}, Q_A, q_{0A}, F_A, \delta_A)$ which represents all equations. As before, R_A contains transitions with the same in- and output values for certain propositions. In the same way as for DFSTs, we build the final transducer as $R_\varphi = (I_A \backslash O_A \to \text{Val}, O_A \to \text{Val}, Q_A, q_{0A}, F_A, \delta_\varphi)$, where

$$(s', h) \in \delta_\varphi(s, g) \Longleftrightarrow (s', h) \in \delta_A(s, g') \wedge g = g'|_{I_A \backslash O_A} \wedge (\forall a \in I_A \cap O_A : g'(a) = h(a))$$

for $g|_I := g \cap (I \times \text{Val})$. In this, again, the only change is that we handle the fact of multiple follow up states for an input in the transition function.

Note that the resulting transducer is indeed functional. This follows from the fact that the transducer for the additional **next** operator is functional (and the transducers for all other operators are deterministic) and that functionality is compositional, which means that after the composition of the transducers, because the transducers we start with are functional, the result is as well.

5.4.3 Results on TeSSLa$^{f}_{\text{bool}}$

By using the result on the relation between TeSSLa$^{f}_{\text{bool}}$ and functional NFSTs, we can now give results on equivalence for TeSSLa$^{f}_{\text{bool}}$. This follows directly from the fact that the functional NFST for a TeSSLa$^{f}_{\text{bool}}$ formula is exponentially larger then the formula and therefore the result for equivalence on functional NFSTs is as well. Again, we only show the inclusion in EXPSPACE and therefore, we do not give a completeness result.

Theorem 5.21 (*Equivalence of TeSSLa$^{f}_{bool}$*)

Equivalence of TeSSLa$^{f}_{bool}$-formulas is in EXPSPACE.

Proof. In Theorem 5.20 we have shown how TeSSLa$^{f}_{\text{bool}}$ and functional NFSTs relate. Additionally, in Theorem 5.12 we have shown that the DFST build for a TeSSLa$_{\text{bool}}$ formula is exponentially larger due to possibly nested **last**s. The only addition for functional NFSTs is the transducer for the **next** operator. As for the **last**, this one is also growing exponentially for every **next** added.

As stated in [Ser99], equivalence for functional NFSTs is in PSPACE. Because the functional NFST for a TeSSLa$^{f}_{\text{bool}}$ formula is exponentially larger in the worst case, equivalence for TeSSLa$^{f}_{\text{bool}}$ is in EXPSPACE. □

As next statement about TeSSLa$^{f}_{\text{bool}}$ we show that it is, compared to TeSSLa$_{\text{bool}}$, not finite memory in general. This is due to the addition of the **next** operator, which, when used recursively, may need to wait until it reaches a certain value in the stream and the recursion ends to be evaluated. Until then, every event that happens needs to be remembered.

Theorem 5.22 (*Finite Memory and TeSSLa$^{f}_{bool}$*)

TeSSLa$^{f}_{bool}$ under E_{TeSSLa} is not finite memory.

Proof. Consider a TeSSLa$^{f}_{\text{bool}}$ formula like

$$s = \text{if } t \text{ then } \mathbf{next}(s,t) \wedge x \text{ else } x$$

where t and x are input streams. The stream s is at the current point in time only true if x is true until t is false. The number of **next**-steps which have to be remembered depends on how many

events on t and x occur before t is false once, which means that a possibly unbounded amount of memory if needed to evaluate this formula under E_{TeSSLa}. □

Even though not every $\text{TeSSLa}^{f}_{\text{bool}}$ specification is finite memory, we can, as for the stack fragment of TeSSLa, decide if a given specification is finite memory. By adding the **next** operator, several problems occur. When evaluating one timestamp after another, when the **next** is triggered, the evaluation strategy has to wait for the following value on the other stream until it can output an event and therefore memorize the current timestamp. Additionally, all other parts of the formula which depend on a **next** and also get input from other parts of the formula have to buffer all those input events, because a **next** may output older timestamps which must then be processed with the other input events with the corresponding timestamp. A simple specification where this case occurs is the following:

$$x := \mathbf{next}(a,b) \wedge c$$

with the boolean input streams a, b and c. Now, according to the description above and the definition of E_{TeSSLa}, the **next** operator has to remember an arbitrarily large timestamp every time an event on its second input stream occurs. The other input stream of the **next** does not matter for this purpose. Therefore, to find out if a given $\text{TeSSLa}^{f}_{\text{bool}}$ formula is finite memory is the same as finding out if there exist a combination of inputs such that the second input stream of a **next** operator gets an event.

The proof for the following theorem is based on building the NFST for the given $\text{TeSSLa}^{f}_{\text{bool}}$ formula and checking afterwards, if a non-deterministic decision is reachable from the starting state. Because this means that a **next** gets an event on the second input stream as the transducer then guesses which value has to be output.

Theorem 5.23 (*FMP for TeSSLa^{f}_{bool} Specifications*)

FMP of TeSSLa^{f}_{bool} under E_{TeSSLa} is in EXPTIME.

Proof. To find out if a $\text{TeSSLa}^{f}_{\text{bool}}$ specification is finite memory, we have to check if a **next** can get an event on its second input stream. For this purpose, consider the NFST created from the given specification. Exactly when a **next** gets an event on its second input stream, the NFST has a non-deterministic choice in a state. Therefore, we create an NFST and make a depth-first-search, beginning in the starting state, to find a state with a non-deterministic choice.

As the NFST is exponentially larger than the TeSSLa$^{f}_{\text{bool}}$ specification it is created from and the depth-first-search is linear in the size of the NFST, deciding FMP for TeSSLa$^{f}_{\text{bool}}$ is in EXPTIME. □

Lastly, in the following theorem, we show that RFM for TeSSLa$^{f}_{\text{bool}}$ is in EXPTIME. The proof for this statement again is based on the relation of TeSSLa$^{f}_{\text{bool}}$ to functional NFSTs and the fact that for functional NFSTs it can be decided in PTIME if one can be transformed into a semantically equivalent TM which only uses a bounded number of binary tape cells.

Theorem 5.24 (*RFM for TeSSLa$^{f}_{bool}$*)

RFM of TeSSLa$^{f}_{bool}$ is in EXPTIME.

Proof. As shown before, the functional NFST for a TeSSLa$^{f}_{\text{bool}}$ formula is exponential in size of the formula. As stated in [Ser99], the question if a given functional NFST can be transformed into a semantically equivalent Turing machine that only needs a bounded number of binary cells on its work tapes is the same as if the functional NFST is *subsequentializable*. Additionally, [WK95] showed that the question if a functional NFST is subsequentializable is in PTIME. This means that the same question for TeSSLa$^{f}_{\text{bool}}$ can be solved by transforming a formula into the corresponding function NFST and then check if it is subsequentializable. Because the size of the functional NFST is exponential, RFM for TeSSLa$^{f}_{\text{bool}}$ is in EXPTIME. □

5.5 Timed Fragment

In this section, we consider a new fragment which extends TeSSLa$_{\text{bool}}$ by the comparison of a timestamp with a certain distance from a previous timestamp. We call this fragment TeSSLa$_{\text{bool+c}}$ for having boolean streams and comparison of timestamps with constants. It shows that by allowing some more freedom within the functions allowed for **lift**, TeSSLa directly reaches the expressive power of timed transducers.

As in TeSSLa$_{\text{bool}}$ we are still restricted to the operators **last** and **lift** and have nearly only boolean streams, but this time additionally allowing the restricted usage of the **time** operator within **lift**, which allows us to get the timestamps of events on streams and compare them with constants. Compared to TeSSLa$_{\text{bool}}$, a new type of expressions $\textbf{lift}(g_v)(\textbf{time}(e), \mathsf{merge}(\textbf{last}(\textbf{time}(e), e), 0))$ is added, in which the first and the last e need to be the same expression and the merge is used

to ensure that the second input stream of the **lift** is initialized with zero when the expression is triggered, if no event on the value stream of the **last** already occurred. Using g_v, this type of expressions exactly represent how clock constraints work in timed automata [AH92, AD94], and thus timed transducers, as we will see later.

Definition 5.25 (*TeSSLa$_{bool+c}$, [CHL+18]*)

A TeSSLa formula φ is called a TeSSLa$_{bool+c}$ formula if $\varphi : S_{\mathbb{B}} \times \ldots \times S_{\mathbb{B}} \to S_{\mathbb{B}} \times \ldots \times S_{\mathbb{B}}$ and the syntax of every equation e is restricted as follows, where $f : \mathbb{B}^n_\perp \rightarrowtail \mathbb{B}_\perp$, $v \in \mathbb{T}$ is a constant and g_v is a function $g_v : \mathbb{T} \times \mathbb{T} \to \mathbb{B}$ of the form $g_v(t_1,t_2) = t_1 \lessgtr t_2 + v$ with $\lessgtr \in \{<,>\}$:

$$e := \textbf{nil} \mid \textbf{unit} \mid x \mid \textbf{lift}(f)(e,\ldots,e) \mid \textbf{lift}(g_v)(\textbf{time}(e), \mathit{merge}(\textbf{last}(\textbf{time}(e),e),0)) \mid \textbf{last}(e,e)$$

where the newly added expressions of the form $\textbf{lift}(g_v)(\textbf{time}(e), \mathit{merge}(\textbf{last}(\textbf{time}(e),e),0))$ are restricted to $\textbf{lift}(g_v)(\textbf{time}(a), \mathit{merge}(\textbf{last}(\textbf{time}(b),a),0))$, thus, the first and the last expression need to be the same.

The semantics for every single operator stay the same as in TeSSLa.

Compared to the expression with slift, for which we showed for the boolean fragment that it does not increase the expressiveness, the new type of expressions for this fragment of the form $\textbf{lift}(g_v)(\textbf{time}(e), \mathsf{merge}(\textbf{last}(\textbf{time}(e),e),0))$ does increase the expressiveness.

We will now show that TeSSLa$_{\text{bool+c}}$ and DTFSTs have the same expressiveness. As for TeSSLa$_{\text{bool}}$, we again encode words as streams and vice versa using the same procedure, but this time α_Σ and β_Σ preserve the timestamps. As before, we will then transform every type of TeSSLa$_{\text{bool+c}}$ formula into a DTFST and use a very similar composition algorithm afterwards and vice versa by transforming the timing constraints of a DTFST into TeSSLa expressions using the new type of expressions we added in the syntax. We will now define α_Σ and β_Σ and then prove the statement later.

This time, the function $\alpha_\Sigma(w) = S$ encodes a DTFST word $w = (w_0,\tau_0)(w_1,\tau_1)\ldots(w_n,\tau_n) \in (\Sigma \times \mathbb{T})^*$ as corresponding set of TeSSLa$_{\text{bool+c}}$ streams, where for every $p \in \Sigma$ a stream $s_p \in S$ exists with

$$s_p = \tau_0 d_0 \tau_1 d_1 \ldots \tau_{n-1} d_{n-1} \tau_n d_n \infty \Leftrightarrow \forall i : (d_i \Leftrightarrow w_i = p)$$

Therefore, this time we do not use artificial timestamps which resemble the position of the word, but instead use the timestamps which exists in the time word now.

The function $\beta_\Sigma(s_1,\ldots,s_k) = w = (w_0,\tau_0)(w_1,\tau_1)\ldots(w_n,\tau_n) \in (\Sigma \times \mathbb{T})^*$ encodes $\text{TeSSLa}_{\text{bool+c}}$ streams as a synchronized DTFST word w over the same alphabet we already used for $\text{TeSSLa}_{\text{bool}}$, $\Sigma = \{z_1,\ldots,z_k\} \to \text{Val}$ with $\text{Val} = \{\bot, \text{tt}, \text{ff}, <', \bot', \text{tt}', \text{ff}'\}$, where $z_1,\ldots,z_k$ are variables which relate to the corresponding streams $s_1,\ldots,s_k$ again. Let again $T = \{t_0,t_1,t_2,\ldots,t_n\} \setminus \{\infty\}$ with $t_0 = 0$ be the set of all timestamps present in the streams including 0 and excluding ∞ with $t_i < t_{i+1}$. Then $\tau_i = t_i$ and each w_i is build as follows:

$$w_i(s) = \begin{cases} <' & \text{if } s = vt_i \\ s(t_i)' & \text{if } s = vt_id \\ s(t_i) & \text{if } \exists t \in T : t > t_i \vee s = v\infty \\ \bot & \text{otherwise} \end{cases}$$

Which is exactly how the symbols of the word have already been build for $\text{TeSSLa}_{\text{bool}}$.

Hence β_Σ encodes the streams as a word with the real timestamps of the events in the stream now instead of adding just indices as timestamps as for $\text{TeSSLa}_{\text{bool}}$, which allows us to later compare those timestamps in the DTFST and vice versa using α_Σ.

Because both functions preserve the timestamps, both representations are now isomorphic and we can use the inverse encoding functions for decoding:

Theorem 5.26 (*Relation between $\text{TeSSLa}_{\text{bool+c}}$ and DTFSTs, [CHL+18]*)

For a DTFST $R = (\Sigma,\Gamma,Q,q_0,C,\delta)$ a $\text{TeSSLa}_{\text{bool+c}}$ formula φ_R exists and for a $\text{TeSSLa}_{\text{bool+c}}$ formula φ a DTFST $R_\varphi = (\Sigma,\Gamma,Q,q_0,C,\delta)$ exists:

$$[\![R]\!] = \alpha_\Gamma^{-1} \circ [\![\varphi_R]\!] \circ \alpha_\Sigma \quad \textit{and} \quad [\![\varphi]\!] = \beta_\Gamma^{-1} \circ R_\varphi \circ \beta_\Sigma.$$

In the next to sections, we will show the translations in both directions and prove the theorem by doing this.

5.5.1 Translating DTFST to $\text{TeSSLa}_{\text{bool+c}}$

Given a DTFST $R = (\Sigma,\Gamma,Q,q_0,\delta,C)$ we will now show how to create the $\text{TeSSL}_{\text{bool+c}}$ formula φ_R from the previous theorem. Therefore, we reuse the translation for DFSTs with the following adjustments. Recall, that for every q, x_q denoted if the transducer is in state q and a stream $d_{q,\sigma,q',\gamma}$

indicated if the transition from state q with input symbol σ to state q' with output symbol γ is active.

For every single transition $\eta_i = (q, \sigma, \vartheta, q', \gamma, r)$ of the DTFST we extend the stream $d_{q,\sigma,q',\gamma}$ to $d_{q,\sigma,\vartheta,q',\gamma,r}$ (and use this instead in every x_q) by adding the timing constraint ϑ:

$$d_{q,\sigma,\vartheta,q',\gamma,r} = d_{q,\sigma,q',\gamma} \wedge \vartheta$$

Where all possible timing constraints ϑ are a boolean combination of elements of the form $T \leqslant x + c$ in the DTFST and are translated by lifting the boolean combination to streams element-wise, as follows for every $T \leqslant x + c$:

$$\mathbf{time}(s_\sigma) \leqslant \mathsf{merge}(\mathbf{last}(\mathbf{time}(\mathsf{merge}(b_x, \mathbf{unit})), s_\sigma), 0) + c.$$

In this case b_x is the stream representing the clock of the constraint as defined later. The part $\mathbf{time}(s_\sigma)$ always represents the current time because all streams always have an event when something happens on any stream because of how we encode the streams. We encode the clocks just as boolean streams, thus $\mathbf{time}(\mathsf{merge}(b_x, \mathbf{unit}))$ is the current time remembered by the clock. The **last** surrounding this part ensures that not the current value of the clock is taken if a new one is set, but instead the previous one and the outer merge initialized the clocks with zero.

The streams b_x for every clock $x \in C$ are defined as follows, where the resetting of clocks is encoded by r:

$$b_x := \mathsf{merge}\{\mathsf{filter}(d_{q,\sigma,\vartheta,q',\gamma,r}, d_{q,\sigma,\vartheta,q',\gamma,r}) \mid (q, \sigma, \vartheta, q', \gamma, r) \in \delta \wedge x \in r\}$$

These newly defined streams allow us to add timing constraints to the transitions modelled in TeSSLa as in timed automata. If we now add those new streams to the encoding of a DFST in $\text{TeSSLa}_{\text{bool}}$, we are able to simulate a DTFST.

5.5.2 Translating $\text{TeSSLa}_{\text{bool+c}}$ to DTFST

To show the other direction, the transducers from the equations in the flattened version of φ are build as the DFSTs before for $\text{TeSSLa}_{\text{bool}}$. But additionally we now have to translate the new equations of the form $\mathbf{lift}(g_v)(\mathbf{time}(a), \mathsf{merge}(\mathbf{last}(\mathbf{time}(b), a), 0))$. We will build one single transducer for such equations instead of flattening them and build one transducer for every operator, because a single transducer for, for example **time**, is not possible, as it would have to handle arbitrary values, not just boolean values. Hence we assume that even in a flattened $\text{TeSSLa}_{\text{bool+c}}$ specification

used in this subsection, the equations of the form $\mathbf{lift}(g_v)(\mathbf{time}(a), \mathsf{merge}(\mathbf{last}(\mathbf{time}(b), a), 0))$ are considered as one operator.

The transducers created from a single equation in a $\text{TeSSLa}_{\text{bool+c}}$ formula φ for the operators **nil**, **unit**, **lift** over boolean streams and **last** are build as before with the following changes for the translation function $\mathsf{toDTFST}$: All resulting transducers now additionally have an empty set of clocks, because they are not using any clocks, and an additional timing constraint on every transition which is just true, because no timing constraints are needed for these operators. But for the composition later these neutral values are necessary, as different transducers with and without clocks and timing constraints need to be composed into one.

The transducer for an equation $z := \mathbf{lift}(g_v)(\mathbf{time}(e), \mathsf{merge}(\mathbf{last}(\mathbf{time}(a), e), 0))$ stays in its initial state s_0 until one stream ends. For every event on a it resets the clock to save the timestamp of the event in the clock c_a. For every event on e it produces an output value on z according to the fulfilment of the current clock constraint, hence, depending on the fulfilment of the clock constraint, the transducer takes a different transition and produces a different output. The states s_v and s_e handle the stream endings similarly to the transducer for the **last** operator, which is a result of a **last** being a subformula of the expression.

Note that, because we consider expressions of the form $\mathbf{lift}(g_v)(\mathbf{time}(e), \mathsf{merge}(\mathbf{last}(\mathbf{time}(a), e), 0))$ as one operator for building transducers, it has a lot less states than it would normally have when every operator would be translated on its own and the resulting transducers would be combined via composition. Most of the functionality that would normally be encoded in states is now directly combined in the timing constraints. This direct translation of that many TeSSLa operators into a transducer is only directly possible because the timing constraints in DTFSTs directly resemble the functionality.

Formally, the new type of equation $z := \mathbf{lift}(g_v)(\mathbf{time}(e), \mathsf{merge}(\mathbf{last}(\mathbf{time}(a), e), 0))$ is translated into an DTFST as follows:

$\mathsf{toDTFST}(z := \mathbf{lift}(g_v)(\mathbf{time}(e), \mathsf{merge}(\mathbf{last}(\mathbf{time}(a), e), 0))) =$
$(\{a, e\} \to \text{Val}, \{z\} \to \text{Val}, \{s_0, s_v, s_e\}, s_0, \{c_a\}, \delta)$ where $g_v(t_1, t_2) = t_1 \lesseqgtr t_2 + v$ and we use the same

$\bot, \bot$	$[true]$	$/\emptyset, \bot$
$\bot, \{\mathrm{tt}, \mathrm{ff}\}$	$[true]$	$/\{c_a\}, \bot$
$\{\mathrm{tt}, \mathrm{ff}\}, \bot$	$[T \leqslant c_a + v]$	$/\emptyset, \mathrm{tt}$
$\{\mathrm{tt}, \mathrm{ff}\}, \bot$	$[\neg T \leqslant c_a + v]$	$/\emptyset, \mathrm{ff}$
$\{\mathrm{tt}, \mathrm{ff}\}, \{\mathrm{tt}, \mathrm{ff}\}$	$[T \leqslant c_a + v]$	$/\{c_a\}, \mathrm{tt}$
$\{\mathrm{tt}, \mathrm{ff}\}, \{\mathrm{tt}, \mathrm{ff}\}$	$[\neg T \leqslant c_a + v]$	$/\{c_a\}, \mathrm{ff}$

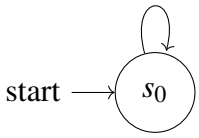


Figure 5.9: The transducer for the in $\text{TeSSLa}_{\text{bool+c}}$ newly added expressions of the form $z := \mathbf{lift}(g_v)(\mathbf{time}(e), \mathsf{merge}(\mathbf{last}(\mathbf{time}(a), e), 0))$, where the first input is e and the second is a. As before, the ending cases and states s_v and s_e have been left out for a better overview on the functionality of the expression.

abbreviations as for toDFST:

$$\begin{aligned}
\delta(s_0, e_\bot \cup a_\bot, \text{true}) &= (s_0, \emptyset, z_\bot) \\
\delta(s_0, e_\bot \cup a_{y \in \mathrm{tf}}, \text{true}) &= (s_0, \{c_a\}, z_\bot) \\
\delta(s_0, e_\bot \cup a'_y, \text{true}) &= (s_v, \emptyset, z_\bot) \\
\delta(s_0, e_{x \in \mathrm{tf}} \cup a_\bot, T \leqslant c_a + v) &= (s_0, \emptyset, z_\mathrm{tt}) \\
\delta(s_0, e_{x \in \mathrm{tf}} \cup a_\bot, \neg(T \leqslant c_a + v)) &= (s_0, \emptyset, z_\mathrm{ff}) \\
\delta(s_0, e_{x \in \mathrm{tf}} \cup a_{y \in \mathrm{tf}}, T \leqslant c_a + v) &= (s_0, \{c_a\}, z_\mathrm{tt}) \\
\delta(s_0, e'_{x \in \{<, \bot\}} \cup a^?_y, \text{true}) &= (s_e, \emptyset, z'_x) \\
\delta(s_0, e'_{x \in \mathrm{tf}} \cup a^?_y, T \leqslant c_a + v) &= (s_e, \emptyset, z'_\mathrm{tt}) \\
\delta(s_0, e'_{x \in \mathrm{tf}} \cup a^?_y, \neg(T \leqslant c_a + v)) &= (s_e, \emptyset, z'_\mathrm{ff}) \\
\delta(s_v, e_{y \in \mathrm{tf}} \cup a^?_x, \text{true}) &= (s_e, \emptyset, z'_<) \\
\delta(s_v, e_\bot \cup a^?_x, \text{true}) &= (s_v, \emptyset, z_\bot) \\
\delta(s_v, e'_\bot \cup a^?_x, \text{true}) &= (s_e, \emptyset, z'_\bot) \\
\delta(s_v, e'_{y \in \{<, \mathrm{tt}, \mathrm{ff}\}} \cup a^?_x, \text{true}) &= (s_e, \emptyset, z'_<)
\end{aligned}$$

The transducer without the ending edge cases is depicted in Figure 5.9.

The parallel composition algorithm for DFSTs is reused here and extended by clocks and conjuncting the timing constraints of the composed transducers as follows: Assuming R and R' have the sets of clocks C and C' respectively. Then the resulting transducer R'' from a parallel composition

of R and R' has the set of clocks $C'' = C \cup C'$. Furthermore, the definition of the transition function δ'' of R'' is updated as follows, where δ and δ' are the transition functions of R and R', respectively. The biggest change is the addition of the clock constraints and that they are combined to ϑ'' for the resulting transducer as $\vartheta'' = \vartheta \wedge \vartheta'$. Additionally, the set of clocks to reset for δ'' is the union of the sets of the original transitions.

$$\begin{aligned}\delta''((s_1,s_2),g'',\vartheta'') = ((s_1',s_2'),h'',r'') \Longleftrightarrow & \delta(s_1,g,\vartheta) = (s_1',h,r) \wedge \delta'(s_2,g',\vartheta') = (s_2',h',r') \wedge \\ & g'' = g \cup g' \wedge \forall \sigma \in I \cap I' : g(\sigma) = g'(\sigma) \wedge h'' = h \cup h' \\ & r'' = r \cup r' \wedge \vartheta'' = \vartheta \wedge \vartheta'\end{aligned}$$

Because all transducers which do not need a timing constraint naturally have true on every transition as constraint, it works, because it is the neutral element for conjunction.

Afterwards the same closure algorithm is applied as for the DFSTs because nothing has to be changed about the clocks or clock constraints.

5.5.3 Results for TeSSLa$_{\text{bool+c}}$

By using the previous result we can now make statements on equivalence of TeSSLa$_{\text{bool+c}}$ formulas. The proof is mainly based on the fact that equivalence of deterministic timed automata is in PSPACE [AD94] and that the constructed DTFSTs can be represented as those.

Theorem 5.27 (*Equivalence for TeSSLa$_{bool+c}$, [CHL+18]*)

Equivalence of TeSSLa$_{bool+c}$ formulas is in EXPSPACE.

Proof. Above, we have shown how TeSSLa$_{\text{bool+c}}$ relates to DTFSTs. To show the complexity of the equivalence problem for TeSSLa$_{\text{bool+c}}$ we use the same approach as for TeSSLa$_{\text{bool}}$ in Theorem 5.12 and show the complexity of DTFSTs by converting them to deterministic timed automata. To do this, we just copy the clocks and clock constraints from the DTFSTs to the construction used for DFSTs. Because equivalence for deterministic timed automata is in PSPACE [AD94] and the constructed DTFST is exponential in the size of the TeSSLa$_{\text{bool+c}}$-formula, equivalence for TeSSLa$_{\text{bool+c}}$ is in EXPSPACE. □

In the following theorem we state that TeSSLa$_{\text{bool+c}}$ is not finite memory in general. This follows from the fact that arbitrarily large timestamps have to be compared.

Theorem 5.28 (*Finite Memory and TeSSLa$_{bool+c}$*)

TeSSLa$_{bool+c}$ under E_{TeSSLa} is not finite memory.

Proof. When evaluating a TeSSLa$_{\text{bool+c}}$ formula under E_{TeSSLa}, there are especially those with subexpressions like $\mathbf{lift}(g_v)(\mathbf{time}(e), \mathsf{merge}(\mathbf{last}(\mathbf{time}(e), e), 0))$. When evaluating such a subexpression with the given evaluation strategy, the part with $\mathbf{last}(\mathbf{time}(e), e)$ needs to remember an arbitrary large timestamp, which means that the memory needs to be unbounded. Because this means there are TeSSLa$_{\text{bool+c}}$ formulas which are not finite memory under E_{TeSSLa}, TeSSLa$_{\text{bool+c}}$ is not finite memory. □

Next, we show that FMP for TeSSLa$_{\text{bool+c}}$ under E_{TeSSLa} is in EXPTIME. Because for TeSSLa$_{\text{bool}}$ FMP is always fulfilled as every TeSSLa$_{\text{bool}}$ formula is finite memory, we only have to consider the new type of expressions. As the new type of expressions does compare arbitrarily large timestamps, a formula can directly not be evaluated with finite memory under E_{TeSSLa} if such an expression exists in the specification and is able to get an event. While the question if such an expression exists is easy to solve by just checking every operator, the question if there exists an input such that timestamps have to be compared or memorized is harder to check.

The proof is based on the DTFST created for a TeSSLa$_{\text{bool+c}}$ specification as it resembles the control flow graph of the evaluation strategy E_{TeSSLa}. Even though the DTFST does not have a notion of memory, is allows us to find out which parts of the given specification are used during the evaluation of an input. It is created compositionally from single transducers for each operator and by construction for each equation $\mathbf{lift}(g_v)(\mathbf{time}(e), \mathsf{merge}(\mathbf{last}(\mathbf{time}(e), e), 0))$, a path from the starting state to the resulting timing constraint can only exist if an input event can reach the equation in the TeSSLa$_{\text{bool+c}}$ specification.

Theorem 5.29 (*FMP for TeSSLa$_{bool+c}$*)

FMP for TeSSLa$_{bool+c}$ under E_{TeSSLa} is in EXPTIME.

Proof. When applying E_{TeSSLa} to a TeSSLa$_{\text{bool+c}}$ formula, subformulas using timestamps, i.e. $\mathbf{lift}(g_v)(\mathbf{time}(e), \mathsf{merge}(\mathbf{last}(\mathbf{time}(e), e), 0))$, are the reason why a TeSSLa$_{\text{bool+c}}$ formula may not be finite memory. If any of the input streams of such an expression gets an event, E_{TeSSLa} needs to memorize an arbitrarily large timestamp.

Consider the DTFST build for an expression $\mathbf{lift}(g_v)(\mathbf{time}(e), \mathsf{merge}(\mathbf{last}(\mathbf{time}(e), e), 0))$. As one can see, if any on the input streams has an event, either a timing constraint has to be evaluated or a

clock has to be set to a certain value. Those are exactly the two cases where a given formula using such an expression is not finite memory. Therefore, we build the DTFST for a given $\text{TeSSLa}_{\text{bool+c}}$ formula and make a depth-first-search to find out if we can reach any transition either having a timing constraint or resetting a clock.

As the DTFST is exponentially larger than the $\text{TeSSLa}_{\text{bool+c}}$ formula and the depth-first-search can be done in linear time regarding to the size of the automaton, FMP for $\text{TeSSLa}_{\text{bool+c}}$ under E_{TeSSLa} is in EXPTIME. □

The last question we answer for $\text{TeSSLa}_{\text{bool+c}}$ is the one about the complexity of RFM. In this case, this question is equivalent to the one if in the resulting DTFST for a given $\text{TeSSLa}_{\text{bool+c}}$ formula, the timing constraints are really needed.

Theorem 5.30 (*RFM for TeSSLa$_{bool+c}$*)

RFM for $\text{TeSSLa}_{\text{bool+c}}$ is in EXPSPACE.

Proof. The given statement refers to the question if a given formula can be rewritten such that all occurrences of subformulas of the form $\textbf{lift}(g_v)(\textbf{time}(e), \mathsf{merge}(\textbf{last}(\textbf{time}(e), e), 0))$ can be eliminated while staying semantically equivalent.

We solve this question by reducing the problem to the question of equivalence for timed automata. Let therefore the function untime for a set S be defined as follows:

$$\text{untime}(S) = \{w_0, \ldots, w_n \mid (w_0, \tau_0) \ldots (w_n, \tau_n) \in S\}$$

First, we build the DTFST A for the given $\text{TeSSLa}_{\text{bool+c}}$ formula. By removing all timing constraints, we create a DFST B from A. Note that B could be non-deterministic in general if one removes timing constraints from an arbitrary DTFST, but it is deterministic because of the structure of the DTFST which results when it is created with the previously described algorithm from a $\text{TeSSLa}_{\text{bool+c}}$ formula, as A does not contain any two different transitions with the same input symbol.

Now, iff $\text{untime}(\mathcal{L}(A)) = \mathcal{L}(B)$ holds, the timing constraints have no effect and can be removed. Both transducers can be transformed into deterministic automata by adding their outputs to the inputs as already used for Theorem 5.12. As [AD94] states that equivalence of deterministic timed automata is in PSPACE, we can do the checking of $\text{untime}(\mathcal{L}(A)) = \mathcal{L}(B)$ in PSPACE, while taking

care of using the untimed language. Therefore, the same holds in our case. Because the DTFST is exponentially larger than the $\text{TeSSLa}_{\text{bool+c}}$ formula, RFM for $\text{TeSSLa}_{\text{bool+c}}$ is in EXPSPACE. □

In the following section, we will not add non-determinism to the timed fragment.

5.5.4 Adding Non-determinism to the Timed Fragment

As we did for the boolean fragment, we will add non-determinism to the timed fragment. Because many properties are already undecidable for non-deterministic timed automata, many properties are also undecidable for this new fragment, as we will show in this section.

Analogously to the boolean fragment, we will call the functional non-deterministic version of the timed fragment $\text{TeSSLa}^{f}_{\text{bool+c}}$. It is created by adding the **next** operator already used for $\text{TeSSLa}^{f}_{\text{bool}}$ to $\text{TeSSLa}_{\text{bool+c}}$. If we combine the translations for $\text{TeSSLa}_{\text{bool+c}}$ to DTFSTs and for $\text{TeSSLa}^{f}_{\text{bool}}$ to functional NFSTs, we can show that $\text{TeSSLa}^{f}_{\text{bool+c}}$ relates to functional NTFSTs in the same way. We just need to consider both, the non-determinism available in $\text{TeSSLa}^{f}_{\text{bool}}$ as well as the clocks and clock constraints from $\text{TeSSLa}_{\text{bool+c}}$.

For equivalence, we again refer to [AD94], which states that equivalence of NTAs is undecidable. One can see that equivalence for functional NTFSTs can only be harder than for NTAs. But because equivalence for NTAs is already undecidable, it has to be as well for functional NTFSTs.

Theorem 5.31 (*Equivalence for $\text{TeSSLa}^{f}_{\text{bool+c}}$*)

Equivalence of $\text{TeSSLa}^{f}_{\text{bool+c}}$ is undecidable.

Proof. We prove the statement by again transforming TeSSLa into transducers, functional NTFSTs to be more precise, which is done with a combination of the translations for $\text{TeSSLa}^{f}_{\text{bool}}$ to NFSTs and $\text{TeSSLa}_{\text{bool+c}}$ to DTFSTs.

Complexity for equivalence can be obtained by referring to NTAs. According to [AD94], equivalence for NTAs is undecidable. Because NTFSTs are only an extension of NTAs by adding an output, the complexity for equivalence can only be higher for NTFSTs than for NTAs. It follows from this that equivalence for functional NTFSTs is also undecidable and the same holds for $\text{TeSSLa}^{f}_{\text{bool+c}}$. □

Obviously, $\text{TeSSLa}^{f}_{\text{bool+c}}$ is not finite memory.

Theorem 5.32 (*Finite Memory and TeSSLa$^f_{bool+c}$*)

TeSSLa$^f_{bool+c}$ under E_{TeSSLa} is not finite memory.

Proof. Because TeSSLa$^f_{\text{bool}}$ and TeSSLa$_{\text{bool+c}}$ are not finite memory under E_{TeSSLa}, TeSSLa$^f_{\text{bool+c}}$ is also not finite memory under this evaluation strategy. □

By the results for TeSSLa$_{\text{bool+c}}$ and TeSSLa$^f_{\text{bool}}$, we can also get a result on FMP for TeSSLa$^f_{\text{bool+c}}$. Because both results on FMP for both fragments can be checked on their own, FMP for TeSSLa$^f_{\text{bool+c}}$ can be decided in the worst complexity of the two.

Theorem 5.33 (*FMP for TeSSLa$^f_{bool+c}$*)

FMP of TeSSLa$^f_{bool+c}$ under E_{TeSSLa} is in EXPTIME.

Proof. As stated by Theorem 5.23 and Theorem 5.29, FMP for the fragments TeSSLa$^f_{\text{bool}}$ and TeSSLa$_{\text{bool+c}}$ is in EXPTIME. We can now check those two properties independently from each other, therefore making one depth-first-search on the NTFST and either find a non-deterministic choice or a timing constraint. Because the NTFST is exponential in the size of the specification and the depth-first-search is linear, this leads to FMP being also EXPTIME for TeSSLa$^f_{\text{bool+c}}$. □

Lastly, we will show that RFM for TeSSLa$^f_{\text{bool+c}}$ is undecidable. As we have seen before, a TeSSLa$^f_{\text{bool+c}}$ formula is not finite memory under the same condition as TeSSLa$_{\text{bool+c}}$ is not, therefore as soon as one subformula of the form $\mathbf{lift}(g_v)(\mathbf{time}(e), \mathbf{last}(\mathbf{time}(e), e))$ exists, which results in clock constraints in the functional NTFST. This means, to find an equivalent formula which is finite memory, it needs to be able to remove all clocks from the NTFST. But it is undecidable for NTAs if this can be done without changing the language accepted by the NTA [Fin06] and this result carries over to NTFSTs.

Theorem 5.34 (*RFM for TeSSLa$^f_{bool+c}$*)

RFM of TeSSLa$^f_{bool+c}$ is undecidable.

Proof. For TeSSLa$_{\text{bool+c}}$ we have shown that a formula is not finite memory iff a timing constraint exists of the form $\mathbf{lift}(g_v)(\mathbf{time}(e), \mathbf{last}(\mathbf{time}(e), e))$. As all those subformulas end to be clock constraints in the resulting NTFST for a TeSSLa$^f_{\text{bool+c}}$ formula, there has to be an semantically

equivalent NTFST without any timing constraints (or without any clocks, analogously). Otherwise the given $\text{TeSSLa}^{f}_{\text{bool+c}}$ formula can not be transformed into a semantically equivalent one which is finite memory. But [Fin06] states that it is only decidable iff an NTA with one clock can be transformed into one without any clocks, but for $n \geq 2$ clocks, it is undecidable. Because our functional NTFSTs are only an extension of NTAs and have possibly an arbitrary number of clocks, RFM is undecidable for $\text{TeSSLa}^{f}_{\text{bool+c}}$. □

5.6 Conclusion

In this section we conclude the results from this chapter. The Figure 5.10 shows the relationship between the different TeSSLa fragments and well known formalisms we considered in this chapter. The formalisms where the arrows start are strictly less expressive compared to the one where the corresponding arrow ends. The formalisms without any path between them in this graph are incomparable. On the top of the graph, also the fragments from the previous chapter, Chapter 4, are depicted.

Additionally, the table in Figure 5.11 shows the complexity results on the various TeSSLa fragments which we considered in this section.

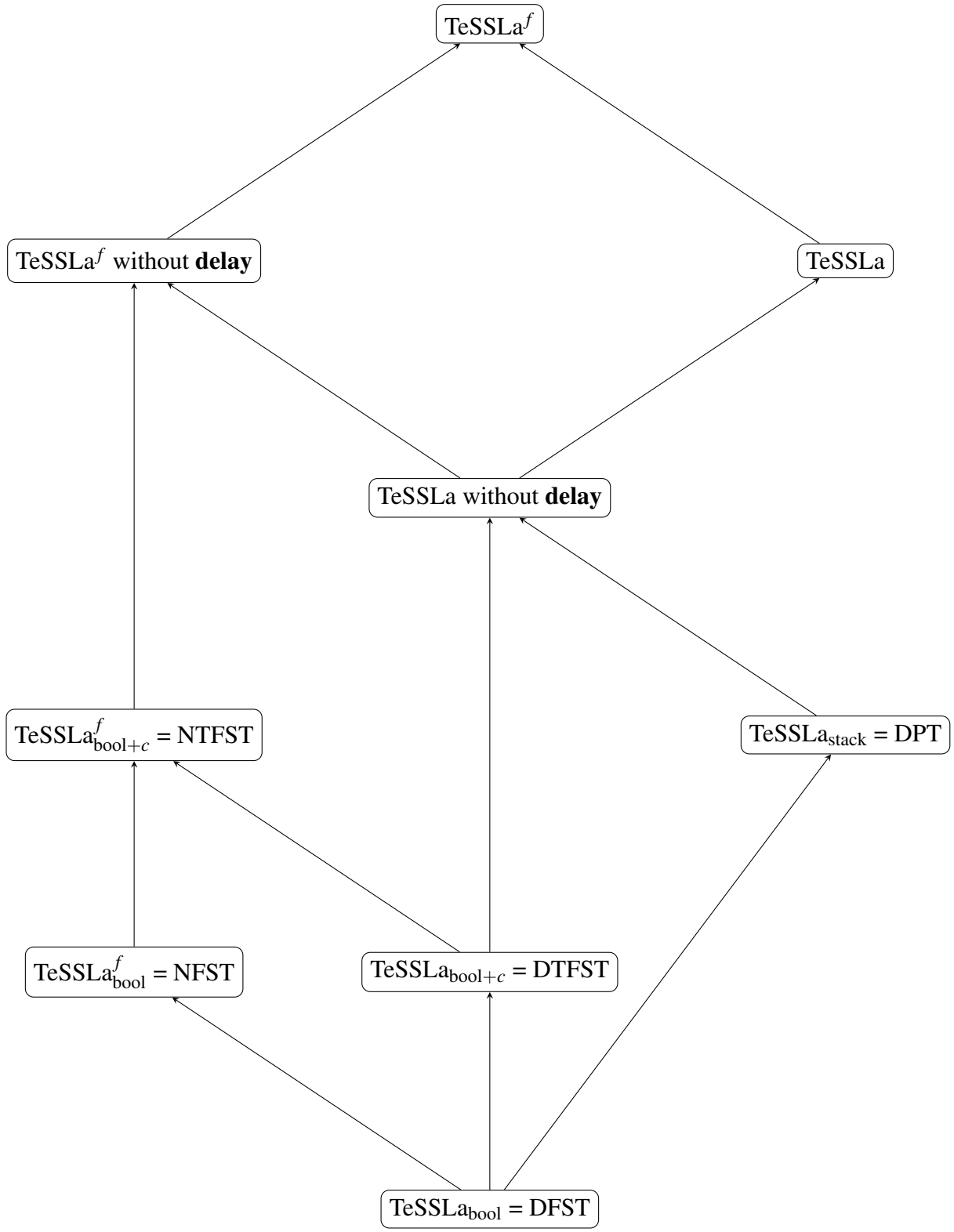


Figure 5.10: Shows the relation between various TeSSLa fragments and different types of transducers as well as the results from Chapter 4. The arrows indicate expressiveness which follows from the associated transducers. The elements in a node at the start of an arrow are strictly less expressive than the elements at its end. Elements between whose nodes no path exist are incomparable.

Fragment	Equiv. Formalism	Equivalence	FM	FMP	RFM
$\text{TeSSLa}_{\text{bool}}$	DFST Theorem 5.9	EXPTIME Theorem 5.12	yes Theorem 5.13	yes Theorem 5.13	yes Theorem 5.13
$\text{TeSSLa}_{\text{stack}}$	DPT Theorem 5.16	decidable Theorem 5.16	no Theorem 5.15	EXPTIME Theorem 5.18	decidable Theorem 5.16
$\text{TeSSLa}^{f}_{\text{bool}}$	func. NFST Theorem 5.20	EXPSPACE Theorem 5.21	no Theorem 5.22	EXPTIME Theorem 5.23	EXPTIME Theorem 5.24
$\text{TeSSLa}_{\text{bool}+c}$	DTFST Theorem 5.26	EXPSPACE Theorem 5.27	no Theorem 5.28	EXPTIME Theorem 5.29	EXPSPACE Theorem 5.30
$\text{TeSSLa}^{f}_{\text{bool}+c}$	func. NTFST Theorems 5.20, 5.26	undecidable Theorem 5.31	no Theorem 5.32	EXPTIME Theorem 5.33	undecidable Theorem 5.34

Figure 5.11: Shows the results of Chapter 5, showing the equivalent formalism for every considered TeSSLa fragment as well as summing up the complexities for the decision problems.

6 Relation of TeSSLa to Other Stream Languages

Contents

In this chapter, we compare TeSSLa to other stream languages like LOLA, Striver, Lustre and Esterel. For this purpose, we take a look at different types of streams as well as fragments of TeSSLa (TeSSLa^f) and see how they compare to the mentioned languages.

While LOLA and Striver are, like TeSSLa, build to be specification languages on streams, thus, the idea is to specify correctness properties for a given system or statistical evaluations, to get certain measurements for a running program, Lustre and Esterel are created to be programming languages for programming on a stream based model. As such, they are not build up as an optimized formal language with a minimal set of operators, but instead are build to be easily usable for programmers of huge programs. Therefore, the comparison with TeSSLa is a bit different in their case.

In general, the comparison on expressiveness mainly takes a look at five typical features of stream languages which differentiate the languages we consider. Some languages have some features of those, some languages have others. We now briefly introduce those features now and at the end of this section, we provide an overview stating which languages contains which feature.

1. *Explicit Time Handling:* The language is able to handle time explicitly and in a special manner, get the timestamps of the incoming events and do calculations on them.

2. *Non-synchronized Events:* The language is able to handle streams without a common clock. Events can occur at any timestamp without any minimal distance between the timestamps proposed by the environment needed.

3. *Create Events:* The language is able to output events at timestamps at which no input events occurred. These output timestamps are not defined by a given grid, but instead arbitrarily calculated by the specification.

4. *Future References:* While all languages are able to reference past values, not all can reference future values. Future references have the advantage of making the specification for certain properties easier, but when evaluated practically, many values may need to be remembered.

5. *Zeno Streams:* The language is able to handle and produce Zeno behaviour on streams. While this allows the language to handle arbitrarily many events in a given time window, Zeno behaviour in general is not helpful on piece-wise constant streams as considered in this thesis. This is because in a practical evaluation, one would never reach a point beyond the timestamp the events timestamps converge to.

It is important to note that those features are not positive in every aspect. Containing a features may increase the expressiveness, but also makes the languages more complex (as, for example, already seen in Chapter 5). Also, some behaviour like specifying properties about future values or Zenoness may be hard to handle in practical applications.

Before we get to the comparison of different languages, the following section explains and discusses how our notion of expressiveness works and what is different to the standard notion of expressiveness.

6.1 Discussion on Expressiveness of Stream Languages

It is important to discuss the notion of expressiveness we use in this thesis regarding the comparison of the stream languages. At first, it can be noted that all the languages considered in this section, LOLA and its extensions, Lustre, Esterel, Striver and TeSSLa, are Turing complete. This means we can do arbitrary calculations regarding a given input to get a certain final output or in other words, all theses languages can easily simulate a Turing machine, using the right data structures for simulating the tapes and the past reference operators to access older states of the tapes. Therefore, in this thesis, we use a different approach in comparing the expressiveness by not only considering the relation between the input and the final output, but instead the stream transformation functions that can be expressed with each language, which relates more to the stream Turing machine we defined in Definition 2.43 instead of conventional Turing machines. This means, the whole output streams matter and thus even additionally created events and even the time wise order of the events, also containing intermediate values that can only be calculated at some timestamp using future references, are important for a higher expressiveness.

Note that our notion of expressiveness is just a natural extension of the standard notion. As common, we compare the languages that can be expressed with a given formalism as defined in Definition 2.1, but in this case, those languages are stream transformations instead of only relations between input streams and single output values or acceptance criteria, as it is for conventional languages or Turing machines. Therefore, we do not change the definition of expressiveness itself, but we change the types of languages we consider, which allows a more distinct view on stream languages and their expressiveness.

Such a discussion about including the output over time and not just the final output into the notion of expressiveness has already been raised in [Gol00, GSAS04, Weg98, GSW01] regarding the question if interactive Turing machines are more powerful then conventional Turing machines. In these papers, a version of a Turing machine has been developed which also considers an input-output-relation like a stream transformation by outputting a value when a new input value occurs interactively. Those have been called interactive Turing machine (iTM) [Weg98, GSW01] or persistent Turing machine (pTM) [Gol00, GSAS04], respectively. Even though the discussion about the question if an iTM or a pTM is more powerful than a TM is ongoing to the best of our knowledge, it is interesting to note it here, because the concept of interaction is quite similar to our concept of considering the complete output stream regarding expressiveness and not only the final value. We gave a notion of such a Turing machine, which we call a stream Turing machine, in Definition 2.43.

Even though the previously mentioned discussion regarding interaction is ongoing, our notion of expressiveness also makes sense in a practical application in monitoring. If we want to check a correctness property during the execution of a system, it is important that we can notify about a violation as soon as possible. In this manner, for example a language including an operator like the **delay** is better or *more expressive*, because it can set a delay to a timestamp at which a certain timeout runs out and the property would be violated. With a language without such an operator, we would have to wait for the next input event to arrive to recognize that there is a violation of the property and therefore it can not be expressed in a property, that the violation is definitely observed when is occurs. This example shows us that we can even practically express tighter properties when including such a **delay** operator and that our notion of expressiveness makes sense in this setting.

6.2 TeSSLa and LOLA

LOLA is able to specify past and future references using negative or positive numbers as offset. Whereas TeSSLa is only able to specify past references, because for practical applications in runtime verification and log file analyses over a continuous time domain it generally makes sense to restrict specifications to past references because otherwise future values need to be known at an earlier timestamp which requires buffering values until the needed timestamp. Nevertheless, future references may make a difference for engineer who want to specify properties, as they can lead to simplified specifications, even if they are not necessary to express certain properties. This is why we extended TeSSLa to TeSSLa^f in the beginning of this thesis.

In the next section, we will compare TeSSLa and LOLA on discrete streams and show that they behave equally on those streams, under the assumption that LOLA can somehow access and process timestamps. Recall that a discrete stream in this thesis is a stream where the time domain is $\mathbb{N}$ and at every timestamp, there is either an event or no progress (therefore ?). Note that discrete streams can be defined in a different way, for example the time domain can be a different set of values. Our definition has the advantage that a language like LOLA, which has no notion of time normally, is able to calculate the time by counting the events. But this behaviour can be restored if the time domain is not $\mathbb{N}$ by, for example, adding an additional input stream for every LOLA specification, containing the timestamps as values of the events. In the end, this approach is the same as the one we use here, only the timestamps in LOLA are represented in a different way. The comparison regarding expressiveness leads to the same results. Also, note that choosing a time domain where two adjacent values can have different distances, results in strange behaviour. But this holds for

TeSSLa in general, as the **delay** operator would then be able to get input values which results in events at timestamps that are not in the time domain any more. Therefore, we do not consider such time domains.

As extension to TeSSLa, we will also take a look at the relation of TeSSLa^f and LOLA, as it also contains future references using the **next**. Furthermore, we will compare some of the fragments introduced in the previous chapter to fragments of LOLA and see, that there exist quite similar fragments for LOLA.

In the section after the next one, we will also compare LOLA and TeSSLa on arbitrary streams over a continuous time domain and see that TeSSLa is more expressive on such streams.

6.2.1 TeSSLa and LOLA on Discrete Streams

At first, let us consider the past fragment of LOLA, LOLA$_{\text{past}}$, which is restricted to negative offsets only. The following theorem states that the past fragment of LOLA and TeSSLa are equally expressive on discrete streams and the proof is done constructively.

Note that we assume for the direction where we show that LOLA can express the TeSSLa operators, that a $\perp$ value is included in the data domain of every stream to represent that there is no event in the meaning of the streams used as stream model for TeSSLa. Otherwise, a lot of the functionality of the TeSSLa operators could not be represented. For LOLA, we then handle this value as if there is an event with $\perp$ as value. This is only a technical adjustment for the proofs, which does not change the functionality of any operator or influence the expressiveness, as this conversion between the stream models can naturally be done and is quite similar to the synchronization of the streams we already did in the previous chapter to encode the streams in transducer words.

Theorem 6.1 (*LOLA$_{past}$ in the Discrete Setting*)

On discrete streams it holds that LOLA$_{past}$ = TeSSLa.

Proof. We give a constructive proof by delivering a translation scheme from arbitrary LOLA$_{\text{past}}$ formulas to TeSSLa formulas and vice versa.

At first we show how every LOLA$_{\text{past}}$ specification can be transformed into an equivalent TeSSLa specification. Therefore, we show how a single line of LOLA$_{\text{past}}$ can be transformed into one or more lines of TeSSLa.

$\text{LOLA}_{\text{past}}$ has two types of expressions, functions applied to streams as well as the operator for accessing older values. For the function application, we just use a **lift** operator to represent it in TeSSLa. For the offset operator, we mainly use a **last** to access to prior value and a merge for adding the constant value at timestamp 0.

Note that for any $n < -1, n \in \mathbb{Z}$, we can just reformulate a $\text{LOLA}_{\text{past}}$ formula $[n,c]$ such that there are only -1 offsets by building n lines of LOLA expressions, each of the form $[-1,c]$, for a given constant c. Therefore, we only consider expressions of type $[-1,c]$ as offsets for the rest of the proof.

For a $\text{LOLA}_{\text{past}}$ expression

$$s = f(s_1, \ldots, s_n)$$

we can express the same behaviour in TeSSLa with

$$s := \textbf{lift}(f)(s_1, \ldots, s_n)$$

and for an expression

$$s = s'[-1,c]$$

we can express the same behaviour in TeSSLa as

$$s := \mathsf{merge}(\textbf{last}(s', \mathit{clock}), \mathsf{const}_c(\textbf{unit}))$$

where *clock* is either s' if $s \neq s'$ or some other stream not involved in the recursion. This works because all streams are discrete and thus have events at every timestamp.

By using the translation above, we can therefore transform any $\text{LOLA}_{\text{past}}$ formula by flattening it first and then transforming the specification line by line into an equivalent TeSSLa formula.

It remains to show how every TeSSLa specification can be transformed into an equivalent $\text{LOLA}_{\text{past}}$ specification. We use the same approach, flatten it first and transform it line by line. We assume the data domain of each LOLA stream to contain $\perp$ as value. Each TeSSLa operator can be transformed as follows:

- $s := \textbf{nil}$

$$s = s[-1, \perp]$$

- $s := \textbf{unit}$

$$s' = s'[-1,-1]+1$$
$$s = ite(s' = 0, \Box, \bot)$$

- $s := \textbf{time}(s')$

$$s = s[-1,-1]+1$$

- $s := \textbf{lift}(f)(s_1,\ldots,s_n)$

$$s = f(s_1,\ldots,s_n)$$

- $s := \textbf{last}(val, trig)$

$$s = ite(trig \neq \bot, v[-1,\bot], \bot)$$
$$v = ite(val \neq \bot, val, v[-1,\bot])$$

- $s := \textbf{delay}(del, r)$

$$count = count[-1,-1]+1$$
$$s = ite(count = timeout[-1,\infty], \Box, \bot)$$
$$timeout = ite((r \neq \bot \vee s \neq \bot) \wedge del \neq \bot, count + del, timeout[-1,\infty])$$

□

As one can see, the **delay** is simulated by counting the timestamps and remembering the last timeout if no reset occurs. This is only possible because our time domain is discrete with the same distance between every two adjacent values.

What we have not considered now are the future references in LOLA, because also TeSSLa has no future references. But we defined an extension of TeSSLa, TeSSLa^f, containing an operator for

future references called **next**, which makes it more expressive as shown in Theorem 4.28 and Theorem 4.30. So we compare LOLA and TeSSLa^f next, as both contain future references compared to the languages considered in the previous theorem.

Because discrete streams are finite and have an ending after a finite amount of time the **next** can be seen as a pendent to **last** and that it works in the same way as **last** in the discrete setting: Every time an event arrives on the trigger stream, **next** returns the value of the next event on the value stream until the ending is reached. Therefore, it directly matches to the future references of LOLA as we see in the following theorem.

Theorem 6.2 (*LOLA in the Discrete Setting*)

On discrete streams it holds that LOLA = TeSSLa^f.

Proof. We already showed $\text{LOLA}_{\text{past}} = \text{TeSSLa}$. It remains to show that the **next** operator in TeSSLa^f is able to express LOLA specifications of the form $s = s'[+1, c]$ and that LOLA is able to express the **next** operator.

A LOLA specification of the form

$$s = s'[+1, c]$$

can be expressed in TeSSLa^f as follows, where we assume the last timestamp of the stream is known, in the same way it is in the beginning with 0, and call it t:

$$s := \textbf{lift}(ite)(\textbf{time}(clock) = t, c, \textbf{next}(s', clock))$$

Furthermore, for the other direction, LOLA can express a TeSSLa$_f$ formula $s := \textbf{next}(val, trig)$ as follows:

$$s = ite(trig \neq \bot, v[+1, \bot], \bot)$$
$$v = ite(val \neq \bot, val, v[+1, \bot])$$

□

Also, with the same restriction to the usage of **last** and **next** as for -1 and +1 in LOLA, TeSSLa^f can be restricted to $\text{TeSSLa}^f_{\text{eff}}$ such that on each path in the dependency graph the difference of *delayed* and *next* labelled edges is positively bounded. Then it holds that $\text{LOLA}_{\text{eff}} = \text{TeSSLa}^f_{\text{eff}}$.

Corollary 6.3 (*Efficient Fragments on Discrete Streams*)

On discrete streams it holds that $LOLA_{eff} = TeSSLa^{f}_{eff}$.

This follows directly from the fact that each operator can be translated one by one as shown in the previous theorems, without creating paths with infinite future references in the dependency graph if there where non before.

All the previous results apply independently of the allowed data domains, thus it does not matter if arbitrary data domains are allowed or if we restrict the data domains to bounded ones (which is the same as only using boolean valued streams). This follows because every operator can be translated independently of the rest of the formulas and without using the data domain into the other formalism.

Corollary 6.4 (*Bounded Data Domains on Discrete Streams*)

On discrete streams the following statements hold:

- $LOLA^{b}_{past} = TeSSLa_{bool}$ *and*
- $LOLA^{b} = TeSSLa^{f}_{bool}$.

Additionally, we can also show that $\text{LOLA}_{\text{bool}}$ is as expressive as $\text{TeSSLa}_{\text{bool+c}}$, which extends $\text{TeSSLa}_{\text{bool}}$ by expressions of the form $\textbf{lift}(g_v)(\textbf{time}(e), \mathsf{merge}(\textbf{last}(\textbf{time}(e), e), 0))$ for the comparison of timestamps and a constant like the timing constraints in timed automata, on discrete streams. The proof is based on showing that $\text{TeSSLa}_{\text{bool+c}}$ and $\text{TeSSLa}_{\text{bool}}$ as equally expressive on discrete streams. The idea of the proof is that the discrete time domain allows us to count the timestamps if necessary, because there can only be a finite number of events between every two timestamps. This leads to the fact that a **last** on a **time** operator as value stream results always in value $t - 1$ at a timestamp t, as it is always triggered and the prior timestamp is always one less than the current timestamp.

Theorem 6.5 (*$LOLA^{b}_{past}$ and $TeSSLa_{bool+c}$ on Discrete Streams*)

On discrete streams it holds that $LOLA^{b}_{past} = TeSSLa_{bool+c}$.

Proof. From Corollary 6.4 we know that $\text{LOLA}^{b}_{\text{past}} = \text{TeSSLa}_{\text{bool}}$. To now prove $\text{LOLA}_{\text{bool}} = \text{TeSSLa}_{\text{bool+c}}$, we show $\text{TeSSLa}_{\text{bool}} = \text{TeSSLa}_{\text{bool+c}}$ in the discrete setting.

Because every $\text{TeSSLa}_{\text{bool}}$ formula is also a $\text{TeSSLa}_{\text{bool+c}}$ formula, it trivially holds that $\text{TeSSLa}_{\text{bool}} \subseteq \text{TeSSLa}_{\text{bool+c}}$.

To show the other direction, we have to show that the subexpressions added in $\text{TeSSLa}_{\text{bool+c}}$ of the form $\mathbf{lift}(g_v)(\mathbf{time}(e), \mathsf{merge}(\mathbf{last}(\mathbf{time}(e), e), 0))$, which are the only addition in $\text{TeSSLa}_{\text{bool+c}}$ compared to $\text{TeSSLa}_{\text{bool}}$, can also be expressed by expressions of $\text{TeSSLa}_{\text{bool}}$, if the input streams are discrete. In the discrete case, **time** does not really have any impact, since the timestamps are natural numbers and on every such timestamp, an event exists. Because expressions of the given form always have to look like $\mathbf{lift}(g_v)(\mathbf{time}(a), \mathsf{merge}(\mathbf{last}(\mathbf{time}(b), a), 0))$, we know that $\mathbf{last}(\mathbf{time}(b), a)$ is always $\mathbf{time}(a) - 1$. With v being a constant, the remaining expression is $\mathbf{lift}(g_v)(\mathbf{time}(a), \mathsf{merge}(\mathbf{time}(a) - 1, 0))$ which is constant as well, for a given formula statically being either true or false, depending on value chosen for v. Thus, for every v, expressions of the form $\mathbf{lift}(g_v)(\mathbf{time}(e), \mathsf{merge}(\mathbf{last}(\mathbf{time}(e), e), 0))$ can be replaced by either true or false on discrete streams, which are values available in $\text{TeSSLa}_{\text{bool}}$. □

Also, this shows not only that both TeSSLa fragments are equally expressive on discrete streams, but also that they can be transformed into each other without increasing the size of the formula, making the timing fragment $\text{TeSSLa}_{\text{bool+c}}$ the same as $\text{TeSSLa}_{\text{bool}}$ in the discrete setting regarding expressiveness. Additionally, from the previous theorem with the same argumentation, the next statement also follows.

Corollary 6.6 (*LOLA^b and Time on Discrete Streams*)

On discrete streams it holds that $\text{LOLA}^b = \text{TeSSLa}^f_{\text{bool+c}}$.

6.2.2 TeSSLa and LOLA on Continuous Streams

While for TeSSLa and LOLA on discrete streams both have fitting fragments or extensions for those from the other language which have the same expressiveness, this changes on a continuous time domain.

While on a discrete streams as defined in this thesis, one can access the current timestamp by counting the events, this is not possible any more in a non-discrete setting. Naturally, LOLA has no possibility to access the timestamps of the events, while TeSSLa has one with the **time** operator. But for LOLA, one could assume the time to be part of the input, therefore adding an input stream

with the timestamps at every event. This would allow LOLA the same functionality as **time** has in TeSSLa. But naturally, LOLA does not contain such a time operation.

Furthermore, LOLA is only able to reference events that already exist and it has no possibility to act on a timestamp at which no event exists on any input stream. TeSSLa can easily do this using the **delay** operator, which allows TeSSLa to express stream transformations that LOLA can not express in the continuous setting.

Let us first consider $\text{LOLA}_{\text{past}}$ and LOLA. We show in the following theorem that TeSSLa is more expressive than $\text{LOLA}_{\text{past}}$ and that TeSSLa^f is more expressive than LOLA. Even if we assume an input stream with timestamps, as explained before, for LOLA, both still holds because of the **delay** operator.

Theorem 6.7 (*LOLA on Continuous Streams*)

On continuous streams the following statements holds:

- $\mathit{LOLA}_{\mathit{past}}$ *= TeSSLa without* **delay** *and*
- $\mathit{LOLA} = \mathit{TeSSLa}^f$ *without* **delay**.

Proof. First, we can show that TeSSLa is able to express all $\text{LOLA}_{\text{past}}$ operators using exactly the same transformation as for Theorem 6.1 and that TeSSLa^f is able to express all LOLA operators using exactly the same transformation as for Theorem 6.2. We can also use the same transformations for the other direction from both theorems for transforming LOLA and $\text{LOLA}_{\text{past}}$ into TeSSLa^f and TeSSLa, respectively, without considering the **delay** operator. For the **time** operator, we assume that the LOLA specification has a special input stream, containing the timestamps for each event.

Additionally, it is easy to see that $\text{LOLA}_{\text{past}}$ and LOLA can not express a stream transformation which creates events on the output streams without any event on the input stream, which TeSSLa and TeSSLa^f can do in many ways using **delay**. Compared to the discrete setting, this holds because the **delay** operator can not be implemented in LOLA, because the timeout can not be calculated any more by counting the events. □

Considering again the boolean versions of LOLA and TeSSLa, we can state again that both types are equally expressive, even on continuous streams. This follows mainly from the fact that both can only act if an event occurs on any stream, which means the streams are like discrete streams for these fragments.

Theorem 6.8 (*LOLA Fragments on Continuous Streams*)

On continuous streams the following statements holds:

- $LOLA^{b} = TeSSLa^{f}_{bool}$ *and*
- $LOLA^{b}_{past} = TeSSLa_{bool}$.

Proof. The two main differences between LOLA and TeSSLa on continuous streams are, that TeSSLa can naturally access the timestamps and that it can act at timestamps without any input events being present. As already shown in the previous section, on discrete streams both statements hold. But being restricted to boolean streams erases the possibility to access timestamps in TeSSLa in both fragments and both also do not include the **delay** operator. Therefore, in this case the continuous streams are also like discrete streams in both cases and therefore, as shown before, both equalities hold. □

Even though the boolean fragments are still equivalent on continuous streams in terms of expressiveness, this does not hold for the timed fragments of TeSSLa and the boolean fragments of LOLA any more. Both timed fragments of TeSSLa are allowed to access and compare timestamps and can therefore make statements considering the distance of the events, which LOLA with only boolean streams can not do. Therefore, the following statement holds.

Corollary 6.9 (*Timed Fragments and LOLA on Continuous Streams*)

On continuous streams it holds that $LOLA^{b}_{past} \subsetneq TeSSLa_{bool+c}$.

Lastly, we state that the efficient fragments are different regarding expressiveness on continuous streams. This directly follows from the fact that the **delay** operator can now create new events which LOLA is not capable of.

Corollary 6.10 (*Efficient Fragments on Continuous Streams*)

On discrete streams it holds that $LOLA_{\text{eff}} \subsetneq TeSSLa^{f}_{\text{eff}}$.

6.2.3 TeSSLa and LOLA2

Besides the original version of LOLA from [DSS+05], there is a second version defined for network monitoring in [FFST16] which is called LOLA 2.0, or LOLA2 for short. LOLA2 adds parametrized stream templates to LOLA which can be instantiated such that, for example, streams can be created for every id of some object. An instantiation is invoked by events on a certain stream as well as the generation as outputs, which is invoked by events on a second stream, called the extension stream. Lastly, there is a termination stream in each stream template to kill an instance.

The ability to create streams is something which is neither present in LOLA nor in TeSSLa. Nevertheless, even though the stream templates add simpler ways to express certain properties, they do not add additional expressiveness to LOLA and therefore, LOLA2 is the same as LOLA in this manner. This holds, because LOLA can handle arbitrary data types and therefore even maps. Where an instance is created in LOLA2, one can put the parameter of the instance, for example the id, into a map and take care of such created streams this way. The same can obviously also be done in TeSSLa in the same way. Therefore, all results regarding LOLA and TeSSLa in the previous section can also be applied to LOLA2.

Note that this result follows from how we defined expressiveness and therefore, how we compare languages. If one would take the feature of being able to create streams into account, this would be a different dimension. But as it does not change the input / output relation, it does not change the expressiveness in the way we defined it.

6.2.4 TeSSLa and RTLola

Additional to LOLA2, another extension has been developed for LOLA, called RTLola [FFST19], which is an extension of LOLA2. While LOLA2 was only able to work on discrete streams or at least only able to act where events on the input streams existed, the purpose of RTLola is being naturally defined on continuous streams, and therefore even able to handle non-synchronized events, like TeSSLa.

In RTLola, the handling of the continuous time domain and the non-synchronized events is done by the possibility of choosing a frequency at which something is done, like every 0.1 seconds or so. Then, the specification reacts at every such time instance as if there would be an event on the input streams. Further more, RTLola contains two additional key features compared to LOLA: The stream templates already known from LOLA2 and a possibility to define sliding windows. A

sliding window in RTLola is a stream $s[r, f]$ where r is a real number defining the length of the window and f is the aggregation function for the data of the events currently in the window.

Compared to TeSSLa, RTLola has still neither a possibility to access the timestamps of events and thus can not execute calculations on those (which can of course be simulated again with an additional input stream), nor is it able to create events at arbitrary positions which TeSSLa can do with **delay**. It can create events in a restricted manner, by using a frequency and output events at positions, where no input stream had an event. But this frequency is given in advance and therefore, it can not create events between two time units, like TeSSLa is still able with the **delay** operator. This leads to the fact that RTLola is, for example, not able to handle or create Zeno streams, but TeSSLa is. In the end, this frequency addition in RTLola is also nothing more than an additional input stream, which could also be done in standard LOLA. It would also be able to operate on each of these frequency events if such an input stream exists, like the one containing the timestamps.

Additionally, even though it is claimed in [FFST19] that TeSSLa can neither handle parametrization, nor sliding windows, this is not true for the standard iteration of TeSSLa from [CHL+18]. Parametrization can be handled as explained in the previous section about LOLA2 using maps or sets and an example for sliding windows in TeSSLa has been given in [LSS+19].

As TeSSLa^f is able to express all types of stream transformations, it follows that RTLola is less expressive than TeSSLa^f and incomparable to TeSSLa, which follows from the future references available in RTLola.

6.3 Striver

Striver [GS18, GS20] is a specification language over the same stream model TeSSLa is defined over, therefore it can handle continuous time domains and an asynchronous arrival of events naturally. Additionally, as TeSSLa, Striver has an explicit notion of time and is able to output events at positions where no input events exist, which was not possible in LOLA and only possible in a very restricted way in RTLola.

Striver was first introduced in [GS18], where the authors defined a version similar to TeSSLa, also without future references, but forbid Zeno streams completely (and lost some additional expressiveness with it). We will consider the version of Striver from the follow-up paper [GS20]. Compared to the previous version, this version of Striver does contain future references over streams with a finite number of events and the paper also contains an extension for its delay operator which

could, in general, allow Zeno streams. But still, while this delay would be able to express Zeno streams, the semantics are defined in a way such that they can not handle Zeno streams and therefore, Striver can still not express them. This is done on purpose, because one of the design goals of Striver was to get as much expressiveness as possible without being able to generate or handle Zeno streams. But, compared to the version of Striver from [GS18], the one in [GS20] is able to express all TeSSLa operators when Zeno streams are not considered. This addition makes the language much more similar to TeSSLa, considering expressiveness.

Before we get to the comparison of TeSSLa and Striver, we give a formal introduction to Striver as it is defined in [GS20]. There are two types of expressions, first, there are ticking expressions α over a time domain $\mathbb{T}$ representing the sets of timestamps when a given specification may have events, given over the following syntax, where $c \in \mathbb{T}$ and $\varepsilon \in \{t \in \mathbb{T} \mid t > 0\}$ are constants, v is a stream variable and $\cup$ is used as union of two sets of timestamps:

$$\alpha ::= \{c\} \mid v.\mathsf{ticks} \mid \mathsf{delay}(\varepsilon, v) \mid \alpha \cup \alpha$$

Second, there are offset expressions τ which allow accessing timestamps of older or newer events on streams. The syntax of those is given as follows, where v is a stream variable and T is the current timestamp:

$$\tau ::= T \mid \tau_v \qquad \tau_v ::= v <= \tau \mid v << \tau \mid v >= \tau \mid v >> \tau$$

Lastly, there are value expressions E, where d is a constant value of an arbitrary data domain, v is a stream variable and f a function mapping multiple input streams to an output stream:

$$E ::= d \mid v(\tau_v) \mid f(E, \ldots, E) \mid \tau$$

Next, we provide the semantics for the Striver operators by using the function $[\![.]\!]_\sigma$ for an expression σ. We begin with timing expressions, where $dom(\sigma_v)$ returns the set of timestamps at which v has an event in the expression σ, t is the current timestamp and $sign(n)$ returns the sign of n:

$$\begin{aligned}
[\![\{c\}]\!]_\sigma &= \{c\} \\
[\![v.\mathsf{ticks}]\!]_\sigma &= dom(\sigma_v) \\
[\![\alpha_1 \cup \cdots \cup \alpha_n]\!]_\sigma &= [\![\alpha_1]\!]_\sigma \cup \ldots [\![\alpha_n]\!]_\sigma \\
[\![\mathsf{delay}(\varepsilon, v)]\!]_\sigma &= \{t' \mid \exists t \in dom(\sigma_v) : t + \sigma_v(t) = t' \wedge |\sigma_v(t)| \geq |\varepsilon| \wedge \\
&\qquad sign(\sigma_v(t)) = sign(\varepsilon) \wedge dom(\sigma_v) \cap (t, t') = dom(\sigma_v)(t', t) = \emptyset\}
\end{aligned}$$

It is important to note that the delay of Striver works differently than the one of TeSSLa. It has no reset, but instead the only input stream resets and sets a new value. Additionally, it takes an ε which sets the minimum value an incoming timeout value needs to have to be set as a timeout, otherwise it is ignored. This stops Zeno behaviour from working.

At first, we present the semantics for the offset expressions in an informal way, to deliver more intuition, because the formal definition is quite lengthy while the operations can be explained easily in an informal way. The operations $x << e$ and $x <= e$ return for a given timestamp t the highest timestamp of the events of x which occurred before the last event on e before t. Thereby, the difference is that $x <= e$ may return the timestamp of the event on e, while $x << e$ must return a timestamp smaller than the last one on e. The same holds for the future variants $x >> e$ and $x >= e$ which return the lowest timestamp at which an event occurred on x after the next event on e after timestamp t.

Formally, the offset expressions are defined as follows, where we write $\max_t$ for $\max\{t' < t \mid e(t') \neq \bot\}$ and $\min_t$ for $\min\{t' > t \mid e(t') \neq \bot\}$:

$$[\![x << e]\!]_\sigma(t) = \begin{cases} d & \text{if } \exists t' < \max_t : x(t') = d \wedge \forall t' < t'' < \max_t : x(t'') = \bot \\ \bot & \text{otherwise} \end{cases}$$

$$[\![x <= e]\!]_\sigma(t) = \begin{cases} d & \text{if } \exists t' < \max_t : x(t') = d \wedge \forall t' < t'' \leq < \max_t : x(t'') = \bot \\ \bot & \text{otherwise} \end{cases}$$

$$[\![x >> e]\!]_\sigma(t) = \begin{cases} d & \text{if } \exists t' > \min_t : x(t') = d \wedge \forall t' > t'' > \min_t : x(t'') = \bot \\ \bot & \text{otherwise} \end{cases}$$

$$[\![x >= e]\!]_\sigma(t) = \begin{cases} d & \text{if } \exists t' > \min_t : x(t') = d \wedge \forall t' > t'' \geq \min_t : x(t'') = \bot \\ \bot & \text{otherwise} \end{cases}$$

Because they do not add something to the expressiveness, we left out the cases where the stream ends.

Finally, the value expression $f(E,\ldots,E)$ is defined as follows:

$$[\![f(E,\ldots,E)]\!]_\sigma(t) = f'([\![E_1]\!]_\sigma(t),\ldots,[\![E_n]\!]_\sigma(t))$$

where f' calculates the output value for the concrete input values at a timestamp t.

While Striver handles time in a more explicit manner than TeSSLa, its handling is also different, but does not lead to more expressiveness. We will show in the following that both languages are mostly equivalent in terms of expressiveness minus the handling of Zeno streams by giving a construction method for transforming Striver specifications into TeSSLa.

We now give a translation from every operator of Striver into TeSSLa^f, which we will use afterwards to present expressiveness results of Striver regarding TeSSLa and its variants.

Striver has three types of expressions: ticking expressions, offset expressions and value expressions. At first, we show how the ticking expressions can be expressed in TeSSLa^f. A Striver ticking expression $s := \{c\}$ can be expressed in TeSSLa^f as

$$s := \mathbf{time}(\mathbf{delay}(\mathsf{const}_c(\mathbf{unit}), \mathbf{unit}))$$

A ticking expression $s := s'.\mathsf{ticks}$ can be expressed in TeSSLa^f as

$$s := \mathbf{time}(s')$$

A ticking expression $s := s_1 \cup \cdots \cup s_n$ can be expressed in TeSSLa^f as

$$s := \mathbf{time}(\mathsf{merge}(s_1, \ldots, \mathsf{merge}(s_{n-1}, s_n)))$$

A Striver ticking expression $s := \mathsf{delay}(\varepsilon, s')$ with $\varepsilon > 0$ can be expressed in TeSSLa^f as

$$s := \mathbf{delay}(\mathsf{filter}(s', s' \geq \varepsilon), s')$$

The last variant of Strivers delay with $\varepsilon < 0$ is special, as it sets events in the past. For TeSSLa^f to simulate this behaviour, we have to foresee when an event would be set by using the **next** and output an event now. A Striver ticking expression $s := \mathsf{delay}(\varepsilon, s')$ with $\varepsilon < 0$ can be expressed in TeSSLa^f as

$$\begin{aligned} trig &:= \mathsf{merge}(s, \mathsf{merge}(s', \mathbf{unit}) \\ nextValue &:= \mathbf{next}(s', trig) \\ s &:= \mathbf{delay}(\mathsf{filter}(\mathbf{time}(trig) + \mathbf{time}(nextValue) + nextValue, nextValue \leq \varepsilon), trig) \end{aligned}$$

Next, we show how to convert Striver offset expressions into TeSSLa^f. An expression $s := x << e$

can be expressed in TeSSLa^f as

$$s := \mathbf{last}(e, x)$$

An expression $s := x <= e$ can be expressed in TeSSLa^f as

$$s := \mathsf{merge}(e, \mathbf{last}(e, x))$$

An expression $s := x >> e$ can be expressed in TeSSLa^f as

$$s := \mathbf{next}(e, x)$$

An expression $s := x >= e$ can be expressed in TeSSLa^f as

$$s := \mathsf{merge}(e, \mathbf{next}(e, x))$$

Finally, the function application to streams works in the same way as in TeSSLa^f, just more implicitly, such that a Striver expression $f(s_1, \ldots, s_n)$ can be expressed in TeSSLa^f as

$$\mathbf{lift}(f)(s_1, \ldots, s_n)$$

Using the translation given above, we can now make the following statements about the relation between Striver and TeSSLa.

Theorem 6.11 (*Expressiveness of Striver*)

The following two statements hold:

- *TeSSLa without* **delay** $\subset$ *Striver*$_{past}$ $\subset$ *TeSSLa and*
- *TeSSLa and Striver are incomparable.*

Proof. Using the previously given translation, we know that Striver$_{\text{past}} \subset$ TeSSLa holds, because every Striver$_{\text{past}}$ operator can be expressed in TeSSLa and TeSSLa can also create Zeno streams using **delay**, while Striver$_{\text{past}}$ can not. For TeSSLa without **delay** $\subset$ Striver$_{\text{past}}$, this follows from the translation given in [GS20] from every TeSSLa operator into Striver code. Because Striver$_{\text{past}}$ has a delay operator, it is more powerful, because TeSSLa without **delay** can not create events at positions where the input had non.

If we now add back future references to Striver, it can express properties TeSSLa can not express, even though future references are only allowed on streams with a finite number of events [GS20]. But still, it can not handle Zeno streams, which TeSSLa can. Therefore, both languages are incomparable. □

As stated in [GS20], the delay originally included in Striver can not only not express Zeno streams, but is also less expressive than the **delay** operator of TeSSLa, even when not considering Zeno streams. Because of this, a variant of its delay operator was added in [GS20] to allow to get the expressiveness of the **delay** of TeSSLa when not considering Zeno streams and to be able to convert all TeSSLa operators into Striver in this setting. This delay operator, which we call delay' now takes three parameters instead of two, where the ε is split into two. One is the sign, and the other is a function which takes the incoming delay and states if it is a legal delay value or if it is ignored. This is an extension of simply using the ε, as it allows a broader variety in which delay are allowed and which not instead of just setting a minimum size of the delay value.

This new delay operator can be expressed for an expression $s := \mathsf{delay}'(sgn, f, s')$ for $sgn > 0$ in TeSSLa as follows:

$$s := \mathbf{delay}(\mathsf{filter}(s', \mathbf{lift}(f)(s')), s')$$

The other direction with $sgn < 0$ can be expressed in TeSSLa^f as

$$\begin{aligned} trig &:= \mathsf{merge}(s, \mathsf{merge}(s', \mathbf{unit}) \\ nextValue &:= \mathbf{next}(s', trig) \\ s &:= \mathbf{delay}(\mathsf{filter}(\mathbf{time}(trig) + \mathbf{time}(nextValue) + nextValue, \mathbf{lift}(f)(nextValue)), trig) \end{aligned}$$

The major change in delay' is that there is not an ε which only allows delays of a certain height, but instead now has a generic function f which calculates if a delay to be set is viable or not. This is represented in TeSSLa by just using a **lift** for the application of f and as a condition of a filter to ignore events with a value which should not be allowed as a timeout value.

Considering this new version of delay in Striver and the translation into TeSSLa^f as presented before, we can now prove the following two statements. The prime added to the Striver fragments indicates the inclusion of the delay'.

Theorem 6.12 (*Expressiveness of Striver with delay'*)

The following two statements hold:

- *Striver$'_{past}$ $\subset$ TeSSLa,*
- *Striver$'$ $\subset$ TeSSLa^f and*
- *TeSSLa and Striver$'$ are incomparable.*

Proof. Using the translations presented before, we can transform every Striver$'_{\text{past}}$ specification into TeSSLa, because Striver$'_{\text{past}}$ does not have any future operators. Even though, it is shown in [GS20] how every TeSSLa operator can be encoded into Striver$'_{\text{past}}$ specifications if we do not consider Zeno streams, using the delay$'$ to encode TeSSLa's **delay**. TeSSLa is still able to handle Zeno streams, which this Striver fragment can not. Therefore it follows that Striver$'_{\text{past}}$ $\subset$ TeSSLa.

As Striver$'$ is an extension of Striver$'_{\text{past}}$ with future references, it can express properties TeSSLa can not express. But still, it is not able to handle Zeno streams, therefore TeSSLa and Striver$'$ are incomparable.

Because the future references in Striver$'$ are only allowed for a underlying stream model with streams with a finite number of events and it can not handle Zeno streams as well as the new delay$'$ operator can be expressed in TeSSLa^f as shown in the translation above, TeSSLa^f is more powerful than Striver$'$. □

6.4 Lustre

Lustre [CPHP87, HCRP91] is a programming language for reactive systems operating on a synchronized stream model over a continuous time domain. All streams in a Lustre specification share a common clock and at each tick, every stream has either an event with a value or no event.

Like a TeSSLa specification, a Lustre specification consists of multiple equations over variables (other equations references), constants and functions which are automatically lifted to streams like typical arithmetic functions or an if-then-else. Additionally, there are two sequence operators. First, the operator pre(X) refers at position i of the streams to the value from position $i-1$ or nil if $i = 0$, thus allows access to past values. The second operator is the $X \to Y$, which takes two streams X and Y and returns a stream with the first event on X and all other events on Y. More

formally, the two sequence operators work as follows, where $X = x_0 x_1 \ldots$ and $Y = y_0 y_1 \ldots$:

$$\text{pre}(X) = \text{nil}\, x_0 x_1 \ldots$$
$$X \to Y = x_0 y_1 y_2 \ldots$$

Lustre normally also includes the when and the current operators. The when just represents a filtering function and can easily be represented by such a function application. The current is included to fill places in a stream where no events exist, for example, when events are filtered out. Function applications to Nil are not allowed in Lustre and therefore, filling event gaps would be hard without current. But other than that, it does not add to expressiveness as in general, it can be expressed using pre, function application and recursion when function applications would be allowed in this setting. Therefore, we do not consider it any further in this thesis.

The following example shows an easy Lustre specification.

Example 6.13 (*Lustre Specification*)

Consider the following equation X of the form

$$X = 0 \to \text{pre}(X) + 1$$

This specification creates a sequence of naturals, starting with 0.

Lustre does not have any notion of time, besides a common clock which synchronizes the events of all streams. But time can not be accessed directly nor do the streams contain any explicit notion of timestamps. Of course, to solve this issue, one could assume a stream containing the timestamps for each synchronized event is always part of the input. Still, Lustre does not contain operators to manipulate timestamps or add events in any way. This leads to the same restrictions or workarounds needed as explained for LOLA and it leads to the problem that Lustre is not able to output events at timestamps where no input events occurred. But if we use, as explained for LOLA and as we do in this thesis, the natural numbers as time domain, Lustre can also calculate the current timestamp by counting events.

Being designed as a language to program system behaviour, Lustre does contain various constructs to write for example subprograms which can also be instantiated, called a *node*. We will not consider those any further in this thesis, as they do not add to expressiveness, because those can also be interpreted as a stream or be remembered by appropriate data structures like a set or a map. We have already considered this behaviour for LOLA2.

6.4.1 Comparing Lustre to TeSSLa

In general, we can as for LOLA, consider a discrete stream model again. Because Lustre may work over a continuous time domain, it does not have any way to really interact with those timestamps, which makes the basic stream model equivalent to one over discrete streams. Therefore, we can handle a discrete and a continuous stream model for Lustre in the same way as for LOLA.

In the following theorem, we will show that Lustre and Lustre over bounded data domains, which we call Lustre^b, are equal to TeSSLa fragments in the same way as $\text{LOLA}_{\text{past}}$ is, by showing that each Lustre operator can be transformed into a $\text{LOLA}_{\text{past}}$ formula and vice versa.

Theorem 6.14 (*Lustre and TeSSLa*)

It holds that Lustre = TeSSLa without **delay** *and Lustre^b = TeSSLa$_{bool}$ on discrete and continuous streams.*

Proof. We will show Lustre = TeSSLa without **delay** by showing Lustre = $\text{LOLA}_{\text{past}}$. We will do this by transforming each operator of the languages into a specification of the other language.

Let us first start by showing Lustre $\subseteq$ $\text{LOLA}_{\text{past}}$. We can transform a Lustre specification

$$s := \text{pre}(s')$$

into a LOLA specification

$$s := s'[-1, \text{nil}]$$

and a Lustre specification

$$s := x \to y$$

into a LOLA specification

$$s' := s'[-1,-1] + 1$$
$$s := \text{if } s' = 0 \text{ then } x \text{ else } y$$

Functions can be applied to streams in LOLA in the same manner as in Lustre. The other direction, $\text{LOLA}_{\text{past}} \subseteq$ Lustre is shown accordingly. Because functions can be handled in both languages in the same way, we only need to consider the offset operator of LOLA. We can transform a $\text{LOLA}_{\text{past}}$ specification

$$s := s'[-1, c]$$

into a Lustre specification

$$s := c \rightarrow \text{pre}(s')$$

Because each operator can be transformed without using the data domains of the streams, also $\text{Lustre}^b = \text{TeSSLa}_{\text{bool}}$ holds. □

As the Lustre operators work exactly like the $\text{LOLA}_{\text{past}}$ ones, the above result holds on both types of streams.

6.5 Esterel

Compared to Lustre, Esterel [Ber92, Ber99, Ber00, Ber04] works also on a synchronized stream model with a common clock over all streams, which is explicitly an input stream called *tick*. This stream has no values but events which tell us when every other stream has an event. As Lustre, Esterel has no notion of time, but also, at least its implementation only allows integer streams and neither more complex data types, nor even real numbers, which comes from the finite-state approach of Esterel. Even though, in theory, the Esterel language is capable of handling real numbers, therefore, we still assume that the time of the events could be inputted as a special input stream with a continuous time domain, as described for Lustre. An operator for accessing past values, also called pre has in Lustre, as only been added recently to Esterel in version 5.21 [Ber04].

Additionally, Esterel is an imperative programming language and is as such not programmed in an equational way as Lustre is, which is quite similar to TeSSLas way of programming. Therefore, comparing Esterel to TeSSLa can not so easily be done formally because Esterel has a lot of primitive statements which are not at all directly connected to the primitive operators of TeSSLa. Because of this, we will compare Esterel in a more feature-focussed way to TeSSLa and see, which features both languages have and which not, such that we find out what can not be expressed by each language in certain cases which the other language may be capable of.

6.5.1 Comparing Esterel to TeSSLa

In this section, we will take a closer look at the features of Esterel. As noted before, we will compare Esterel with TeSSLa more in a concept focussed way and less in a mathematical way,

because Esterel does not really have a feasible set of core operators which we can compare the ones of TeSSLa to. In [CPHP87] it is said that the difference between Lustre and Esterel is only the programming style, which is declarative for Lustre and imperative for Esterel. By now, both teams collaborate and both languages even share intermediate code and compilation tools [Ber04]. Therefore, both languages, Lustre and Esterel are very similar and share the same features, as we see in the following paragraphs.

Time As mentioned before, Esterel is not capable of handling time explicitly. As Lustre, it can only be considered by adding an additional input stream containing the timestamps of each event.

While on discrete streams this is enough, but as soon as the time domain is continuous, Esterel also lacks the ability of outputting events at timestamps, where no input events exist and therefore misses, for example, the possibility of calculating sliding windows or setting deadline for computation.

Temporal References As Lustre, Esterel has also past references which it implements by using variables (and pre since version 5.21), but no future references. While we extended TeSSLa to TeSSLa^f by adding the **next** operator for future references, such an extension has not yet been done for Esterel.

In the end, Esterel is the same as Lustre and the results for Lustre also hold for Esterel. A major difference lies in its implementation, which is focussed on keeping the program in a finite state manner for better hardware implementations.

6.6 Conclusion

In this chapter, we compared various other stream languages to TeSSLa and its fragments by expressiveness as well as general features of the languages.

Figure 6.1 gives an overview over the comparison of TeSSLa to stream languages which are naturally defined over discrete streams, like LOLA, Lustre and Esterel, while the stream model is restricted to discrete streams. We see that TeSSLa matches LOLA on this kind of streams and Lustre and Esterel are as expressive as LOLA and TeSSLa without future references. This also

shows that the **delay** operator does not add any expressiveness to TeSSLa on this kind of streams, contrary to what has been shown in Chapter 4 for arbitrary streams.

Figure 6.2 sums up the results on arbitrary (continuous) streams. In this case, we included all languages, those which are naturally defined on discrete streams and showed what those still can do on continuous streams and those naturally defined on continuous streams like Striver and RTLola. Additionally, we included the results from Chapter 4 and Chapter 5. It can be seen how the relations in regards to expressiveness change, because an operator like the **delay** now adds additional expressiveness to a language and therefore, LOLA, Lustre and Esterel can not express such properties any more. As Striver has an operator similar to the **delay** (which is also called delay, but works a bit different), it can still express such properties. It remains to show if RTLola is equals to LOLA and therefore TeSSLa^f without **delay** operator in terms of expressiveness if we assume the given frequency as an input stream, which is indicated by the dashed line.

Lastly, the table in Figure 6.3 presents a more abstract overview over the features each language has, that are interesting for a language operating on streams. All languages have past references and can lift arbitrary functions for calculations at a given time instant, therefore those features are not listed here, but instead the ones where the languages differ. The tilde indicates that the language only has this feature in a restricted and less powerful way, but that it is still there in some form.

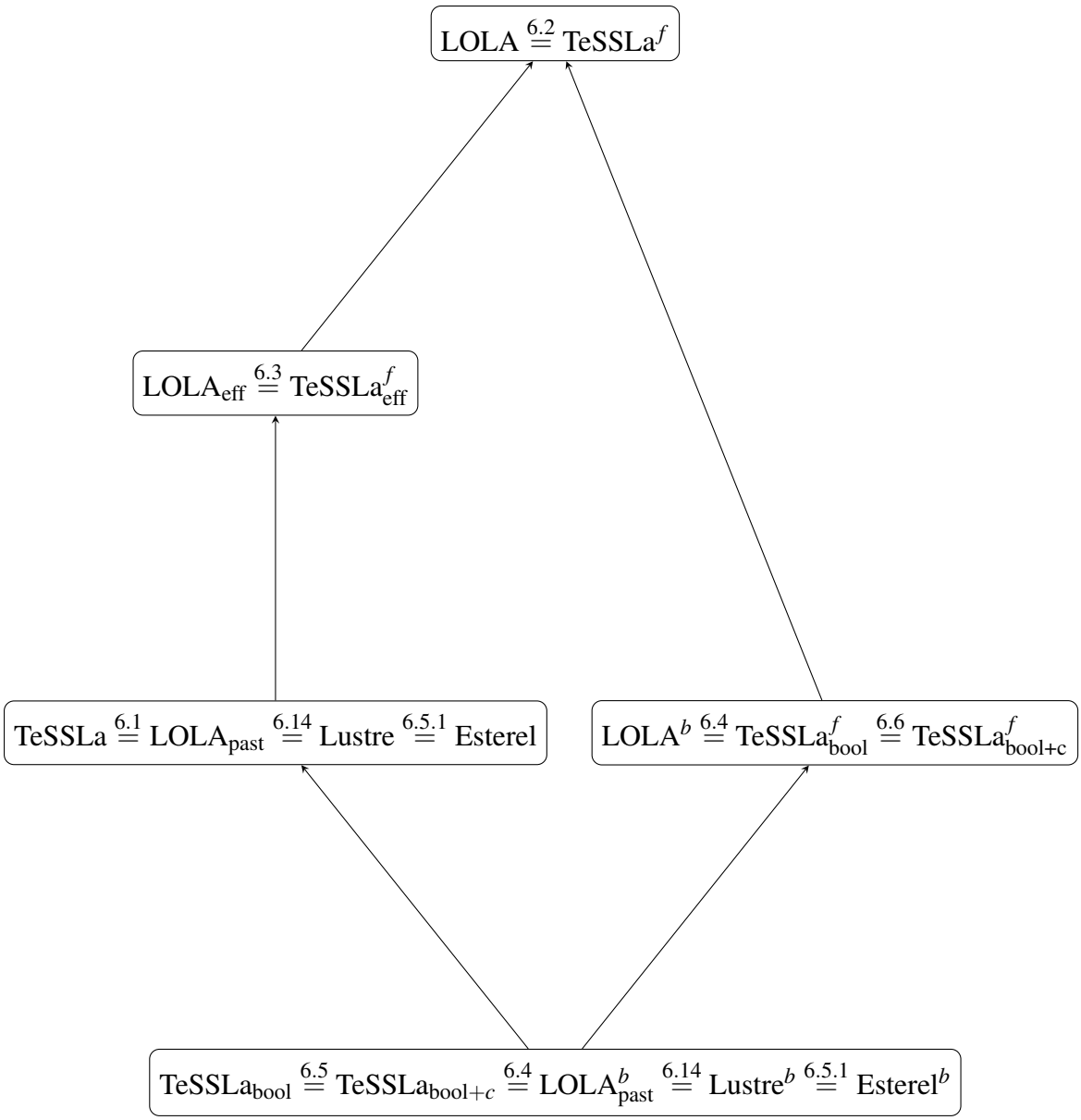


Figure 6.1: Shows the relation of different stream languages in the case of discrete streams. LOLA2 has been left out for clarity, but it and its fragments are equal to LOLA and its corresponding fragments in terms of expressiveness and can be placed accordingly. The arrows indicate expressiveness, the elements in a node at the start of an arrow are strictly less expressive than the elements at its end. Elements between whose nodes no path exist are incomparable.

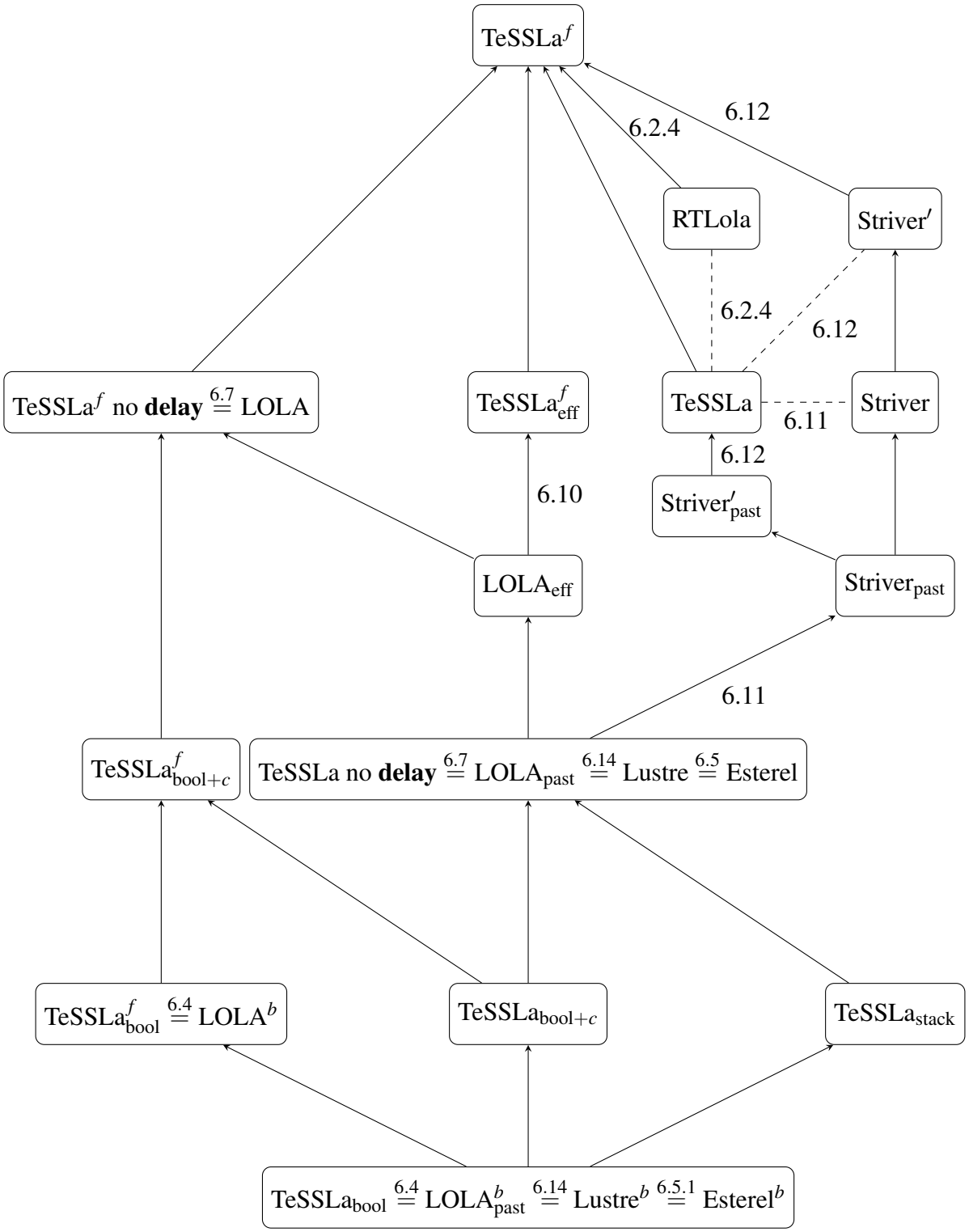


Figure 6.2: Shows the relation of different stream languages in the case of continuous streams. LOLA2 has been left out for clarity, but it and its fragments are equal to LOLA and its corresponding fragments in terms of expressiveness and can be placed accordingly. The arrows indicate expressiveness, the elements in a node at the start of an arrow are strictly less expressive than the elements at its end. If no path exists between two nodes, they are incomparable. We added some dashed arrows indicating the same just to be able to add the theorem or section stating this.

Language	Explicit Time	Non-synch. Events	Create Events	Future Ref.	Zeno streams
LOLA	×	×	×	✓	×
LOLA2	×	×	×	✓	×
RTLola	×	✓	~	✓	×
Striver	✓	✓	✓	~	×
Lustre	×	×	×	×	×
Esterel	×	×	×	×	×
TeSSLa	✓	✓	✓	×	✓
TeSSLa^f	✓	✓	✓	✓	✓

Figure 6.3: Provides an overview over the different features of the various considered stream languages. While the cross indicates that a feature is absent, the checkmark marks that a feature is present. The tilde indicates that the concept is there partially, it does miss some features other languages have.

7 Conclusion and Future Work

This chapter summarizes the contents and results of this thesis and provides an overview over possible extensions as well as open questions remaining.

7.1 Summary

In this thesis, we analyzed TeSSLa in various aspects. First, we introduced a new operator for future references, called **next** and named TeSSLa with **next** TeSSLa^f. After that, we considered different versions of TeSSLa with or without **delay** as well as with or without **next** and found out that both operators add something unique to the expressiveness of TeSSLa and that distinct types of stream transformations can not be expressed without one of those operators. Additionally, we defined the well-formed fragments (syntactic restrictions, whose formulas fixed-point is unique) of TeSSLa and TeSSLa^f. Thereby, we found out that the **next** has to be restricted heavily in a syntactic way, because it is not completely dual to **last**, as there is no finite stream ending in general.

In Chapter 5, we considered many fragments over finite data domains, their relation to each other as well as the complexities for different decision problems for those fragments. Additionally, we have shown how those fragments relate to transducers and gave a constructive translation from the TeSSLa code of each fragment into the corresponding transducer and back. The results were that most of these fragments have decidable decision problems, even if a possibly unbounded stack is added or real-time constraints are considered. Besides the typical decision problems like emptiness and equivalence, we also considered two memory related decision problems. One considering a compositional evaluation strategy for a TeSSLa formula and if it can be evaluated with this strategy using only a bounded amount of memory and one considering the optimal memory usage and the question, if there exists are Turing machine semantically equivalent to the given formula, such that only a bounded amount of memory is needed for the evaluation. While those decision problems are trivial if there are only bounded data structures, we were able to show that even with an unbounded

stack or real-time constraints, it is still decidable if a concrete formula would need only a bounded amount of memory for every input under a given evaluation strategy, which could be of special interest when considering hardware implementations with a small amount of available memory. A graph has been created to give an overview over all fragments as well as the corresponding transducers and a table that contains all complexity results regarding those fragments.

Lastly in Chapter 6, we compared various stream languages to TeSSLa and to each other to gain a distinct separation of the languages and their fragments as well as the concepts used in each language. Even though most of the languages and fragments considered there are Turing complete, we developed a way to distinguish the expressiveness by considering the set of stream transformations expressible by each languages. This also takes into account the development of the output stream over time and not only the final results, which leads to a clear distinction of languages like LOLA, Striver, Lustre, Esterel and TeSSLa. As such a way to compare those languages has not been considered before and mostly stream languages have been compared informally, there is still work to do in comparing other languages formally to each other. Again, this resulted in a graph containing all the languages and the considered fragments as well as a table comparing the languages conceptually regarding five key features of stream languages.

7.2 Future Work

There are different directions in which this thesis can be extended. First and foremost, TeSSLa is designed to be able to handle also streams with continuously changing values, like a sinus curve. But in this thesis, we only consider piece-wise constant streams. Therefore, allowing even streams like a sinus curve would require one to also add liftable functions like an integral function and so on. While many results regarding different variants of TeSSLa have been considered in this thesis, it is unclear whether those results carry over to non piece-wise constant streams and what results can be obtained considering such streams. This would be an interesting direction, because TeSSLa on such streams would allow us to directly make calculations without having to discretize the streams beforehand.

Even though in Chapter 5 many fragments have been considered, the list is definitely not exhaustive. Therefore, looking for more interesting fragments of TeSSLa or even slight restrictions to one of its operators, without completely removing it, is something which may lead to interesting results. Additionally, one could look at other evaluation strategies than just a compositional and an

optimal one, like a parallel evaluation strategy and also could consider the runtime of the evaluation and not just the memory usage.

This also leads to the next field of possible future research regarding TeSSLa. This thesis does answer various theoretical questions, but it does not offer a more practical view. While some work has been done on software and hardware implementations for TeSSLa, there are many questions unanswered. As TeSSLa specifications are often supposed to handle huge amounts of data per second, one could ask if there are feasible, parallel evaluation methods for a TeSSLa specification, to enhance the throughput of data evaluated. While one could consider hardware implementations for this matter, like on an FPGA, also parallel software implementations are of interest.

As already mentioned briefly in the introduction, work has been done on how abstractions of streams can be considered in TeSSLa in [LSS+19]. There, only the possibilities of having timestamps with completely unknown behaviour or events with an unknown value are taken into account. A possibility to extend this would be uncertainty regarding the timestamps, where, for example, every event is only known to be occurred in a certain time frame, but the concrete timestamps is unknown.

Lastly, following the method for comparing stream languages used in Chapter 6, it would be interesting to compare more stream languages and their fragments to each other and categorize existing and new stream languages by their features and expressiveness as well as their properties regarding decision problems. This would lead to an overview to see which features do bring certain advantages and what is the cost for those advantages.

List of Figures

Bibliography

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.

[ACM02] Eugene Asarin, Paul Caspi, and Oded Maler. Timed regular expressions. *J. ACM*, 49(2):172–206, 2002.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *TCS*, 126(2):183–235, 1994.

[AFH91] Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. The benefits of relaxing punctuality. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, pages 139–152, 1991.

[AFR16] Rajeev Alur, Dana Fisman, and Mukund Raghothaman. Regular programming for quantitative properties of data streams. In *ESOP*, pages 15–40. Springer, 2016.

[AH90] Rajeev Alur and Thomas A. Henzinger. Real-time logics: Complexity and expressiveness. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90)*, pages 390–401. IEEE Computer Society, 1990.

[AH92] Rajeev Alur and Thomas A. Henzinger. Back to the future: Towards a theory of timed regular languages. In *IEEE FOCS*, pages 177–186, 1992.

[BB79] Jean Berstel and Luc Boasson. Transductions and context-free languages. *Ed. Teubner*, 1979.

[Ber92] Gérard Berry. The semantics of pure esterel. In Manfred Broy, editor, *Program Design Calculi, Proceedings of the NATO Advanced Study Institute on Program Design Calculi*, volume 118 of *NATO ASI Series*, pages 361–409. Springer, 1992.

[Ber99] Gérard Berry. The constructive semantics of pure esterel, draft version 3. 1999.

[Ber00] 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.

[Ber04] Gérard Berry. The esterel v5 language primer version v5 91. 2004.

[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.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001.

[BS14] Laura Bozelli and César Sánchez. Foundations of Boolean stream runtime verification. In *In Proc. RV'14*, volume 8734 of *LNCS*, pages 64–79. Springer, 2014.

[Büc90] J. Richard Büchi. *On a Decision Method in Restricted Second Order Arithmetic*, pages 425–435. Springer, 1990.

[CHL+18] Lukas Convent, Sebastian Hungerecker, Martin Leucker, Torben Scheffel, Malte Schmitz, and Daniel Thoma. TeSSLa: Temporal stream-based specification language. In *Formal Methods: Foundations and Applications - 21th Brazilian Symposium, SBMF*. 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 *Runtime Verification*. Springer, 2018.

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, 1987.

[Dav58] Martin D. Davis. *Computability and Unsolvability*. McGraw-Hill Series in Information Processing and Computers. McGraw-Hill, 1958.

[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*. IEEE, 2018.

[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*. Springer, 2017.

[DLT16] Normann Decker, Martin Leucker, and Daniel Thoma. Monitoring modulo theories. *International Journal on Software Tools for Technology Transfer*, 18:205–225, 2016.

[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, pages 92–106, 2012.

[DMF12] Luca Deri, Simone Mainardi, and Francesco Fusco. tsdb: A compressed database for time series. In *Traffic Monitoring and Analysis (TMA)*, pages 143–156, 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, 2005.

[EH97] Conal Eliot and Paul Hudak. Functional reactive animation. In *ICFP*, pages 163–173. ACM, 1997.

[FFS+19] Peter Faymonville, Bernd Finkbeiner, Malte Schledjewski, Maximilian Schwenger, Marvin Stenger, Leander Tentrup, and Torfah Hazem. StreamLAB: Stream-based monitoring of cyber-physical systems. In *Proc. of the 31st Int'l Conf. on Computer-Aided Verification (CAV'19)*, pages 421–431. Springer, 2019.

[FFST16] Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah. A stream-based specification language for network monitoring. In *Runtime Verification (RV)*, pages 152–168, 2016.

[FFST19] Peter Faymonville, Bernd Finkbeiner, Maximilian Schwenger, and Hazem Torfah. Real-time Stream-based Monitoring. *arXiv:1711.03829*, 2019.

[Fin06] Olivier Finkel. Undecidable problems about timed automata. In *Formal Modeling and Analysis of Timed Systems, 4th International Conference, FORMATS*, pages 187–199, 2006.

[FKRT18] Yliès Falcone, Srdan Krstic, Giles Reger, and Dmitriy Traytel. A taxonomy for classifying runtime verification tools. In Christian Colombo and Martin Leucker, editors, *Runtime Verification*, pages 241–262. Springer, 2018.

[GL87] Thierry Gautier and Paul Le Guernic. SIGNAL: A declarative language for synchronous programming of real-time systems. In *Functional Programming Languages and Computer Architecture*, pages 257–277, 1987.

[Gol00] Dina Q. Goldin. Persistent turing machines as a model of interactive computation. In *Foundations of Information and Knowledge Systems*, pages 116–135, 2000.

[GS18] Felipe Gorostiaga and César Sánchez. Striver: Stream runtime verification for real-time event-streams. In *Runtime Verification*, pages 282–298, 2018.

[GS20] Felipe Gorostiaga and César Sánchez. Stream runtime verification of real-time event-streams with the Striver language. *International Journal on Software Tools for Technology Transfer*, To appear, 2020.

[GSAS04] Dina Q. Goldin, Scott A. Smolka, Paul C. Attie, and Elaine L. Sonderegger. Turing machines, transition systems, and interaction. *Inf. Comput.*, pages 101–128, 2004.

[GSW01] Dina Q. Goldin, Scott A. Smolka, and Peter Wegner. Turing machines, transition systems, and interaction. In *8th International Workshop on Expressiveness in Concurrency*, pages 120–136, 2001.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 1991.

[HG05] Klaus Havelund and Allen Goldberg. Verify your runs. In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, pages 374–383. Springer, 2005.

[HK17] Sylvain Hallé and Raphaël Khoury. Event stream processing with beepbeep 3. In *RV-CuBES*, pages 81–88, 2017.

[HK18] Sylvain Hallé and Raphaël Khoury. Writing domain-specific languages for beepbeep. In *Runtime Verification (RV)*, pages 447–457, 2018.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[HR02] Klaus Havelund and Grigore Roşu. Synthesizing monitors for safety properties. In *TACAS*, pages 342–356. Springer, 2002.

[HV09] Sylvain Hallé and Roger Villemaire. Browser-based enforcement of interface contracts in web applications with beepbeep. In *Computer Aided Verification*, pages 648–653, 2009.

[JBG+15] Stefan Jaksic, Ezio Bartocci, Radu Grosu, Reinhard Kloibhofer, Thang Nguyen, and Dejan Nickovic. From signal temporal logic to FPGA monitors. In *International Conference on Formal Methods and Models for Codesign*, pages 218–227. IEEE, 2015.

[Koy90] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real Time Syst.*, pages 255–299, 1990.

[LBBG86] Paul Le Guernic, Albert Benveniste, Patricia Bournai, and Thierry Gautier. Signal-a data flow-oriented language for signal processing. *IEEE Trans. Acoustics, Speech, and Signal Processing*, 34:362–374, 1986.

[LS07] Martin Leucker and César Sánchez. Regular linear temporal logic. In *Theoretical Aspects of Computing*, pages 291–305, 2007.

[LS09] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Logic Algebr. Progr.*, 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*. 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 *Runtime Verification*, pages 273–291, 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.*, pages 745–787, 2020.

[MH84] Satoru Miyano and Takeshi Hayashi. Alternating finite automata on omega-words. *Theor. Comput. Sci.*, 32:321–330, 1984.

[MMMP12] Andreas Malcher, Katja Meckel, Carlo Mereghetti, and Beatrice Palano. Descriptional complexity of pushdown store languages. In Martin Kutrib, Nelma Moreira, and Rogério Reis, editors, *Descriptional Complexity of Formal Systems*, pages 209–221. Springer, 2012.

[MN04] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *FTRTFT*, pages 152–166, 2004.

[NBN+16] Thang Nguyen, Ezio Bartocci, Dejan Nickovic, Radu Grosu, Stefan Jaksic, and Konstantin Selyunin. The HARMONIA project: Hardware monitoring for automotive systems-of-systems. In *ISoLA*, pages 371–379, 2016.

[OW05] Joël Ouaknine and James Worrell. On the decidability of metric temporal logic. In *Symposium on Logic in Computer Science*, pages 188–197, 2005.

[OW06] Joël Ouaknine and James Worrell. On metric temporal logic and faulty turing machines. In *Foundations of Software Science and Computation Structures (FOSSACS)*, pages 217–230, 2006.

[PGMN10] Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot: A hard real-time runtime monitor. In *Runtime Verification*, pages 345–359. Springer, 2010.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[Ras99] J.-F. Raskin. *Logics, automata and classical theories for deciding real-time*. PhD thesis, Namur, Belgium, 1999.

[RRS14] Thomas Reinbacher, Kristin Yvonne Rozier, and Johann Schumann. Temporal-Logic Based Runtime Observer Pairs for System Health Management of Real-Time Systems. In *TACAS*, pages 357–372. Springer, 2014.

[RS97] Jean-François Raskin and Pierre-Yves Schobbens. State clock logic: A decidable real-time logic. In *Hybrid and Real-Time Systems, International Workshop*, pages 33–47, 1997.

[Sén99] Géraud Sénizergues. T(A) = T(B)? In Jirí Wiedermann, Peter van Emde Boas, and Mogens Nielsen, editors, *Automata, Languages and Programming*, pages 665–675. Springer, 1999.

[Ser99] Frédéric Servais. *Visibly Pushdown Transducers*. PhD thesis, Université Libre de Bruxelles, 1999.

[SLG94] Viggo Stoltenberg-Hansen, Ingrid Lindström, and Edward R. Griffor. *Mathematical theory of domains*. Cambridge University Press, 1994.

[Ste67] Richard Edwin Stearns. A regularity test for pushdown machines. *Information and Control*, 1967.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955.

[Tur36] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, (42):230–265, 1936.

[Vic89] Steven Vickers. *Topology via Logic*. Cambridge University Press, USA, 1989.

[Weg98] Peter Wegner. Interactive foundations of computing. *Theor. Comput. Sci.*, 192(2):315–351, 1998.

[WK95] Andreas Weber and Reinhard Klemm. Economy of description for single-valued transducers. *Inf. Comput.*, 1995.