UNIVERSITÄT ZU LÜBECK

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

Synchronous Stream Runtime Verification with Uncertainties and Assumptions

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

from the Department of Computer Sciences and Technical Engineering

Submitted by

Hannes Kallwies
from Bayreuth

Lübeck 2024

First referee: Prof. Dr. Martin Leucker

Second referee: Prof. Bernd Finkbeiner, PhD

Date of oral examination: August 23, 2024

Approved for printing. Lübeck, September 6, 2024

Synchronous Stream Runtime Verification with Uncertainties and Assumptions

A Generic Abstraction-based Theory for Synchronous Monitoring

Hannes Kallwies

Acknowledgments

After the successful defense of this dissertation and during the preparation of the final version, I have the opportunity to add this chapter with a few grateful words to all the people who were directly or indirectly involved in the process of this thesis.

First of all, I would like to thank my supervisor Martin Leucker. For five years, he has guided me on my way from being a novice in research to this doctoral thesis. With his broad experience, he advised me on which topics to work on and which directions to take in my research. He was always available and highly motivated for fruitful discussions about this thesis and the papers we wrote together, even though he was busy running an institute. It's no exaggeration to say that this thesis would not have been successful without his efforts.

I further wish to thank my second referee, Bernd Finkbeiner, for agreeing without hesitation to review this thesis and to travel to Lübeck for the colloquium. I would also like to thank Till Tantau for chairing the examination committee and ensuring that everything went smoothly during the examination.

Special thanks are also due at this point to César Sánchez, who co-authored all four papers on which this thesis is based, and who shared with me all his great intuition in research and helped me to write down ideas in a nice and understandable way. I also thank you, César, for the opportunity to visit you and your institute in Spain for three months. Looking back, it was one of the periods of my Ph.D. I look back with deepest joy. I loved the time in Madrid and all the great people from the IMDEA Software team that I met there.

Not only in Spain, however, but also here at my institute in Germany, I have had the opportunity to meet people with enchanting skills that I lack myself and from whom I have learned so much. Thanks for supervising my bachelor and master thesis. Thanks for writing papers and developing tools together, and thanks for chatting in the office or explaining to me LaTeX and functional programming "features" I never dreamed existed (in my nightmares). And – of course – first and foremost thanks for making the tool from chapter 6 of this thesis run on a doctor hat – to anticipate the bitcoin price with a LOLA specification ;). (Dear future readers of this text, I'd be so curious if bitcoin is actually still known when you read this text...).

No less, I want to send a special thanks to all the amazing people I got to know and spent my free time with throughout the last years. Thank you so much! For proofreading this thesis, for meeting me at the café to accompany me while writing, for playing theater with me or just going out for a drink to distract me and give me a life besides writing this thesis, and for so much more.

Last but far from least, I would like to express my gratitude to my parents, who have always supported and encouraged me on my way through school and university, and given me wise counsel at life's crossroads. Without your help, I would probably not be writing this last sentence of the acknowledgments chapter in a doctoral thesis right now.

Abstract

Runtime verification (RV) is a technique from the field of formal methods that aims at monitoring system executions for verification purposes. In particular, a so-called monitor is used, which evaluates a run of the system under scrutiny either at runtime (online monitoring) or afterwards (offline monitoring).

The monitors used for this purpose are usually not programmed in a conventional way, but rather synthesized from a specification of the property to be monitored. A particular challenge, however, is that the specification languages used, do not directly provide a strategy to evaluate the specification, e.g. if the described properties depend on future events in the system. To produce optimal results, an online monitor would then require a strategy to consider all possible extensions of the current system execution. Also, the capabilities of a monitor often go beyond a simple evaluation of a given property over a fully observable system execution. For example, there may be so-called uncertainties, i.e. points in the execution where the monitor does not have access to exact data values, e.g. because sensors have failed or provide inaccurate measurements. There may also be additional knowledge about the monitored system or its environment, called assumptions. Taking these into account during monitoring can often lead to more accurate monitoring results.

This thesis studies monitoring of synchronous properties, in the presence of uncertainties and assumptions. Synchronous properties are those that assign a property value to each monitor input, but not to the points in time between these inputs. In the thesis, the following aspects are addressed:

First, a definition of synchronous properties is worked out, together with a notion of sound and perfect monitoring in various versions. Besides, it is shown how the stream runtime verification language LOLA can be used as a general formalism for synchronous properties.

Building on this, a general framework for efficient online monitoring of LOLA specifications in the presence of uncertainties and assumptions and a concrete instantiation based on symbolic reasoning is presented. In particular, the thesis discusses how the framework can be used to show when perfect monitoring of certain LOLA fragments

is possible and how to construct such monitors. The theory presented is essentially based on foundations from the field of abstract interpretation.

Finally, three case studies and an implementation of the symbolic monitoring technique are used to demonstrate concrete application scenarios of the presented method. In addition, the potentials as well as shortcomings and possible extensions of the approach are discussed.

Zusammenfassung

Runtime Verification (RV) ist eine Technik aus dem Bereich der formalen Methoden, welche darauf abzielt, die Ausführung von Systemen zu Verifikationszwecken zu überwachen. Konkret wird dabei ein sogenannter Monitor eingesetzt, der entweder zur Laufzeit (Online-Monitoring) oder auch danach (Offline-Monitoring) den Lauf eines zu überwachenden Systems auswertet.

Die hierfür verwendeten Monitore werden in der Regel nicht auf herkömmliche Weise programmiert, sondern aus einer Spezifikation der zu überwachenden Eigenschaften synthetisiert. Eine besondere Herausforderung besteht hierbei darin, dass die verwendeten Spezifikationssprachen oft keine direkte Strategie mit sich bringen die Spezifikationen auszuwerten, z.B. wenn die dort beschriebenen Eigenschaften von zukünftigen Ereignissen im System abhängen. Um optimale Ausgaben generieren zu können, müsste ein Online-Monitor dann alle möglichen Fortsetzungen der aktuellen Systemausführung berücksichtigen. Auch gehen die Fähigkeiten eines Monitors oft über die bloße Auswertung einer Eigenschaft über einem vollständig einsichtigen Systemlauf hinaus. So kann es z.B. sogenannte Uncertainties geben, d.h. Stellen im Systemlauf, an denen dem Monitor keine exakten Datenwerte zur Verfügung stehen, etwa weil Sensoren ausgefallen sind oder ungenaue Werte liefern. Darüber hinaus können zusätzliche Informationen über das überwachte System oder seine Umgebung vorliegen, die als Assumptions bezeichnet werden. Deren Berücksichtigung während des Monitorings kann oft zu präziseren Ausgaben des Monitors führen.

Diese Arbeit untersucht das Monitoring von synchronen Eigenschaften unter Berücksichtigung von Uncertainties und Assumptions. Synchrone Eigenschaften sind solche, die jeder Monitor Eingabe einen Eigenschaftswert zuweisen, nicht aber den Zeitpunkten zwischen diesen Eingaben. In der Arbeit werden folgende Aspekte behandelt:

Zunächst werden synchrone Eigenschaften und der Begriff des korrekten und perfekten Monitorings in verschiedenen Ausprägungen definiert. Daneben wird auch gezeigt, wie die Strom Runtime Verification Sprache LOLA als allgemeiner Formalismus für synchrone Eigenschaften dienen kann.

Darauf aufbauend wird dann ein allgemeines Framework für das effiziente Online-Monitoring von LOLA-Spezifikationen bei Vorliegen von Uncertainties und Assumptions zusammen mit einer konkreten Instanziierung auf Basis von symbolischem Schließen vorgestellt. Dabei behandelt diese Arbeit im Speziellen, wie das Framework benutzt werden kann, um zu zeigen wann ein perfektes Monitoring bestimmter LOLA-Fragmente möglich ist und wie solche Monitore konstruiert werden können. Die eingeführte Theorie basiert dabei im Wesentlichen auf Grundlagen aus dem Feld Abstract Interpretation.

Abschließend werden konkrete Anwendungsfälle für die vorgestellte Methodik anhand von drei Fallstudien und einer Implementierung des symbolischen Ansatzes aufgezeigt. Dabei werden sowohl Potentiale als auch Probleme des Ansatzes identifiziert und mögliche Erweiterungen diskutiert.

Contents

1

Introduction

Especially since the emergence of personal computers in the second half of the 1970s, computing machinery has found its way into almost every area of modern life. Beyond the private sphere, computers are used in everyday office life, in industry, and embedded in a wide variety of devices where they help to make processes much faster and more efficient. It's no exaggeration to say that it's hard to imagine living without them these days.

In addition to the enormous productivity gains, their widespread use is fraught with dangers too, especially in the case of a deviation between the desired and the actual behavior, often due to software bugs. In safety-critical applications (e.g. airplanes or nuclear power plants), even minor errors can have disastrous consequences, as for example in the crash of NASA's mars climate orbiter, where the use of different units of measurement throughout the code base led to its destruction [Mis99]. On the other hand, software bugs can also be easily exploited by attackers, posing a substantial threat to individual companies, industries or even states. With the increasing complexity of software in recent years, this problem has grown. Especially when machine learning techniques are used, as is increasingly the case in software applications, the behavior of the software is usually no longer fully predictable by its developers.

For these reasons, the area of secure and reliable software development is gaining more and more attention in both academia and industry. A fundamental part of this field is the development of tools, e.g. programming languages, that avoid the creation of erroneous code up front, and the study of how the software development process can be shaped to enable the creation of quality-assured software. Finally, a further integral component are formal techniques that can be used to check or even prove that the functionality of software conforms to its specification. This dissertation finds itself located in this subarea, called formal methods.

1.1. Formal methods

Formal methods are described as "mathematically and logically based framework[s] for specifying, developing, and verifying systems" [BNK16]. Thereby *specification* names a formal description of the system behavior and *verification* the process of examining whether a system (called *system under scrutiny (SUS)*) adheres to its specification [IEE98, LS09, BNK16]. As such the term is to be distinguished from *validation*, which deals with the question if the system satisfies the requirements of the user [Ins13, IEE98].

Prominent manifestations of formal methods are theorem proving, static analysis, model checking and runtime verification. The first three of these techniques aim at the irrefutable proof that a system behaves correctly. *Theorem proving* [Lov78, BC04] does so by providing tools to write (partially automated) computer-verifiable proofs that a program is correct. *Static analysis* [RY20] names a family of code analysis techniques to infer properties about its behavior. *Model checking* [CGK$^+$18] finally relies on techniques to check automatically that a formal representation (model) of the SUS satisfies the specification on all possible executions.

Besides concrete algorithms all of the mentioned approaches rely on a formal specification of the desired system behavior. Therefore several logics and specification languages have been developed over time, starting from simple logics like *linear temporal logic (LTL)* [Pnu77]. In general these specification languages are formal languages that precisely describe properties of the system. However, in difference to programming languages, they tend to have a more declarative character, describing the system behavior or parts of it, rather than indicating the particular computation steps. Furthermore they often involve a temporal dimension that enables them to relate the current program state with past and future ones.

While theorem proving is complex and increases the effort of software development tremendously, static analysis and model checking suffer from the problem that in general (even simple) semantic properties of Turing-complete programming languages cannot be automatically decided (see [Ric53]).

Usually not considered a formal method, but also used for program verification, is *testing*. Testing consists of executing the SUS on a predefined set of inputs and checking the behavior against a set of (usually manually) prepared outputs. As such testing is perfectly feasible, yet can never provide the proof that a system is correct for inputs which have not been tested (see [Dij72]).

Runtime verification (RV) [LS09], often classified as partial or lightweight verification technique, finds itself located between full verification approaches (like model checking or static analysis) and conventional testing [LS09] and can be considered a compromise between both. In difference to the other formal methods, runtime verification does not consider every possible execution of a system but only a single

run, for which it checks satisfaction of the specification. The major part of this thesis deals with the topic runtime verification.

1.2. Runtime verification

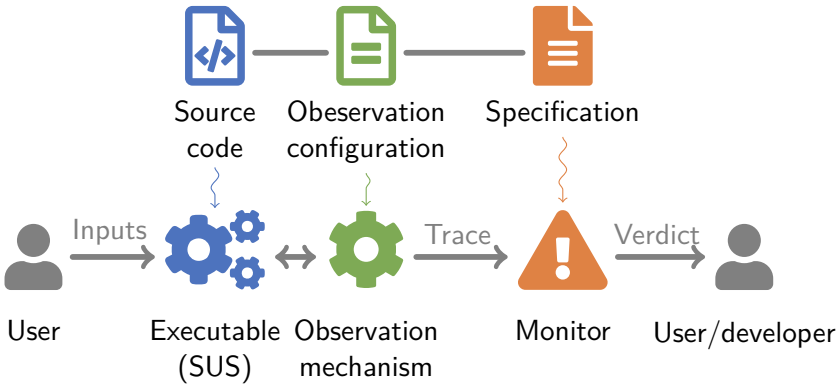


Figure 1.1.: General runtime verification architecture (figure based on [KLS+22b]).

In the standard RV setting [LS09, BFFR18, FKRT21, KLS+22b], visualized in figure 1.1, a so-called *monitor* is generated automatically from a *correctness property*, formalized in a *specification*, and supervises whether the current *run* of the *system under scrutiny (SUS)* fulfills this property.

To this end, the monitor receives a sequence of information about the SUS (so-called *observations*, often states, i.e. mappings of the variables to their current values or notifications about relevant events in the system like function calls or variable assignments), called *trace*. This trace is usually produced by an *observation/instrumentation mechanism*, which tracks the execution of the SUS and yields the relevant inputs for the monitor. There is a variety of such observation tools. They differ in aspects like intrusiveness, i.e. whether the observer manipulates the (timing) behavior of the observed system – e.g. by directly instrumenting the binary's (source) code – or performs non-influencing observation. In addition to software observers, there are also fully non-intrusive hardware tracing tools, which extract the observations of the SUS for example directly from the tracing interface of the SUS's CPU (e.g. [ADFdB13, DGH+17, CHS+18]). In general, the SUS is not limited to a pure computing system, but may also include sensors and actuators. In the case of monitoring such *cyber-physical systems (CPS)*, the input trace usually also includes sensor readings and other measurements from the environment.

The correctness property itself is formalized in a specification by means of a dedicated formalism (e.g. a logic like LTL). The instructions on which outputs shall be generated by the observer tooling are given through an *observation configuration*.

However, some specification formalisms also allow an embedding of this configuration directly in the specification language (e.g. [KLS+22b]).

One generally distinguishes between *online* and *offline* runtime verification, depending on whether the monitor is executed during the observed system's runtime or works on pre-recorded traces.

In the traditional setting the monitor produces *verdicts* during receiving the SUS's observations, which indicate to which degree the correctness property is satisfied by the current run. In the simplest case these verdicts are a value from the boolean domain, i.e. true or false, depending on whether the specified behavior is violated or not. However in more advanced settings also other domains, like numbers between 0 and 1, expressing a measure to which degree the property is satisfied or similar are thinkable [LS09].

The ongoing research in the field runtime verification mainly focuses on the specification formalisms and their properties, like conciseness and expressiveness, and algorithms for the efficient synthesis of monitors from specifications. However also related topics, like trace generation or *runtime reflection* [LS09] (i.e. the combination of verification and automated failure mitigation), are investigated. An overview of relevant topics in RV can be found in [SSA+19, FKRT21].

More recently, extensions to RV have been considered that depart from the traditional idea of monitoring a correctness property over the entire trace and casting a verdict. Instead, monitors are used to perform more complex computations on the received observations, such as determining statistical values or providing debug information about faulty locations. A prominent approach in this direction is *stream runtime verification (SRV)*, where the monitored property is described as a stream transformation.

A notable feature of these SRV languages and several other recent RV approaches is that they assign a value to each position of the input trace. Traditional formalisms instead tend to assign a single value, often a truth value, to the whole trace, e.g. to indicate whether the trace is satisfied by the property as a whole, or not. While for the latter monitoring is (especially in the case of online monitoring) concerned with producing outputs, that enclose this single property value as good as possible, the previously named formalisms necessitate different monitoring techniques being able to answer for different trace locations. For online monitoring, a common approach is to cast the value for a given trace position once all the inputs on which the value depends are available. This is not necessarily for the current position up to which the monitor has received inputs, because the specification may relate a property value at a particular position to past and future input values. As a result, this monitoring approach can lead to a situation where outputs can only be cast when the full input trace is available. A specification could for example define the property value at trace position t as the sum of all future input values from t to the trace end. In

this case the monitor would have to receive all inputs (i.e. the full trace) in order to provide the values for the specific trace positions.

1.2.1. Stream runtime verification

As mentioned, the fundamental concept of stream runtime verification is to consider the monitoring process as stream transformation from input streams, which originate from the observed system, and bear information about events that occur inside of it, to output streams. The output streams contain the results of the monitoring, i.e. verdicts about the correctness or any other quantitative information about the monitored system. Streams in this context are (possibly timed) sequences of data events, whose type is - from a theoretical point of view - usually not restricted to specific domains. The monitor can ultimately be considered as an execution engine of the stream-based specification.

Stream runtime verification languages usually define three different kinds of streams in their specifications. Input streams, whose data is passed from outside to the monitor. Intermediate streams (sometimes also called defined streams) which are defined as the application of certain stream operators on input or intermediate streams. These streams are computed internally by the monitor and usually represent interim results of the computation. Finally some of the intermediate streams are marked as output streams. The events on these streams are cast as monitor outputs.

Stream runtime verification was pioneered by the language LOLA [DSS+05]. LOLA belongs to the family of *synchronous stream runtime verification* languages (see [GDS20]). In this setting, the set of instants, i.e. points in time at which events can occur on input and output streams, is discrete. Thus there is a fixed "grid" where each stream of the specification has an event, but not in between. In the case of LOLA, the instant domain is simply the set of natural numbers between 0 and a maximal timestamp t_{max}.

```
input vel: ℝ^≥0

err := err[-1|false] ∨ (vel[now] ≥ 5)

output err
```

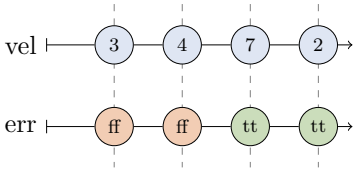


Figure 1.2.: Example LOLA specification and visualization.

A very simple LOLA specification together with a visualization can be found in figure 1.2. The specification defines an input stream `vel` (for velocity) that receives inputs of type $\mathbb{R}^{\geq 0}$, which could for example come from the speed sensor of an autonomous system. Output stream `err` of boolean type indicates whether `vel` was greater or equal 5 now or at any instant in the past. This is done with help of the

LOLA offset operator `err`[-1|`false`]. Because of the -1 offset it takes the value of stream `err` one instant before with `false` as default value at instant 0, when there is no previous instant (similar to the negative offset operator, LOLA also supports offset operators with positive offset, which refer to future events). Altogether the definition of `err` effects the corresponding stream to bear a true event, if the current event of `vel` exceeds or equals 5 or `err`'s previous event was true. Right to the specification there is a visualization of an input stream for the specification and its corresponding output stream `err`. A full introduction to syntax and semantics of LOLA will follow in section 2.3.1 of this thesis.

Another SRV paradigm is that of *asynchronous stream runtime verification* [GDS20] as followed by the SRV language TeSSLa [CHL+18]. In difference to the synchronous setting the instant domain might be dense, i.e. non-discrete, e.g. the set of all reals. Additionally the streams in this setting may have events only at a subset of instants and a specification can have output events at instants where there are no input events.

The main focus of this thesis will be on the monitoring of synchronous, particularly LOLA specifications. However also adjustments to the asynchronous setting will briefly be discussed but details left for future work.

1.2.2. Monitoring

In runtime verification a monitor is said to be synthesized from a specification. For most RV approaches this synthesis goes beyond a pure execution of the specification and requires non-trivial constructions.

On the one hand this is because runtime verification formalisms focus on a convenient description of complex temporal (correctness) properties rather than providing the single steps the monitor should perform. This is e.g. reflected in the capability of lots of RV formalisms to refer to events that will occur in the future. An online monitor construction for such formalisms in turn requires a strategy to evaluate them without the future being actually available. To do so, monitors usually need a technique to efficiently consider all future continuations of the received trace, which is often done by exploration of the monitoring state space during the synthesis. The ability of monitoring approaches to consider all possible future trace continuations and to cast most precise verdicts as soon as possible is called *anticipation* [BLS10].

On the other hand monitors should adhere to certain execution guarantees. Especially in online monitoring the complexity of each monitoring step should be constant. This often requires sophisticated strategies to extract and store the gist of the received trace so far, as memorizing the whole trace would directly lead to non-constant resource demands. However, whether a constant-resource monitoring is possible at all is of course also a matter of the utilized formalism and the concrete property.

Finally, runtime verification approaches often also go beyond pure trace checking or evaluation. Two extensions of the basic setting, which will play a significant role in this thesis, are monitoring under uncertainty and assumptions.

Uncertainties (in the inputs) denote the situation that not all events of the input trace or input streams are fully accessible. Either the value or existence of some input events is completely unknown or imprecise, i.e. a set of concrete values is possible. The presence of uncertainties generally makes monitoring harder, because the monitor has to consider more potential system states and handle them in parallel. However uncertainties play an important role in runtime verification as oftentimes not all parts of the supervised system are observable, though they play a role in the monitored property. Likewise monitor inputs resulting from cyber-physical systems often rely on sensor readings with a measurement error and thus inherently contain uncertainty.

Assumptions can be considered as the natural counterpart of uncertainty. An assumption describes a piece of additional knowledge about the observed system or the environment in which system and monitor run [HS20]. Assumptions restrict the set of actually possible input traces or streams that can be passed to the monitor. They can thus be used in the monitoring process to rule out uncertain input readings or future continuations of the trace, thereby providing an opportunity for runtime verification that may lead to more accurate monitoring results.

Although the user of course has to provide the concrete assumptions in the runtime verification process, the handling of uncertainties and assumptions is usually a monitoring feature and should not require the user to adjust the specification, as from a logical point of view the monitored (correctness) property does not change with the presence of uncertainties and assumptions. As a consequence the capability of a monitor to deal with assumptions and uncertainties usually requires an enhanced monitor synthesis strategy, but is mostly transparent for the user of the monitor.

```
input vel: ℝ^≥0

diff := vel[now] - vel[-1|0]
err := err[-1|false] ∨ (vel[now] ≥ 5)

output err

Assumption:
    |diff[now] - diff[-1|0]| ≤ 1
```

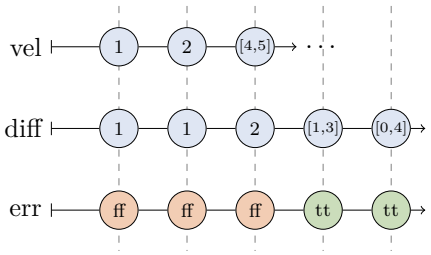


Figure 1.3.: Example of anticipatory LOLA monitoring under uncertainties and assumptions.

The interplay of monitoring with uncertainties and assumptions is illustrated in the following example (specification and visualization can be found in figure 1.3): Imagine an anticipatory LOLA monitor for the specification from figure 1.2 with an

additional stream `diff` that computes the speed difference between the current and previous instant. Now take the additional assumption: The value of `diff` can vary by at most 1 from the value one step ago. In other words: the absolute value of the the velocity's second derivation is bounded by 1.

Imagine the monitor in the example has received an event with value 1, followed by an event with value 2 on the input stream. As third event it received the uncertain input reading $[4, 5]$, i.e. the actual value is somewhere in the interval between 4 and 5. The rest of the input stream is not available yet.

An anticipatory monitor is already able to determine all events on output stream `err` without having to receive further inputs. This is because of the following reasoning: The difference between the first two `vel` events (i.e. the value of `diff` at the second instant) is 1. Thus by assumption, the third event of stream `diff`, i.e. the difference between the second and third event on stream `vel`, can only be between 0 and 2. This implies, however, that the only possible value of stream `vel` at the third instant is 4. A higher value would cause the difference to be more than 2 and thus break the assumption. The value of stream `err` is thus false at this instant. As further consequence, the difference between second and third event of stream `vel` (i.e. the value of `diff` at the third instant) can be determined as 2. This in turn causes the value of `diff` at the subsequent, fourth instant to be between 1 and 3 due to the assumption, and thus the value of the next velocity reading (which has not yet been received) must be between 5 and 8. With this information the anticipatory monitor can finally conclude that `err` at the fourth instant is true and by definition also at all following positions.

1.3. Contributions of this work

This thesis studies general runtime verification of synchronous properties. This includes the anticipatory monitoring of arbitrary synchronous formalisms under presence of uncertainty and assumptions.

The work is based on a formal classification of synchronous properties and their monitoring. In this respect, the thesis is not restricted to correctness properties as used in traditional runtime verification, but allows for arbitrary value domains. Two types of properties are distinguished, *initial properties* as used in traditional runtime verification, where a single value is assigned to a complete input trace, and *pointwise properties*, which provide a value for each trace location. The pointwise properties can be seen as a generalization of the initial properties. The SRV language LOLA is also shown to be a general formalism for finite synchronous pointwise properties. Thus, the work bridges the gap between traditional runtime verification formalisms, like LTL, and more advanced approaches, such as stream runtime verification.

Based on the concept of pointwise properties, several novel monitoring approaches (and conditions for their perfectness under uncertainty and assumptions) are defined. This covers in particular *recurrent monitoring*, where the monitor always yields outputs about the property valuation at the current instant and *k-offset recurrent monitoring* where the outputs are delayed or preponed by a fixed offset. Finally so-called *random access recurrent monitoring* is proposed, where the monitor operates as a "question answering machine" being able to cast outputs for arbitrary trace locations within a query domain.

The second part of the thesis then covers an approach to recurrent LOLA monitoring in the presence of uncertainty and assumptions. Specifically, a general framework based on abstract interpretation [CC77, Cou21] is introduced, which allows the creation of a sound or perfect monitor if an adequate abstraction for the stream values is available. In addition, a concrete instantiation of this general framework, based on symbolic representation of monitoring states combined with SMT solving, is presented and its perfectness for some specific LOLA fragments is investigated. The proposed recurrent LOLA monitoring approach is further shown to be able to solve the initial and random access recurrent monitoring problem for several relevant query domains.

Finally, a prototypical implementation of the aforementioned symbolic approach and the opportunities and limitations of a practical application are discussed on the basis of three case studies.

Overall, the main contribution of this work consists in the introduction of a general, abstract framework that provides a theory on how synchronous formalisms can perfectly or soundly be monitored in the presence of uncertainties and assumptions, and in the development of a symbolic monitoring approach as a powerful realization of this framework.

The content of this thesis is mainly based on the following four publications:

1. [KLSS22] Hannes Kallwies, Martin Leucker, César Sánchez, and Torben Scheffel. Anticipatory recurrent monitoring with uncertainty and assumptions. In Thao Dang and Volker Stolz, editors, *Runtime Verification - 22nd International Conference, RV 2022, Tbilisi, Georgia, September 28-30, 2022, Proceedings*, volume 13498 of *Lecture Notes in Computer Science*, pages 181–199. Springer, 2022

   where the concept of anticipatory recurrent monitoring, i.e. monitoring of pointwise synchronous properties together with reasoning about the future is introduced.

2. [KLS22a] Hannes Kallwies, Martin Leucker, and César Sánchez. Symbolic runtime verification for monitoring under uncertainties and assumptions. In Ahmed Bouajjani, Lukás Holík, and Zhilin Wu, editors, *Automated Technology for Verification and Analysis - 20th International Symposium, ATVA 2022,*

Virtual Event, October 25-28, 2022, Proceedings, volume 13505 of *Lecture Notes in Computer Science*, pages 117–134. Springer, 2022

where symbolic runtime verification for LOLA, but restricted to past specifications, is studied.

3. [KLS23] Hannes Kallwies, Martin Leucker, and César Sánchez. General anticipatory monitoring for temporal logics on finite traces. In Panagiotis Katsaros and Laura Nenzi, editors, *Runtime Verification - 23rd International Conference, RV 2023, Thessaloniki, Greece, October 3-6, 2023, Proceedings*, volume 14245 of *Lecture Notes in Computer Science*, pages 106–125. Springer, 2023

   where an anticipatory monitoring algorithm for boolean LOLA and several finite logics over atomic propositions is developed and implemented.

4. [HKLS24] Raik Hipler, Hannes Kallwies, Martin Leucker, and César Sánchez. General anticipatory runtime verification. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part II*, volume 14682 of *Lecture Notes in Computer Science*, pages 133–155. Springer, 2024

   where a general anticipatory monitoring approach for LOLA is presented.

1.4. Related work

The use of temporal logics to reason about the correctness of computer programs was first studied by Amir Pnueli in his 1977 work [Pnu77], where linear temporal logic (LTL) was employed to describe the correct temporal behavior of the system. Pnueli's approach was later adapted to model checking and from 2000 on the field of runtime verification emerged. Besides the development of tools and ecosystems for runtime verification the research in this area extended the basic approach of checking whether a run adheres to an LTL formula into various directions.

On the one hand (not restricted to runtime verification, but in general in the area of formal methods) several new logics, often extensions of standard LTL (see [MP91]) have been developed. In [LPZ85, GV13] adaptions of LTL's semantics to finite traces have been studied. In [LS07, SL10] LTL was extended with special operators to check if substrings of the trace match a regular expression, to gain higher expressiveness. A similar approach was followed by De Giacomo and Vardi in [Var11, GV13] where LTL has also been enriched with regular expressions. In 1990 Koymans investigated the extension of LTL with time by adding time bounds to the operators to give temporal restrictions when they have to hold [Koy90]. The resulting metric temporal logic (MTL) was adjusted to metric interval temporal logic (MITL) [AFH91] which has later been extended to signal temporal logic (STL) [MN04]. STL is able to express properties of real-valued streams (signals) and is therefore often used for monitoring

of cyber-physical systems. With LOLA [DSS+05] the paradigm of stream runtime verification was introduced which was subsequently adopted and further developed by numerous other formalisms: Lola 2.0 [FFST16], RTLola [FFS+19, FFST17], Striver [GS18], TeSSLa [LSS+18, CHL+18] and Copilot [PGMN10]. Stream runtime verification languages are closely related to synchronous dataflow programming languages, whose most prominent representatives are Lustre [CPHP87], Esterel [BG92] and SIGNAL [GL87] and to the field of functional reactive programming [EH97]. Other well-known advanced formalisms for runtime verification include QTL [HPU17], mission-time LTL [RRS14], Eagle [GH05], RuleR [BRH10], MFOTL [BHKZ11], and many more. Besides more expressive formalisms, another direction in the development of RV languages is to make them more natural language like and thus easier to use. Corresponding approaches are e.g. the LTL extension SALT [BLS06b] or FRETish [GPMS20].

On the other hand research in runtime verification also focuses on approaches for monitoring. For online monitoring, this mainly covers strategies how to cast precise verdicts for trace prefixes that do not directly reveal if a monitored property is ultimately satisfied or not. For future LTL, monitoring semantics and belonging monitor constructions have been introduced which assign values from a truth domain to a trace prefix, e.g. LTL_3 [BLS06a] or LTL_4 [Leu11]. As quality characteristic, [BLS10] defined the concepts of impartiality and anticipation. While these approaches are all based on the initial semantics of future LTL, an online monitoring strategy for past LTL w.r.t. its pointed semantics was presented in [HR02].

The presence of uncertainty in the input trace which is passed to the monitor, has also been studied in several fashions. In general, one can distinguish between approaches that consider (learned) probability distributions when inputs are noisy or unknown (e.g. [SBS+11]), and those that assume a set of possible values without further information about their likelihood (e.g. [LSS+19]). A common approach for the latter one is to compute in an abstract domain which is capable of representing sets of potential concrete values. This idea was first applied to runtime verification in [LSS+19]. Another foundational work on uncertainty in runtime verification is [KHF19, KHF21], where Kauffman et al. provide a theoretical consideration of the circumstances under which a property is ignorant of inaccuracies or gaps in the input trace. A detailed overview of the general topic of runtime verification under uncertainty can be found in [THK23].

The incorporation of knowledge about the system during the monitoring process and the resulting relation to model checking was first discussed in [Leu12]. The idea was studied for LTL under the term “assumption” in [CTT19] and later in [CTT21] for an extension of LTL with linear arithmetic. In [HS20] theoretical foundations of monitoring under assumptions are laid.

In recent times also the usage of symbolic approaches in monitoring has come to the focus of research, e.g. [WAH19, CTT19]. The techniques used in [CTT21] and [FMPW23] also belong to this family and are the closest ones to the symbolic

monitoring strategies presented in this thesis, as they allow for anticipatory monitoring of advanced (linear arithmetic) properties. They use symbolic constraints to describe the current monitoring state and solve the monitoring problem – in the case of [CTT21] also under presence of uncertainties and assumptions – by symbolic reasoning. This thesis however differs from them in that they both restrict to the monitoring of correctness and not general properties. Likewise the approach for anticipation varies quite heavily. While [CTT21] uses bounded model checking from the current position and [FMPW23] performs a (precomputed) emptiness per state check of the current monitor state by exploring the whole space state (and is thus only guaranteed to terminate in case of a finite one), the strategy followed in this thesis is to symbolically unroll the specification from backward. In case of undecidability or intractability of anticipatory monitoring, the symbolic technique from this thesis falls back to an over-approximation, ending up with a sound but imperfect monitor (defined later). Furthermore, this work suggests a general anticipatory monitoring theory with uncertainty and assumptions and is not restricted to a symbolic implementation.

The basis for the general monitoring strategy in this thesis is the concept of abstraction, which is a fundamental approach in the area of static analysis as well. Several theoretical findings are based on abstract interpretation [CC77, CC92, Cou21], presented by Patrick and Radhia Cousot in 1977, which in its most general form serves as a framework for construction of sound fixed point approximations.

The idea of symbolic computation has also been explored in program verification. Most notably in symbolic execution [Kin76], a method introduced in 1976 by J.C. King as a debugging technique for computer programs. It is also a common approach to use symbolic descriptions of program states in static analysis, for example studied in [CH78] by Cousot and Halbwachs in 1978.

1.5. Thesis structure

In the second chapter of this thesis basic formalisms and concepts from the areas runtime verification and fixed point computation are introduced.

Chapter 3 then presents a general monitoring theory. It provides formal definitions for initial and synchronous pointwise properties and presents online monitoring approaches for them, namely initial, recurrent, k-offset and random access recurrent monitoring. Further it discusses the notion of sound and perfect monitoring under uncertainty and assumptions with respect to these monitoring techniques. Finally it is shown that LOLA is a general formalism for synchronous properties and translations from LTL, M(I)TL and STL to LOLA are provided. Furthermore the standard LOLA monitoring algorithm is classified in terms of the previously defined monitoring theory.

Anticipatory online monitoring of arbitrary LOLA specification under uncertainty and assumptions is studied in chapter 4. Therefore a general semantics for perfect online LOLA monitoring is introduced. Subsequently the chapter studies recurrent monitoring under uncertainties and assumptions:It is shown how several problems of practical relevance, including random access monitoring for certain query domains, can be reduced to recurrent monitoring. Thereafter, a generic abstraction-based recurrent monitoring theory is presented.

In chapter 5 an instantiation of this generic framework using symbolic constraints is outlined. To this end, the encoding of stream events and their relation in symbolic constraint sets – also in the presence of uncertainty – is addressed. Furthermore, rewriting strategies for these sets into a more concise form for different classes of constraints are discussed, which form the basis for resource-efficient monitoring. Finally, a symbolic monitoring approach and results on the perfect and efficient monitorability of certain LOLA fragments are presented.

Chapter 6 covers an evaluation of a prototypical implementation of the symbolic, anticipatory LOLA monitoring approach from the previous chapter. Therefor the strengths and weaknesses of the approach are examined on the basis of three practical case studies.

In chapter 7 the results of the overall work are summed up and discussed again. Additionally, directions for future research are identified.

2

Preliminaries

In advance of the main contribution of the thesis, this chapter contains the theoretical foundations that will be used in the following chapters. It starts with the introduction of basic concepts used throughout the work, continues with a general overview of (stream) runtime verification, provides syntax and semantics of various popular formalisms and presents common monitoring constructions. Furthermore, it prepares the necessary technical details in the areas of fixed point computation and abstract interpretation.

2.1. Basic concepts

We start with basic definitions of traces and (timed) streams, ordered sets and lattices as well as algebras, which will play an important role in the remainder of the work.

2.1.1. Traces

As outlined in the previous chapter, runtime verification (RV) is concerned with monitoring of system runs. Therefore, we will start with a formal introduction to different concepts of these runs.

The most basic notion is to consider a run as a (finite or infinite) sequence of system observations. These observations are elements from a possibly infinite data domain.

> **Definition 2.1** (Data domain)**.**
>
> A *data domain* is a set of data values. A finite data domain is called *alphabet*.

We call a recorded sequence of values from a data domain trace. In the case of a finite data domain also word.

Definition 2.2 (Finite and infinite trace)**.**

Let Σ be a data domain.

A finite sequence $w = \langle a_0, a_1, \ldots, a_n \rangle$ with $a_i \in \Sigma$ is called a *finite trace* over Σ. An infinite sequence $w = \langle a_0, a_1, \ldots \rangle$ with $a_i \in \Sigma$ is called an *infinite trace* over Σ.

A finite trace over a finite alphabet Σ is called a *word*. An infinite trace over a finite alphabet Σ is called an *ω-word*.

The expression Σ^* denotes the set of finite traces over Σ, Σ^n the set of traces of length $n \in \mathbb{N}$ over Σ, Σ^ω the set of infinite traces over Σ and $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$.

We will use $|w|$ to denote the length of a finite trace $w \in \Sigma^*$, for $w \in \Sigma^\omega$ we have $|w| = \infty$. We write $w(i)$ to refer to w's element (a.k.a. *letter*) at position $i \in \mathbb{N}$, with $0 \leq i < |w|$. Additionally we use $w^{(i)} = \langle w(0), w(1), \ldots, w(i) \rangle \in \Sigma^*$ for w's prefix up to letter i. For $w = \langle a_1, a_2, a_3, \ldots \rangle \in \Sigma^\infty$ we also use the notation $w = a_1 a_2 a_3 \ldots$. With $\epsilon = \langle\rangle \in \Sigma^*$ we denote the empty trace.

2.1.2. Streams

A significant part of this thesis deals with stream runtime verification. This special form of runtime verification conceives the inputs and outputs of the monitor as so-called streams.

The most basic form of streams are synchronous event streams without time information.

Definition 2.3 (Synchronous event stream; based on [DSS+05])**.**

A *synchronous (event) stream* over data domain $\mathbb{D}$ is a total function $s : \mathbb{T} \to \mathbb{D}$ where $\mathbb{T} \subseteq \mathbb{N}$ is either a finite ($\mathbb{T} = \{0, 1, \ldots, t_{max}\}$) or an infinite ($\mathbb{T} = \mathbb{N}$) *instant domain*.

$\mathcal{S}^{\mathbb{T}}_{\mathbb{D}}$ is used to denote the set of all synchronous event streams over data domain $\mathbb{D}$ and instant domain $\mathbb{T}$.

A synchronous event stream is called finite if and only if $\mathbb{T}$ is *finite*, else *infinite*.

Unless otherwise stated, we assume throughout this thesis that the time domain of the streams, including t_{max} in the case of finite streams, is known. Note, however,

that this is not always necessarily the case in standard online monitoring scenarios, where a monitor continuously receives data events, but does not know about the total number that will eventually be read. If $\mathbb{T}$ is clear from the context, we will also use $\mathcal{S}_{\mathbb{D}}$ to denote $\mathcal{S}_{\mathbb{D}}^{\mathbb{T}}$.

Note also that synchronous streams are essentially equivalent to traces of fixed length $t_{max} + 1$ or infinite traces, respectively. For this reason we can identify the two concepts and use synchronous event streams and traces of corresponding lengths interchangeably.

2.1.3. Timed streams

An extension to the concept of streams as defined above is obtained by allowing $\mathbb{T}$ to be an arbitrary time domain and not restricted to a contiguous subset of $\mathbb{N}$. For time domains we require a unique smallest element and an order on the timestamps.

Definition 2.4 (Time domain; based on [Koy90])**.**

A *time domain* is a totally and strictly ordered structure $(\mathbb{T}, <, 0)$ with a smallest element 0. An element of $\mathbb{T}$ is called *timestamp*.

A time domain $(\mathbb{T}, <, 0)$ is called *finite* if and only if there is a $t_{max} \in \mathbb{T}$ such that

$$\forall t \in \mathbb{T}.\, t \leq t_{max}.$$

Further, a time domain $(\mathbb{T}, <, 0)$ is called *non-discrete* if and only if

$$\exists t \in \mathbb{T} \backslash \{0\}.\, \forall t' \in \mathbb{T}.\, t' < t \rightarrow \exists t'' \in \mathbb{T}.\, t' < t'' < t$$

otherwise *discrete*.

It is considered *dense* if and only if

$$\forall t, t' \in \mathbb{T}.\, t' < t \rightarrow \exists t'' \in \mathbb{T}.\, t' < t'' < t.$$

When $\cdot < \cdot$ and 0 are clear from the context we just use $\mathbb{T}$ to denote the time domain. Discrete time domains can be iterated according to their order. Non-discrete time domains are exactly those where you can have an infinite ordered sequence of increasing timestamps with a bound. If there are infinitely many timestamps between all pairs of timestamps the domain is said to be dense. A dense time domain is by definition always non-discrete.

Formalisms over timed streams usually require some kind of distance measure between the elements of the time domain. In this thesis we restrict ourselves to the

special case of an $\mathbb{R}^{\geq 0}$ distance measure. In general, however, arbitrary other measure domains are conceivable, as long as they satisfy certain criteria that guarantee a sound computation on them (see [Koy90]).

Definition 2.5 ($\mathbb{R}^{\geq 0}$ time distance measure; based on [Koy90])**.**

A structure $(\mathbb{T}, <, 0, \ominus)$ is called *time domain with* $\mathbb{R}^{\geq 0}$ *distance measure* if $(\mathbb{T}, <, 0)$ is a time domain and $\ominus : \mathbb{T} \times \mathbb{T} \to \mathbb{R}^{\geq 0}$ is a distance measure between two timestamps s.t.

- $t \ominus t' = 0 \Leftrightarrow t = t'$,
- $t \ominus t' = t' \ominus t$,
- $t < t' < t'' \Rightarrow (t'' \ominus t) = (t' \ominus t) + (t'' \ominus t')$.

Common time domains are

- the discrete, infinite time domain $(\mathbb{N}, <, 0, |\cdot - \cdot|)$,
- the discrete, finite time domain $(\mathbb{N} \cap [0, t_{max}], <, 0, |\cdot - \cdot|)$ for $t_{max} \in \mathbb{N}$
- the dense, infinite time domain $(\mathbb{R}^{\geq 0}, <, 0, |\cdot - \cdot|)$ and
- the dense, finite time domain $(\mathbb{R} \cap [0, t_{max}], <, 0, |\cdot - \cdot|)$ for $t_{max} \in \mathbb{R}^{\geq 0}$.

where the distance measure is chosen as the absolute difference between two timestamps.

When clear from the context we use $\mathbb{N}$, $\mathbb{N}^{t_{max}}$, $\mathbb{R}^{\geq 0}$, $\mathbb{R}^{\geq 0, t_{max}}$ respectively to denote these time domains.

Based on the notion of a time domain, we define timed streams as extension of synchronous event streams. The concept of timed streams is equivalent to that of metric models, as used in metric temporal logic (MTL) [Koy90]. Therefore we identify both concepts in this thesis.

Definition 2.6 (Timed stream; based on [Koy90, MN04])**.**

Let $(\mathbb{T}, <, 0)$ be a time domain. A *timed stream (or metric model)* over data domain $\mathbb{D}$ and time domain $\mathbb{T}$ is a total function $s : \mathbb{T} \to \mathbb{D}$.

$\mathcal{S}_{\mathbb{D}}^{\mathbb{T}}$ is used to denote the set of all timed streams over data domain $\mathbb{D}$ and time domain $\mathbb{T}$.

We call a timed stream finite, infinite, non-discrete, discrete, and dense if and only if its time domain is. Timed streams over dense time domains are also referred to as *signals* (see [MN04]). As before, we can drop the $\mathbb{T}$ and just write $\mathcal{S}_{\mathbb{D}}$ if it is clear from the context.

Note that synchronous event streams (definition 2.3) are a special case of timed streams with $\mathbb{T} = \mathbb{N} \cap [0, t_{max}]$ or $\mathbb{T} = \mathbb{N}$ resp., which justifies the same notation $\mathcal{S}_{\mathbb{D}}^{\mathbb{T}}$ for both. However, unlike synchronous event streams, timed streams can generally

not be represented by traces, since there may be an infinite number of timestamps between any two timestamps of the time domain. Yet, for discrete time domains such a representation is still possible. In this case a stream $s \in \mathcal{S}_{\mathbb{D}}^{\mathbb{T}}$ is encoded by the trace $w = \langle\ (s(t_1),\ s(t_2),\ s(t_3)\ \ldots\ \rangle \in \mathbb{D}^{\infty}$ or $w = \langle\ ((t_1, s(t_1)),\ (t_2, s(t_2)),\ (t_3, s(t_3))\ \ldots\ \rangle \in (\mathbb{T} \times \mathbb{D})^{\infty}$ (called *timed trace*) where t_i is the i^{th} smallest timestamp in $\mathbb{T}$[1].

Formalisms that support timed streams with non-discrete time domain (such as MTL [Koy90], MITL [AFH91], STL [MN04] or TeSSLa [CHL+18]) however face the problem of how to handle time domains and corresponding streams with a non-denumerable number of timestamps on discrete computing machinery. A common approach is to compute on timed traces that contain values for only a subset of timestamps, which are then interpreted as a timed stream, i.e. $w = \langle\ (t_0\ , s(t_0)),\ (t_1,\ s(t_1)),\ (t_2\ , s(t_2))\ \ldots\ \rangle \in (\mathbb{T} \times \mathbb{D})^{\infty}$ with $t_0 = 0$ and $\forall i.t_i < t_{i+1}$. There are several strategies for interpreting a timed trace $w = \langle\ (t_0,\ s(t_0)),\ (t_1,\ s(t_1)),\ (t_2,\ s(t_2))\ \ldots\rangle \in (\mathbb{T} \times \mathbb{D})^{\infty}$ as a timed stream over $\mathbb{T}$, e.g.

- ***as piece-wise constant timed stream:*** The timestamps for which no value is known are assumed to have the value of the last known timestamp. Thus w corresponds to stream $s \in \mathcal{S}_{\mathbb{D}}^{\mathbb{T}}$ with $s(t) = d$ s.t. w contains (t', d) for any $t' \leq t$, but not (t'', d') for any other $d' \neq d$ and $t' < t'' < t$.
- ***as timed stream with unknowns:*** The data domain $\mathbb{D}$ is extended by a special symbol ? (meaning no information available about the value) which is assigned to all timestamps not contained in w. Thus w corresponds to stream $s \in \mathcal{S}_{\mathbb{D}}^{\mathbb{T}}$ with $s(t) = d$ if (t, d) is contained in w and $s(t) = ?$ if (t, d) is not contained for any $d \in \mathbb{D}$.
- ***as timed stream with gaps:*** The data domain $\mathbb{D}$ is extended by a special symbol † (meaning no value at this timestamp) which is assigned to all timestamps not contained in w. Thus w corresponds to stream $s \in \mathcal{S}_{\mathbb{D}}^{\mathbb{T}}$ with $s(t) = d$ if (t, d) is contained in w and $s(t) = \dagger$ if (t, d) is not contained for any $d \in \mathbb{D}$.

The latter two interpretations are technically equivalent but differ in the semantic meaning of the ? and † events.

When using the encoding above for streams over non-discrete time domains, the events in the trace may accumulate before a particular timestamp. That is, the trace contains letters for an infinite number of timestamps that are smaller than a specific $t \in \mathbb{T}$, e.g.

$$w = (0, a), (0.9, b), (0.99, a), (0.999, b), (0.9999, a) \ldots$$

for $t = 1 \in \mathbb{T} = \mathbb{R}$. If this is the case, the timed trace will only encode the stream up to this position t, and no information for t and the timestamps after is revealed.

[1] The two given encodings are equivalent. Which one to use is a design choice, especially for non-regular time domains it might be preferable to include the current timestamp in the trace.

In particular, when computing such a trace, the system gets "stuck" at timestamp t and does not go beyond it. The described phenomenon is commonly known as *zenoness* or *zeno behavior* [ZJLS00], named after Zeno of Elea and is an obstacle when computing with timed streams over non-discrete time domains.

2.1.4. Ordered sets and lattices

Ordered sets with special properties are another key concept used in several places in this thesis, e.g. for the representation of the monitor's output verdicts. Therefore, some basic definitions from order theory are introduced below.

Definition 2.7 (Bounds, directed sets, meets and joins; based on [DP90])**.**

Let $(S, \sqsubseteq)$ be a non-empty partially ordered set.

For $S' \subseteq S$, an element $e \in S$ is called an *upper bound* of S' if $s \sqsubseteq e$ for all $s \in S'$ and a *lower bound* of S' if $e \sqsubseteq s$ for all $s \in S'$. A non-empty subset $S' \subseteq S$ is called *upward-directed* if every finite $F \subseteq S'$ has an upper bound in S' and *downward-directed* if every finite $F \subseteq S'$ has a lower bound in S'.

An upper bound $j \in S$ of $S' \subseteq S$ is called *least upper bound* or *join* or *supremum* of S' (denoted $\sqcup^S S'$) if and only if for all upper bounds u of S' , $j \sqsubseteq u$ holds. A lower bound $m \in S$ of $S' \subseteq S$ is called *greatest lower bound* or *meet* or *infimum* of S' (denoted $\sqcap^S S'$) if and only if for all lower bounds l of S' , $l \sqsubseteq m$ holds.

Note that for $S' = \emptyset$ any element $b \in S$ is an upper and lower bound. For a partially ordered set $(S, \sqsubseteq)$ we use the notation $a \sqsubset b$ for $a, b \in S$ (or an analogous representation for other relation symbols) to denote that $a \sqsubseteq b$ holds but not $b \sqsubseteq a$, i.e. that a is strictly smaller than b. If the base set is clear from the context we denote $\sqcup$ and $\sqcap$ without superscript to refer to the corresponding meets and joins. For the meet or join of two elements we also use the infix notation, i.e. $a \sqcup^S b$ or $a \sqcap^S b$. For partially ordered sets $(S, \sqsubseteq)$ where $\sqsubseteq$ is clear from the context we just write S to denote the ordered set.

Based on the upper definitions we now introduce the concept of a (complete, semi-) lattice, which is an ordered set, where all pairs or subsets have meets and joins:

Definition 2.8 (Lattices; based on [DP90])**.**

Let $(S, \sqsubseteq)$ be a non-empty partially ordered set.

$(S, \sqsubseteq)$ is a *join semi-lattice* if for all $a, b \in S$, $a \sqcup^S b$ exists and a *complete join semi-lattice* if for all $S' \subseteq S$, $\sqcup^S S'$ exists.

$(S, \sqsubseteq)$ is a *meet semi-lattice* if for all $a, b \in S$, $a \sqcap^S b$ exists and a *complete meet semi-lattice* if for all $S' \subseteq S$, $\sqcap^S S'$ exists.

> If $(S, \sqsubseteq)$ is a meet and join semi-lattice it is called a *lattice* and a *complete lattice* if it is a complete meet and join semi-lattice.

A complete lattice S always contains a greatest and least element which we denote with $\top^S$ and $\bot^S$ in the following.

2.1.5. Algebras

This last part of the basic concepts section contains fundamental definitions related to algebras, which will play an important role in chapter 5 of this work. We begin with a definition of signatures.

> **Definition 2.9** (Signature; based on [EM85])**.**
>
> A *signature* is a pair $S = (\mathbb{S}, \mathbb{O})$ consisting of
>
> - a set $\mathbb{S}$ of *sorts* (also called *types*) and
> - a set $\mathbb{O} = \{K_s \mid s \in \mathbb{S}\} \cup \{OP_{w,s} \mid s \in \mathbb{S}, w \in \mathbb{S}^+\}$ of constant and operation symbols
>
> s.t. K_s contains all constant symbols of sort s and $OP_{w,s}$ all operation symbols of signature $w \to s$ and all sets are pairwise disjoint.

The interpretation of a signature $S = (\mathbb{S}, \mathbb{O})$ is given by an algebra which assigns concrete domains, values and operations to the elements in $\mathbb{S}$ and $\mathbb{O}$.

> **Definition 2.10** (Algebra; based on [EM85])**.**
>
> Let $S = (\mathbb{S}, \mathbb{O})$ be a signature.
>
> An *algebra of* S (also called S*-algebra* or S*-structure*) is a pair $\mathfrak{A} = (\mathfrak{S}, \mathfrak{O})$ consisting of
>
> - a family $\mathfrak{S} = (\mathfrak{S}_s)_{s \in \mathbb{S}}$ of *domains* or *base sets* for all sorts in $\mathbb{S}$,
> - a family $\mathfrak{O} = (\mathfrak{O}_o)_{o \in \mathbb{O}}$ of *constants* and *operations* for all symbols in $\mathbb{O}$, s.t. $\mathfrak{O}_o$ has the sort or function signature of o.

Throughout this thesis we assume that the signature is implicitly given along with an algebra. Therefore we sometimes also use the symbol for the algebra in place of the signature. Furthermore, we require for all utilized algebras in this thesis that they contain the boolean domain $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$ with the corresponding constants `true` and `false`, the operation $\wedge$ for conjunction and that each domain comes with the operator $\cdot = \cdot$ for checking the equality of two elements.

Over a given signature and variable identifiers, expressions can be built.

Definition 2.11 (Expression; based on [EM85])**.**

Let $S = (\mathbb{S}, \mathbb{O})$ be a signature and $\mathbb{V} = (\mathbb{V}^s)_{s \in \mathbb{S}}$ a family of disjoint *variable identifiers*. Let further K_s be the set of constant symbols of sort $s \in S$ and $OP_{w,s}$ the set of operation symbols of signature $w \to s$ in $\mathbb{O}$.

The set of *expressions (also called terms) over S and $\mathbb{V}$ of sort s* ($\mathbb{E}^s_{S,\mathbb{V}}$) is the smallest set s.t.

- $\mathbb{V}^s \subseteq \mathbb{E}^s_{S,\mathbb{V}}$ and $K_s \subseteq \mathbb{E}^s_{S,\mathbb{V}}$
- $f(t_1, \ldots, t_n) \in \mathbb{E}^s_{S,\mathbb{V}}$ with $f \in OP_{(s_1,\ldots,s_n),s}$, $t_i \in \mathbb{E}^{s_i}_{S,\mathbb{V}}$ for all $i \in \{1, \ldots, n\}$.

With $\mathbb{E}_{S,\mathbb{V}} = (\mathbb{E}^s_{S,\mathbb{V}})_{s \in \mathbb{S}}$ the family of expressions over the signature is denoted.

Given an algebra corresponding to the signature and an assignment of the variables, such expressions can be evaluated.

Definition 2.12 ((Extended) assignment; based on [EM85])**.**

Let $\mathfrak{A} = (\mathfrak{S}, \mathfrak{O})$ be an algebra of signature $S = (\mathbb{S}, \mathbb{O})$ and $\mathbb{V} = (\mathbb{V}^s)_{s \in \mathbb{S}}$ a family of disjoint variable identifiers.

An *assignment (of $\mathbb{V}$ over $\mathfrak{A}$)* is a function $val : \mathbb{V} \to \mathfrak{S}$ that assigns to every $v \in \mathbb{V}^s$ a value of the corresponding sort from $\mathfrak{S}_s$.

The *extended assignment (also called evaluation) of an expression $e \in \mathbb{E}_{S,\mathbb{V}}$ under assignment val*, $[e]_{val}$, is recursively defined as

- $[v]_{val} = val(v)$
- $[c]_{val} = \mathfrak{O}_c$
- $[f(e_1, \ldots, e_n)]_{val} = \mathfrak{O}_f([e_1]_{val}, \ldots, [e_n]_{val})$

for $v \in \mathbb{V}, c, f \in \mathbb{O}$ and $e_1, \ldots, e_n \in \mathbb{E}_{S,\mathbb{V}}$.

Note: If an algebra contains the sort $\mathbb{B}$ and operations $\neg$ and $\wedge$, then its expressions of sort $\mathbb{B}$ form a quantifier-free fragment of first-order logic. Allowing the usage of an existential quantifier ($\exists$) would consequently lift the S-algebra to first-order logic over signature S.

We now introduce some additional notations for expressions of type boolean.

For $e \in \mathbb{E}^{\mathbb{B}}_{S,\mathbb{V}}$ and a given S-algebra $\mathfrak{A} = (\mathfrak{S}, \mathfrak{O})$ we write $val \models e$ (val models e) if and only if $[e]_{val} = \texttt{true}$ and define the *set of models of e* as

$$[\![e]\!] = \{val \in (\mathbb{V} \to \mathfrak{S}) \mid val \models e\}.$$

Let $R = \{r_1, \ldots, r_n\} \subseteq \mathbb{V}$ be a subset of variables. The *set of models of e restricted to R* is given as

$$[\![e]\!]_R = \{(v_1, \ldots, v_n) \mid val \in [\![e]\!], \forall i \in \{1, \ldots, n\}.\, val(r_i) = v_i\}.$$

For $e_1, e_2 \in \mathbb{E}^{\mathbb{B}}_{S,\mathbb{V}}$ we write

$$\begin{aligned} e_1 \models e_2 \quad &\text{iff} \quad [\![e_1]\!] \subseteq [\![e_2]\!], \\ e_1 \equiv e_2 \quad &\text{iff} \quad [\![e_1]\!] = [\![e_2]\!], \\ e_1 \models_R e_2 \quad &\text{iff} \quad [\![e_1]\!]_R \subseteq [\![e_2]\!]_R, \\ e_1 \equiv_R e_2 \quad &\text{iff} \quad [\![e_1]\!]_R = [\![e_2]\!]_R. \end{aligned}$$

Thus, $e_1 \models e_2$ ($e_1 \equiv e_2$) holds if e_1 has a subset of the models of (the same models as) e_2. I.e. $e_1 \models e_2$ indicates that e_1 semantically implies e_2; $e_1 \equiv e_2$ means that e_1 and e_2 are semantically equivalent. The relation $e_1 \models_R e_2$ ($e_1 \equiv_R e_2$) holds if e_1 has a subset of the models of (the same models as) e_2, but only variables in R are considered.

We call a set of boolean expressions over some signature S, $\mathcal{C} \subseteq \mathbb{E}^{\mathbb{B}}_{S,\mathbb{V}}$, a *constraint set* and $\gamma = \bigwedge_{c \in \mathcal{C}} c$ its *term representation*. We use the relations $\models$, $\equiv$, $\models_R$, $\equiv_R$ for a pair of constraint sets, if and only if they hold for their term representations.

In the subsequent chapters we will pay special attention to the following algebras:

- ***Boolean algebra***
  - Domains: $\mathbb{B}$
  - Constants: $\texttt{true}, \texttt{false}$
  - Operators: $=, \wedge, \neg$
  - Syntactic sugar: $\vee, \rightarrow, \leftrightarrow$
- ***Linear algebra***
  - Domains: $\mathbb{B}, \mathbb{R}$
  - Constants: $\texttt{true}, \texttt{false}$, all $r \in \mathbb{R}$
  - Operators: $=$, $\wedge$, $+$, $r\cdot$ for all $r \in \mathbb{R}$
- ***Linear real arithmetic***
  - Domains: $\mathbb{B}, \mathbb{R}$
  - Constants: $\texttt{true}, \texttt{false}$, all $r \in \mathbb{R}$
  - Operators: $=$, $\wedge$, $\neg$, $<$, $+$, $r\cdot$ for all $r \in \mathbb{R}$
  - Syntactic sugar: $\vee, \rightarrow, \leftrightarrow, \geq, \leq, >$

The constants and operations have the common semantics for domains $\mathbb{R}$ and $\mathbb{B}$. Note that linear algebra and linear real arithmetic do not contain binary multiplication operators but only unary ones for each constant. This way it is not possible to build non-linear expressions in these algebras, i.e. such where a variable is multiplied with a variable. The syntactic sugar operators are defined as usual: $\alpha \vee \beta := \neg(\neg\alpha \wedge \neg\beta)$, $\alpha \rightarrow \beta := \neg\alpha \vee \beta$, $\alpha \leftrightarrow \beta := (\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$, $\alpha \geq \beta := \neg(\alpha < \beta)$, $\alpha \leq \beta := (\alpha < \beta) \vee (\alpha = \beta)$, $\alpha > \beta := \neg(\alpha \leq \beta)$.

2.2. Runtime verification

Based on the formalisms introduced in the previous section, we will now discuss some fundamental aspects of runtime verification.

At its core, *runtime verification (RV)* deals with the synthesis of a monitor from a formally specified property, which evaluates this property on a run of the supervised system. Thereby the run is usually given as a finite or infinite trace, stream or signal which is fed to the monitor.

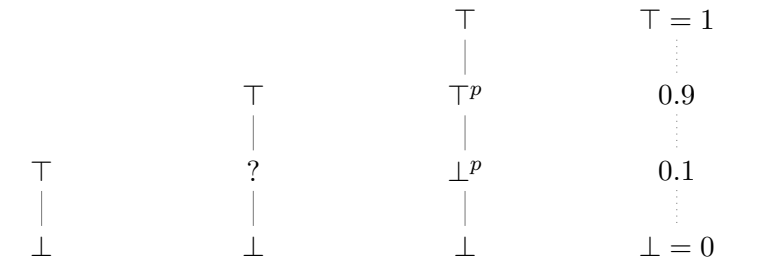
One generally distinguishes between *offline runtime verification*, where the monitor receives the whole recorded run at once (e.g. from a log file) and subsequently determines the monitoring outputs, and *online runtime verification*, where the monitor receives the run incrementally, e.g. letter by letter or event by event, and simultaneously yields outputs. In this thesis we focus on the latter one.

Traditional RV deals with solving the word problem [LS09]: The runs are finite or infinite words over letters from a finite alphabet (usually sets of atomic propositions). The property being checked is a *correctness property*, i.e. one that is either satisfied for the run, or not, but assigns no quantitative measure to it. For a correctness property φ in some specification language and $w \in \Sigma^*$ or $w \in \Sigma^\omega$, depending on whether the semantics of the formalism is defined on finite (e.g. RegEx, FLTL [Leu11]) or infinite (e.g. LTL [Pnu77]) words, we write $w \models \varphi$ if and only if φ is satisfied for w and $w \not\models \varphi$ if not. A correctness property φ defines the language $\mathcal{L}_\varphi = \{w \mid w \models \varphi\}$ of traces satisfied by φ.

In this traditional, most basic setting an online monitor M_φ, generated from the specification φ, yields a sequence of *verdicts*. They indicate whether for the whole run w, $w \in \mathcal{L}_\varphi$ holds or not, depending on the prefix of w received so far. For this purpose the verdict domain $\mathbb{V}$ has to contain at least two conclusive (aka final) verdicts, $\top \in \mathbb{V}$ indicating satisfaction of the property φ for w (i.e. $w \in \mathcal{L}_\varphi$) and $\bot \in \mathbb{V}$ indicating violation of φ (i.e. $w \notin \mathcal{L}_\varphi$). Besides that, the domain may also contain further non-conclusive verdicts, which indicate that $w \in \mathcal{L}_\varphi$ cannot yet be finally decided with the received prefix of w [LS09]. For a clear order of these verdicts, it is usually required that $\mathbb{V}$ is a lattice with greatest element $\top$ and least element $\bot$ [LS09]. In this setting, a monitor M_φ is characterized by its output

function $M_\varphi : \Sigma^* \to \mathbb{V}$ which assigns a verdict from domain $\mathbb{V}$ to every received prefix of w. Thus, for a run $w \in \Sigma^\infty$, M_φ receives the input letter by letter and produces the verdict sequence $M_\varphi(w^{(1)}), M_\varphi(w^{(2)}), \ldots$.

Four common verdict domains are depicted as Hasse diagrams in figure 2.1 [Leu11]. A dashed line between a lower element a and a higher element b symbolizes $a \sqsubseteq b$. A solid line additionally enforces that there is no $c \notin \{a, b\}$ s.t. $a \sqsubseteq c \sqsubseteq b$. The most simple domain $\mathbb{B}_2$ only contains the mandatory elements, $\top$ and $\bot$. The domain $\mathbb{B}_3$ additionally contains the "don't know" verdict ?. In $\mathbb{B}_4$ the ? is further subdivided into $\top^p$ and $\bot^p$ which mean presumably true or false. $\mathbb{P}$ finally is the domain of real numbers between 0 and 1 with their standard ordering, representing percentages of fulfillment and violation of the correctness property. In this domain $\top$ is an alias for $100\% = 1$ and $\bot$ for $0\% = 0$.



(a) $\mathbb{B}_2$ (b) $\mathbb{B}_3$ (c) $\mathbb{B}_4$ (d) $\mathbb{P} = ([0, 1], \leq)$

Figure 2.1.: Hasse diagrams of common verdict domains used in runtime verification.

Concerning the quality of an RV monitor, there are three characteristics usually considered desirable:

- ***Impartiality* [LS09]:** Whether a correctness property is satisfied or unsatisfied by the current run can be conclusively decided when the full trace is available. However, as long as only a prefix of the run is received by the monitor, the property may be inconclusive, i.e. the satisfaction depends on the future continuation of the trace. Consider for example the property "Atomic proposition p does not hold in the trace". It is unsatisfied if the monitor has already received an input where p did hold. It is satisfied if the end of the trace is reached and no input where p holds has been received. Otherwise it is inconclusive. Impartiality means a monitor may not be in favour of one of the final verdicts $\top$ or $\bot$, as long as it is not inevitable that the property is satisfied or breached. Hence as long as the observed property is not known to be (un-)satisfied for the full trace, an impartial monitor may not yield the verdicts $\top$ or $\bot$, but only other verdicts from the lattice like $\top^p$ (presumably true) or ? (unknown) or a percentage based on the usual behavior of the system. Note that in general a monitor with verdict domain $\mathbb{B}_2$ cannot be impartial, as it is forced to cast a final verdict in every step [Leu11].

- ***Anticipation* [LS09]:** Besides impartiality it is also desirable for an RV monitor to cast a final verdict as soon as possible. This means that if for all continuations of the currently received prefix the observed property is satisfied or unsatisfied, the monitor should directly cast the final verdict $\top$ or $\bot$. An RV monitor that meets this requirement is called anticipatory. The property is desirable because otherwise the monitor could continuously yield an inconclusive verdict even though a conclusive one would be possible.
- ***Trace-length-independence* [BKV13]:** This property requires the monitor's memory and runtime bounds to be independent of the length of the received trace. This means the computation time and memory per received input may not increase with the number of received observations. The property is necessary as monitors are intended to potentially run forever if they observe permanently running systems, like servers. If the need for resources grew with the runtime of the monitor it would eventually run out of resources and crash or get unresponsive. As a consequence a monitor must be able to condense the trace that has been received so far to a finite essence to be trace-length-independent.

A formal definition of these criteria can be found in definition 2.13. Therefore, for a given monitor M over input domain Σ, we use $M_M : \mathbb{N} \to \mathbb{N}$ to denote the maximal memory requirement of the monitor, s.t. $M_M(n)$ is the maximal amount of memory needed for output generation of $M(w)$ for any monitor input $w \in \Sigma^n$. Further we use $T_M : \mathbb{N} \to \mathbb{N}$ for the maximal runtime requirement: With T_M we denote the maximal number of computation steps between receiving the last letter of w and generating the output $M(w)$ for any $w \in \Sigma^n$.

Definition 2.13 (Monitor characteristics; based on [LS09, BLS10, BKV13])**.**

Let M be a monitor with the characterizing function $M : \Sigma^* \to \mathbb{V}$ and $\top, \bot \in \mathbb{V}$ the two final verdicts in $\mathbb{V}$. Let $\mathcal{L}_\varphi \subseteq \Omega$ with $\Omega \in \{\Sigma^*, \Sigma^\omega, \Sigma^\infty\}$ be the language for which M shall decide language containment.

M is called

- *impartial* if for any $w \in \Sigma^*$

  $(M(w) = \top \Rightarrow \forall wv \in \Omega.\, wv \in \mathcal{L}_\varphi)$ and $(M(w) = \bot \Rightarrow \forall wv \in \Omega.\, wv \notin \mathcal{L}_\varphi)$

- *anticipatory* if for any $w \in \Sigma^*$

  $(\forall wv \in \Omega.\, wv \in \mathcal{L}_\varphi \Rightarrow M(w) = \top)$ and $(\forall wv \in \Omega.\, wv \notin \mathcal{L}_\varphi \Rightarrow M(w) = \bot)$

- *trace-length-independent* if $T_M \in \mathcal{O}(1)$ and $M_M \in \mathcal{O}(1)$.

Note that the notion of impartiality and anticipation from definition 2.13 is only applicable if the monitor answers a boolean language containment problem and the output domain contains final verdicts $\top$ and $\bot$. However, advanced approaches in

runtime verification, like stream runtime verification, are able to monitor properties different from language containment, especially non-boolean properties, e.g. for calculating some numerical metrics on the system run. Further the output domains of such monitors do not necessarily have lattice structure. Thus, these monitoring approaches cannot simply be classified as impartial or anticipatory in terms of the upper definition. In chapter 3 of this thesis we will discuss a generalization of the criteria from definition 2.13 for general RV approaches.

In the remainder of this section the most popular RV formalism, linear temporal logic, and its extensions metric (interval) and signal temporal logic will be introduced formally. Additionally some corresponding monitor constructions will be presented. The subsequent section then deals with the field of stream runtime verification, its concepts, languages and monitoring algorithms.

2.2.1. Linear temporal logic

Conditioned by the historical impact of model checking, where *linear temporal logic (LTL)* plays a major role, it has also established as one of the first and best-studied RV formalisms.

LTL belongs to the family of temporal logics, which originate in the field of philosophy and have increasingly been studied there from the 1950s on [Øhr19, ØH07, Pri58]. Amir Pnueli was the first who suggested the use of such logics to specify correctness properties for computer programs. Based on the tense logic fragment K_b [RU71], he outlined in his 1977 work [Pnu77] how reasoning about these temporal properties can be used for program verification and proposed this technique as a unified approach for this purpose. The variant used there only contained two time-related operators called "globally" and "finally", and was later extended by further ones to the linear temporal logic used today [MP79, GPSS80]. While LTL was originally defined with future operators only, there are also variants where past operators are considered part of LTL [LPZ85, MP92]. Definition 2.14 defines the syntax of the full (past and future) version of linear temporal logic used in this thesis. Later in definition 2.17, syntactic sub-fragments of this full LTL are specified, which will also play a role throughout the rest of this work.

Definition 2.14 (LTL syntax; based on [LPZ85, MP92])**.**

The set of *linear temporal logic (LTL)* formulas over a finite set of atomic propositions $\mathcal{AP}$, $\text{LTL}_{\mathcal{AP}}$, is given by the following grammar:

$$\varphi ::= \texttt{true} \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathcal{X}\varphi \mid \varphi\,\mathcal{U}\,\varphi \mid \mathcal{P}\varphi \mid \varphi\,\mathcal{S}\,\varphi$$

where $p \in \mathcal{AP}$ is an atomic proposition.

Thus, LTL is an extension of standard propositional logic by the operators $\mathcal{X}$ (next), $\mathcal{U}$ (until), $\mathcal{P}$ (previously) and $\mathcal{S}$ (since). In cases where it is clear from the context we skip the $\mathcal{AP}$ in the index and just write LTL for the set of all LTL formulas. LTL is interpreted over structures (finite or infinite words) of sets of atomic propositions. In the following let $\Sigma = 2^{\mathcal{AP}}$. We define two versions of LTL semantics. The first one (definition 2.15), called *pointed semantics*, assigns a truth value to an LTL formula and a pointed word, i.e. a word and a given position. The second one (definition 2.16) is the more common and original LTL semantics. It is called *initial semantics* and its definition is based on the pointed semantics. It assigns a truth value to an LTL formula and a given word without a dedicated position.

Definition 2.15 (Pointed LTL semantics; based on[MP92, LPZ85, KLSS22])**.**

Let $\varphi \in \text{LTL}$ be an LTL formula, $w \in \Sigma^\infty$ a (possibly infinite) word and $t \in \mathbb{N}$ a position. The model relation $(w,t) \models \varphi$, which indicates that φ is satisfied at position t in w, is inductively defined as follows:

For $t \geq |w|$:

$$(w,t) \not\models \varphi$$

For $t < |w|$:

$$\begin{aligned}
&(w,t) \models \texttt{true} \\
&(w,t) \models p && \text{iff} \quad p \in w(t) \\
&(w,t) \models \neg\varphi && \text{iff} \quad (w,t) \not\models \varphi \\
&(w,t) \models \varphi_1 \wedge \varphi_2 && \text{iff} \quad (w,t) \models \varphi_1 \text{ and } (w,t) \models \varphi_2 \\
&(w,t) \models \mathcal{X}\varphi && \text{iff} \quad (w,t+1) \models \varphi \\
&(w,t) \models \varphi_1 \,\mathcal{U}\, \varphi_2 && \text{iff} \quad \text{for some } t' \geq t : (w,t') \models \varphi_2 \text{ and} \\
&&& \qquad \text{for all } t \leq t'' < t' : (w,t'') \models \varphi_1 \\
&(w,t) \models \mathcal{P}\varphi && \text{iff} \quad t > 0 \text{ and } (w,t-1) \models \varphi \\
&(w,t) \models \varphi_1 \,\mathcal{S}\, \varphi_2 && \text{iff} \quad \text{for some } 0 \leq t' \leq t : (w,t') \models \varphi_2 \text{ and} \\
&&& \qquad \text{for all } t' < t'' \leq t : (w,t'') \models \varphi_1
\end{aligned}$$

No property is satisfied at positions which are beyond the length of w. The predicate formulas (those consisting solely of `true`, atomic propositions, $\neg$, $\wedge$) are evaluated exclusively over the letter at position t, which is interpreted as the set of atomic propositions holding at this position. The next operator $\mathcal{X}$ demands that a property holds at the next position in the word. It is not satisfied at the end of the trace, where no next position exists. Likewise $\mathcal{P}$ (previous) requires a property to hold at the previous position. If there is no previous position because the formula is evaluated at position 0 it is not satisfied. $\varphi_1 \,\mathcal{U}\, \varphi_2$ is satisfied if and only if φ_2 holds for some letter in the future or at the current position and φ_1 holds for all letters

from position t up to there. $\varphi_1 \mathcal{S} \varphi_2$ on the other hand requires φ_2 to hold somewhere in the past including the current position t, and φ_1 at every position from there up to position t.

The second semantics, the initial LTL semantics, can directly be derived from the pointed semantics.

Definition 2.16 (Initial LTL semantics; based on [MP92, KLSS22])**.**

Let $\varphi \in \text{LTL}$ be an LTL formula and $w \in \Sigma^\infty$ a (possibly infinite) word. The initial semantics of φ, is given as:

$$w \models \varphi \quad \text{iff} \quad (w, 0) \models \varphi$$

Hence, in the initial semantics, an LTL formula is always evaluated for the first position in the word, i.e. from the beginning on.

Note that in literature the LTL semantics are usually restricted to either infinite structures (i.e. Σ^ω words) or finite ones (i.e. Σ^* words, often referred to as finite LTL). Our semantics from definition 2.15 and definition 2.16 is capable of both, yet throughout the thesis we will sometimes also consider only finite or infinite words, which will then be clarified accordingly. For a formula $\varphi \in \text{LTL}$ we use

$$\mathcal{L}^*_\varphi = \{w \in \Sigma^* \mid w \models \varphi\}, \mathcal{L}^\omega_\varphi = \{w \in \Sigma^\omega \mid w \models \varphi\} \text{ and } \mathcal{L}^\infty_\varphi = \{w \in \Sigma^\infty \mid w \models \varphi\}.$$

Further, the attentive reader may already have noticed that some of the traditional LTL operators, like globally ($\mathcal{G}$) or finally ($\mathcal{F}$), were not contained in the LTL syntax definition (definition 2.14). This is because their semantics can be expressed with use of the other operators and they can hence be considered syntactic sugar. For convenience, following additional operators and symbols will be allowed:

$$\begin{aligned}
\texttt{false} &:= \neg\texttt{true} \\
\varphi_1 \vee \varphi_2 &:= \neg(\neg\varphi_1 \wedge \neg\varphi_2) \\
\varphi_1 \rightarrow \varphi_2 &:= \neg\varphi_1 \vee \varphi_2 \\
\varphi_1 \leftrightarrow \varphi_2 &:= (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1) \\
\mathcal{F}\varphi &:= \texttt{true}\, \mathcal{U}\, \varphi \\
\mathcal{G}\varphi &:= \neg\mathcal{F}(\neg\varphi) \\
\mathcal{O}\varphi &:= \texttt{true}\, \mathcal{S}\, \varphi \\
\mathcal{H}\varphi &:= \neg\mathcal{O}(\neg\varphi) \\
\varphi_1\, \mathcal{R}\, \varphi_2 &:= \neg((\neg\varphi_1)\, \mathcal{U}\, (\neg\varphi_2)) \\
\widetilde{\mathcal{X}}\varphi &:= \mathcal{X}\varphi \vee \neg\mathcal{X}\texttt{true} \\
\widetilde{\mathcal{P}}\varphi &:= \mathcal{P}\varphi \vee \neg\mathcal{P}\texttt{true}
\end{aligned}$$

Besides `false` and the common logical operators, there are seven additional temporal operators introduced: $\mathcal{F}$ (finally), $\mathcal{G}$ (globally), $\mathcal{O}$ (once), $\mathcal{H}$ (historically), $\mathcal{R}$ (release), $\widetilde{\mathcal{X}}$ (weak next) and $\widetilde{\mathcal{P}}$ (weak previous). Due to the semantics of $\mathcal{U}$, which requires the right hand side to hold somewhere in the trace, $\mathcal{F}\varphi$ ensures that φ holds at some future position or now. The formula $\mathcal{G}\varphi$ on the other hand is satisfied, if the inner formula φ holds at every future position and the current one. The operators once and historically build the past counterparts to finally and globally. While $\mathcal{O}\varphi$ evaluates to true if φ held at any position in the past or now, $\mathcal{H}\varphi$ requires that φ held at all positions in the past, including the current one. The formula $\varphi \,\mathcal{R}\, \psi$ requires ψ to hold from the current position until φ is satisfied or forever. Finally, the operators $\widetilde{\mathcal{X}}\varphi$ and $\widetilde{\mathcal{P}}\varphi$ evaluate to true when either the sub-formula is satisfied at the next/previous position, or if they are evaluated at the end/beginning of the trace. For infinite traces these operators are equivalent to the usual (strong) next and previous operators.

Throughout this thesis we will not always use the full LTL syntax, as given by definition 2.14. We will also investigate the following fragments which are restricted, in the sense that they can only reference future, past or past and a constant number of future positions, determined by the number of nested $\mathcal{X}$ in the formula.

Definition 2.17 (Linear temporal logic fragments; based on [KLSS22])**.**

The following syntactic fragments of LTL are defined:

- **Future LTL**: $\text{LTL}^{f}_{\mathcal{AP}}$ is the subset of $\text{LTL}_{\mathcal{AP}}$ that solely consists of LTL formulas without the operators $\mathcal{P}$ and $\mathcal{S}$.
- **Past LTL**: $\text{LTL}^{p}_{\mathcal{AP}}$ is the subset of $\text{LTL}_{\mathcal{AP}}$ that solely consists of LTL formulas without the operators $\mathcal{X}$ and $\mathcal{U}$.
- **Past LTL with bounded future**: $\text{LTL}^{bf}_{\mathcal{AP}}$ is the subset of $\text{LTL}_{\mathcal{AP}}$ that solely consists of LTL formulas without the operator $\mathcal{U}$.

Indeed for initial semantics and on infinite words, the future LTL fragment was shown to be as expressive as full LTL. I.e. every full LTL formula can be rewritten into a future LTL formula that is modeled by the same ω-words [GPSS80]. The size of these rewritten formulas however was proven to have an exponentially higher bound in general [Mar03]. Note that the finding about the same expressiveness also implies past LTL and past LTL with bounded future as subsets of full LTL to be expressible in future LTL. However this equivalence only holds for the initial LTL semantics, where the expressiveness of the past fragments is highly limited anyway. This is because from the the first position in the word there is no past to reason about. For pointed semantics, though, full LTL is clearly more expressive than past LTL, past LTL with bounded future, or future LTL, because in full LTL the value at a specific location can be influenced by letters from arbitrary previous and subsequent locations, which is not possible in the other fragments. Past LTL with bounded future in turn is also more expressive than past LTL in terms of the pointed semantics.

2.2.2. Monitor constructions

Due to its popularity as an RV formalism, several monitor constructions have been developed for linear temporal logic. The different techniques for monitor synthesis mainly differ in the LTL fragment which they support as well as in the verdict domain they use. In this section two popular constructions will be presented, as they will play a role in the following chapters. The first is called LTL_3 construction [BLS06a] and allows the synthesis of monitors for future LTL casting verdicts from the three-valued truth domain $\mathbb{B}_3$ (see figure 2.1). The second one is the standard approach for monitoring past LTL as presented in [HR02], where the resulting monitor yields verdicts from the $\mathbb{B}_2$ domain (see figure 2.1).

In both cases the monitor constructions yield a Moore machine [M$^+$56], i.e. a finite-state machine casting an output with each received piece of input, dependent on the current state of the machine. A formal definition of such a Moore machine is given in definition 2.18.

Definition 2.18 (Moore machine; based on [M$^+$56, HU79])**.**

A *Moore machine* $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, \gamma, q_I)$ is a 6-tuple consisting of

- a finite, non-empty set of states Q,
- a finite, non-empty input alphabet Σ,
- a finite, non-empty output alphabet Γ,
- a deterministic transition function $\delta : Q \times \Sigma \to Q$,
- an output function $\gamma : Q \to \Gamma$,
- an initial state $q_I \in Q$.

A sequence of states $q_0, q_1, \ldots, q_n$ with $q_i \in Q$ is called the *run* of $\mathcal{M}$ for $w \in \Sigma^*$ with $n = |w|$, if and only if

- $q_0 = q_I$ and
- $q_{i+1} = \delta(q_i, w(i))$ for all $i \in \{0, \ldots, n-1\}$

The corresponding sequence $\gamma(q_0), \gamma(q_1), \ldots, \gamma(q_n)$ is called *output* of $\mathcal{M}$.

Another popular machine model for monitors are *Mealy machines* [Mea55]. They differ from Moore machines in that the outputs are associated with transitions rather than states. However both models are equivalent and can easily be transformed into each other [HU79]. Yet, for the following constructions we will exclusively make use of Moore machines.

LTL_3 construction

The LTL_3 monitor construction is fundamentally based on the translation of an LTL formula into a corresponding non-deterministic Büchi automaton [Büc90]. Büchi

automata belong to the family of ω-automata as they describe languages (sets) of ω-words (i.e. infinite words). A nondeterministic Büchi automaton and its corresponding language are defined as follows.

Definition 2.19 (Nondeterministic Büchi automaton; based on [Büc90, HL11])**.**

A *nondeterministic Büchi automaton (NBA)* $\mathcal{A} = (Q, \Sigma, q_I, \delta, F)$ is a 5-tuple consisting of

- a finite, non-empty set of states Q,
- a finite, non-empty input alphabet Σ,
- an initial state $q_I \in Q$,
- a transition function $\delta : Q \times \Sigma \to 2^Q$,
- a set of accepting states $F \subseteq Q$.

An infinite sequence of states $\rho = q_0, q_1, \ldots$ with $q_i \in Q$ is called a *run* of $\mathcal{A}$ for $w \in \Sigma^\omega$, if and only if

- $q_0 = q_I$ and
- $q_{i+1} \in \delta(q_i, w(i))$ for all $i \in \mathbb{N}$

Let $\mathit{Inf}(\rho) \subseteq Q$ be the set of states which are contained infinitely many often in ρ. NBA $\mathcal{A}$ *accepts* $w \in \Sigma^\omega$ if and only if there is a run ρ for w such that $\mathit{Inf}(\rho) \cap F \neq \emptyset$.

The *language* of $\mathcal{A}$ is given as $\mathcal{L}_\mathcal{A} = \{w \in \Sigma^\omega \mid \mathcal{A} \text{ accepts } w\}$.

As known from other types of automata, there is also an alternating version of Büchi automata. While an NBA changes non-deterministically from the current state to the next one depending on the received input letter, an *alternating Büchi automata (ABA)* proceeds to a positive boolean combination (i.e. a propositional logic formula with conjunction, disjunction `true` and `false` but without negation) of successor states. Consequently a run of an ABA is not an infinite state sequence, but an infinitely deep tree, labeled with states, where the children of a node satisfy the corresponding positive boolean formula from the automaton's transition function. For states Q we formally describe such a labeled tree as triple (V, E, l), where (V, E) is a tree with infinitely many nodes and $l : V \to Q$ is a total function assigning a state from Q to every tree node. Further we denote the set of positive boolean formulas over states Q as $\mathbb{B}^+(Q)$. For $\psi \in \mathbb{B}^+(Q)$ and $S \subseteq Q$ we write $S \models \psi$ if and only if ψ evaluates to true with all atomic propositions in S set to `true` and all others to `false`. Definition 2.20 gives a formal definition of alternating Büchi automata and their acceptance criterion.

Definition 2.20 (Alternating Büchi automaton; based on [MH84, HL11])**.**

An *alternating Büchi automaton (ABA)* $\mathcal{A} = (Q, \Sigma, q_I, \delta, F)$ is a 5-tuple consisting of

- a finite, non-empty set of states Q,
- a finite, non-empty input alphabet Σ,
- an initial state $q_I \in Q$,
- a transition function $\delta : Q \times \Sigma \to \mathbb{B}^+(Q)$,
- a set of accepting states $F \subseteq Q$.

A Q-labeled infinite tree $\mathcal{T} = (V, E, l)$ with root $r \in V$ is called a *run* of $\mathcal{A}$ for $w \in \Sigma^\omega$, if and only if

- $l(r) = q_I$ and
- $\{l(v') \mid (v, v') \in E\} \models \delta(l(v), w(i))$ for all $v \in V$ with a distance of $i \in \mathbb{N}$ from r.

Let ρ be an infinite path in $\mathcal{T}$. $\mathit{Inf}(\rho) \subseteq Q$ denotes the set of states which are contained infinitely many often in ρ's labels. ABA $\mathcal{A}$ *accepts* $w \in \Sigma^\omega$ if and only if there is a run $\mathcal{T}$ for w, such that $\mathit{Inf}(\rho) \cap F \neq \emptyset$ for every infinite path in $\mathcal{T}$, starting from $\mathcal{T}$'s root.

The *language* of $\mathcal{A}$ is given as $\mathcal{L}_\mathcal{A} = \{w \in \Sigma^\omega \mid \mathcal{A} \text{ accepts } w\}$.

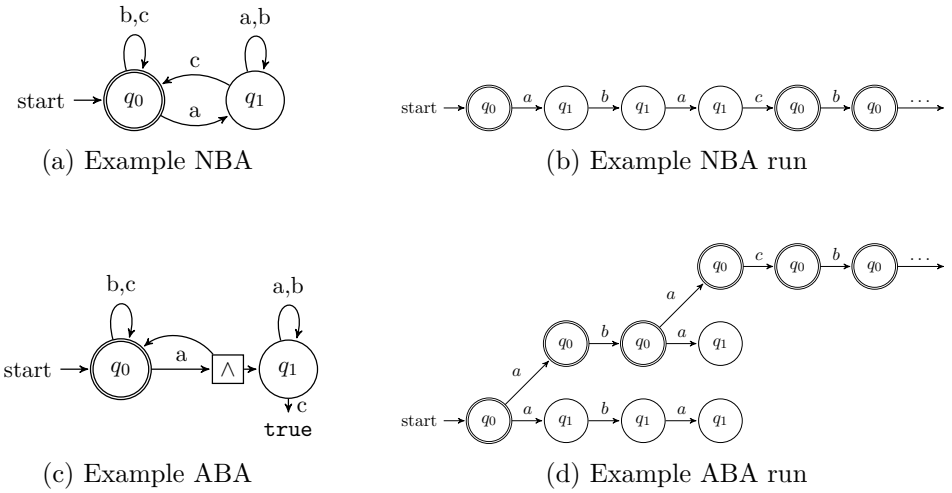
An example of a nondeterministic and an alternating Büchi automaton and a corresponding run for $w = abacb\ldots$ over $\Sigma = \{a, b, c\}$ can be found in figure 2.2. Accepting states are marked with a double circle around the state name. Transitions are depicted by arrows leading from one state to another. In the case of ABAs the positive boolean combination of successor states is expressed by combinations of $\wedge$ and $\vee$ operators inside square boxes. Alternatively the transitions may also lead to the positive boolean formulas `true` and `false`. The ABA from figure 2.2 for example contains the transition $\delta(q_0, a) = q_0 \wedge q_1$ and $\delta(q_1, c) =$ `true`.

The two automata from figure 2.2 have exactly the same language, namely

$$L = \{w \in \Sigma^\omega \mid \forall i.\, (w(i) = a) \to \exists j > i.\, (w(j) = c)\}.$$

Hence, they accept every infinite word where each a is proceeded by a c at a later position. Therefore the NBA remains in its initial and accepting state q_0 until it reads an a. Then it changes to non-accepting state q_1. It remains in this state q_1 as long as letter c is not received. Consequently, if every a is proceeded by a c, the NBA always returns to accepting state q_0, hence for the corresponding run ρ, $q_0 \in \mathit{Inf}(\rho)$ holds and the automaton accepts. If on the other hand there is an a without a c at a later position, the automaton remains forever in state q_1 from the last a without subsequent c on, and hence does not accept the word.

The ABA in contrast changes from initial and accepting state q_0 to $q_0 \wedge q_1$ whenever it reads the input letter a in state q_0. Further, it remains in non-accepting state q_1 until letter c is received; after that it requires no change to another state, as the corresponding positive boolean formula `true` is satisfied by the empty set of



(a) Example NBA (b) Example NBA run

(c) Example ABA (d) Example ABA run

Figure 2.2.: Example of NBA and ABA with corresponding runs for input word $w = abacb\ldots$.

successor states. Overall, if every a in the input trace is followed by a c, the run of the ABA contains only one infinite path starting from the root, which is the one labeled exclusively with q_0. All other paths are finite since they end at the positions where the c appears. Such a run is accepting. Contrary, if there is an a with no c after, the run tree contains another infinite path with a finite number of q_0 states in the beginning and then continues with an infinite sequence of q_1 states. Therefore such a run is non-accepting.

Indeed it is not a coincidence that for this language there is a corresponding alternating and nondeterministic Büchi automaton. In fact both automata types have the same expressiveness. While it is easy to see that an NBA can be transformed into an ABA by simply replacing the set of successor states in the transition function by a disjunction of those states, the converse is not so obvious. In 1983, Miyano and Hayashi presented a construction that transforms arbitrary alternating Büchi automata into equivalent nondeterministic ones [MH84] with a worst-case exponential blow-up of the state space. The idea behind this construction is that the NBA simulates the stages of the ABA's run tree and checks that on every branch infinitely many accepting states are traversed. The exponential bound of the NBA's state space was later also shown to be tight [BKR10].

The cause why Büchi automata play a role in LTL monitor construction, as well as in LTL model checking, is due to the fact, that for any future LTL formula φ an ABA can be constructed which accepts exactly the language $\mathcal{L}^{\omega}_{\varphi}$ (i.e. the set of infinite traces satisfying φ in terms of the initial semantics, see definition 2.16). However the opposite does not hold. There are ABAs with corresponding languages that cannot be expressed in LTL, i.e. alternating Büchi automata are more expressive than LTL [Wol81].

The translation scheme from LTL to Büchi automata, presented in the following (see [Var95]), is restricted to future LTL. However [GPSS80] showed that full LTL has the same expressiveness as future LTL w.r.t. the initial semantics. Hence a full LTL formula can be rewritten into an equivalent future LTL formula (i.e. one with the same models). The rewriting, though, causes an exponential blow-up of the formula's length. In [GO03] also a translation of full LTL to alternating two-way Büchi automata, which can in turn be translated to alternating and thus nondeterministic Büchi automata can be found.

For the following we assume the future LTL formula to be in *negation normal form (NNF)*. This normal form restricts the operators and constants to `true`, `false`, $\wedge$, $\vee$, $\neg$, $\mathcal{U}$, $\mathcal{R}$, $\mathcal{X}$ and only allows atoms to be negated, no other sub-formulas. One can easily see that every future LTL formula can be transformed to NNF by exchanging $\rightarrow$, $\leftrightarrow$, $\mathcal{G}$, $\mathcal{F}$ with the permitted operators according to the common rules and then moving all negations inwards with help of the following equivalence transformations:

- $\neg\neg\varphi \equiv \varphi$, $\neg\texttt{true} \equiv \texttt{false}$ and $\neg\texttt{false} \equiv \texttt{true}$
- $\neg(\varphi \wedge \psi) \equiv \neg\varphi \vee \neg\psi$ and $\neg(\varphi \vee \psi) \equiv \neg\varphi \wedge \neg\psi$
- $\neg(\varphi\,\mathcal{R}\,\psi) \equiv (\neg\varphi)\,\mathcal{U}\,(\neg\psi)$ and $\neg(\varphi\,\mathcal{U}\,\psi) \equiv (\neg\varphi)\,\mathcal{R}\,(\neg\psi)$
- $\neg\mathcal{X}(\varphi) \equiv \mathcal{X}(\neg\varphi)$

By repeated application of these rules a formula is received containing $\neg$ operators exclusively in front of the atomic propositions.

The basic idea behind the translation from future LTL to an ABA is the following: The states of the ABA express the sub-formulas of the original NNF LTL formula φ that still have to be satisfied from the current position on. Consequently the initial state of the automaton is φ itself. The transition function for input letter $l \in \Sigma = 2^{\mathcal{AP}}$ obeys to the following scheme:

- $\delta(\texttt{true}, l) = \texttt{true}$ and $\delta(\texttt{false}, l) = \texttt{false}$
- $\delta(\alpha \wedge \beta, l) = \delta(\alpha, l) \wedge \delta(\beta, l)$ and $\delta(\alpha \vee \beta, l) = \delta(\alpha, l) \vee \delta(\beta, l)$
- $\delta(a, l) = \begin{cases} \texttt{true} & \text{if } a \in l \\ \texttt{false} & \text{else} \end{cases}$ and $\delta(\neg a, l) = \begin{cases} \texttt{false} & \text{if } a \in l \\ \texttt{true} & \text{else} \end{cases}$ for $a \in \mathcal{AP}$
- $\delta(\mathcal{X}(\alpha), l) = \alpha$
- $\delta(\alpha\,\mathcal{U}\,\beta, l) = \delta(\beta, l) \vee (\delta(\alpha, l) \wedge \delta(\mathcal{X}(\alpha\,\mathcal{U}\,\beta), l))$
- $\delta(\alpha\,\mathcal{R}\,\beta, l) = \delta(\beta, l) \wedge (\delta(\alpha, l) \vee \delta(\mathcal{X}(\alpha\,\mathcal{R}\,\beta), l))$

The transition for the until and release operator are based on the equivalences $\alpha \mathcal{U} \beta = \beta \vee (\alpha \wedge \mathcal{X}(\alpha \,\mathcal{U}\, \beta))$ and $\alpha \,\mathcal{R}\, \beta = \beta \wedge (\alpha \vee \mathcal{X}(\alpha \,\mathcal{R}\, \beta))$. Note that δ applied to sub-expressions of φ again only produces positive boolean combinations of sub-expressions from φ, i.e. states of the ABA. Yet it remains open how the accepting states of the ABA are determined. Without accepting states the ABA does not pay heed to the fact that $\alpha \mathcal{R} \beta$ is satisfied if β but not α holds forever on. Consequently we add all states of form $\alpha \mathcal{R} \beta$ to the set of accepting states. For other (sub-)formulas the acceptance of the automaton is already correct.

The size of the ABA's state space, generated by the described construction, is equivalent to the number of sub-formulas of the initial formula φ without $\wedge$ and $\vee$ on the outermost level (as these operators are handled as positive boolean state combinations by the ABA internally) plus one for the state φ itself. As a consequence the automaton's state space is linear in terms of the LTL formula's length.

As an example take the LTL formula $\varphi = \mathcal{F}p \wedge \mathcal{G}\neg q$ over alphabet $\Sigma = 2^{\mathcal{AP}} = \{\emptyset, \{p\}, \{q\}, \{p, q\}\}$. A transformation to NNF yields:

$$\begin{aligned} \varphi &\equiv \mathcal{F}p \wedge \mathcal{G}\neg q \equiv \mathcal{F}p \wedge \neg\mathcal{F}q \equiv (\mathtt{true}\,\mathcal{U}\, p) \wedge \neg(\mathtt{true}\,\mathcal{U}\, q) \\ &\equiv (\mathtt{true}\,\mathcal{U}\, p) \wedge (\mathtt{false}\,\mathcal{R}\, \neg q) \end{aligned}$$

The ABA for φ resulting from the previously described construction is depicted in figure 2.3.

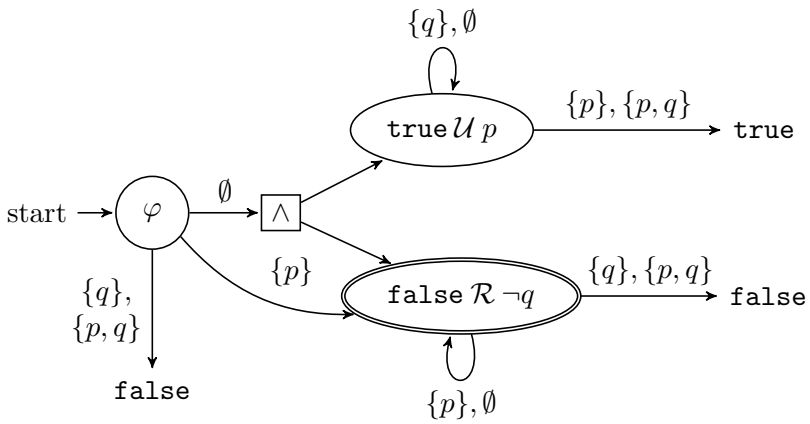


Figure 2.3.: ABA constructed from LTL formula $\varphi = \mathcal{F}p \wedge \mathcal{G}\neg q$.

The automaton already contains some basic simplifications of the transition function's positive boolean formulas. For example the transition from state $\mathtt{false}\,\mathcal{R}\,\neg q$ with letter $\{q\}$ or $\{p, q\}$ would lead to $\mathtt{false} \wedge (\mathtt{false} \vee (\mathtt{false}\,\mathcal{R}\,\neg q))$. However this formula cannot be satisfied by any set of successor states and hence the automaton directly contains the transition to `false`.

It is quite easy to convince oneself about the correctness of the automaton in figure 2.3. If in the initial state a letter with q is received, the automaton transitions

to $\texttt{false}$, as $\mathcal{G}\neg q \equiv (\texttt{false}\,\mathcal{R}\,\neg q)$ is eventually violated. If letter $\{p\}$ is received the $\mathcal{F}p \equiv (\texttt{true}\,\mathcal{U}\,p)$ sub-formula is already satisfied by this letter and the automaton proceeds to state $(\texttt{false}\,\mathcal{R}\,\neg q)$ to check if this part of φ is satisfied by the rest of the trace. Otherwise the automaton changes to a conjunction of both sub-formulas. State $(\texttt{false}\,\mathcal{R}\,\neg q)$ leads to acceptance if there is no q for the rest of the trace, state $(\texttt{true}\,\mathcal{U}\,p)$ if there is a p somewhere in the remaining trace.

The translation presented here shows quite intuitively how future LTL formulas can be translated to ABAs (and thus NBAs). Yet due to the practical relevance of this translation in runtime verification as well as in model checking, several other methods with partially better performance on practical examples have been proposed over time [Wol00, SB00, GO01, GL02].

We have now seen how LTL formulas can be translated to Büchi automata, which accept their language with respect to infinite words. However such Büchi automata are not suited for monitoring purposes. This is because in monitoring we aim at producing a sequence of verdicts for finite prefixes of a (potentially) infinite trace. Büchi automata however decide on infinite words. To overcome this discrepancy, in [BLS06a] Bauer et al. presented the so-called LTL_3 monitor construction, which – based on the Büchi automaton translation – yields a Moore machine that casts verdicts from the $\mathbb{B}_3 = \{\top, \bot, ?\}$ domain (see figure 2.1). The Moore machine constructed this way behaves as a trace-length-independent, anticipatory and impartial (future) LTL monitor according to the initial LTL semantics on infinite words (see definitions 2.13 and 2.16).

The idea behind the construction (depicted in figure 2.4) is to generate a nondeterministic Büchi automaton $\mathcal{A}^{\varphi}$ from the LTL formula φ and another one $\mathcal{A}^{\neg\varphi}$ from the negation of the formula and simplify them as far as possible. After that, so-called empty states are identified in both automata (denoted as $\mathcal{F}^{\varphi}, \mathcal{F}^{\neg\varphi}$ in figure 2.4). Empty states are those, from which the Büchi automaton cannot accept anymore, because they have no connection to a cycle with accepting states. They can be identified by a simple graph analysis (see [CVWY90]). After this, two nondeterministic finite automata $\hat{\mathcal{A}}^{\varphi}$ and $\hat{\mathcal{A}}^{\neg\varphi}$ are generated from $\mathcal{A}^{\varphi}$ and $\mathcal{A}^{\neg\varphi}$ by preserving state space, initial state and transition function and making all non-empty states accepting and the others non-accepting. The two NFAs are then determinized to receive $\tilde{\mathcal{A}}^{\varphi}$ and $\tilde{\mathcal{A}}^{\neg\varphi}$. Finally the Moore machine $\bar{\mathcal{A}}^{\varphi}$ is built from the product automaton of the two deterministic finite automata $\tilde{\mathcal{A}}^{\varphi}$ and $\tilde{\mathcal{A}}^{\neg\varphi}$. All states of $\bar{\mathcal{A}}^{\varphi}$ that correspond to a non-accepting state of $\tilde{\mathcal{A}}^{\neg\varphi}$ cast the output $\top$ (as there is no possibility to satisfy $\neg\varphi$, i.e. to violate φ anymore). Likewise all states that correspond to a non-accepting state of $\tilde{\mathcal{A}}^{\varphi}$ are linked to the output $\bot$. All other states (those corresponding to accepting states in $\tilde{\mathcal{A}}^{\varphi}$ and $\tilde{\mathcal{A}}^{\neg\varphi}$) yield the verdict ?. Note that states in the Moore machine which consist of non-accepting states in $\tilde{\mathcal{A}}^{\varphi}$ and $\tilde{\mathcal{A}}^{\neg\varphi}$ are not reachable, as the prefixes leading to them could neither be extended to satisfy φ nor $\neg\varphi$. However this is not possible as $\mathcal{L}^{\omega}_{\varphi}$ and $\mathcal{L}^{\omega}_{\neg\varphi}$ partition Σ^{ω}.

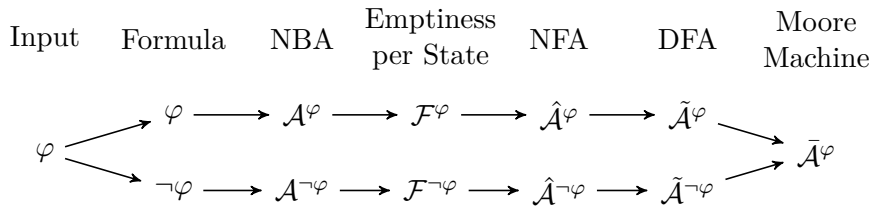


Figure 2.4.: Scheme of the LTL_3 monitor construction (figure from [BLS06a]).

A monitor resulting from the construction is impartial, anticipatory and trace-length-independent. The trace-length-independence follows from the finite number of states in the NBAs and consequently in the monitor. Since a state in the NBAs is marked as empty if and only if there is no more chance for the received prefix to lead to acceptance, the monitor switches to final verdict $\top$ or $\bot$ as soon as possible, but not earlier. This implies impartiality and anticipation.

An example of the LTL_3 monitor construction for the LTL formula $\varphi = \mathcal{F}p \wedge \mathcal{G}\neg q$ can be found in figure 2.5. As φ can ultimately be violated but not satisfied by some prefix due to the $\mathcal{G}\neg q$ part, the resulting monitor may cast the verdict ? (orange states) and $\bot$ (red states) but never $\top$. Also note that it is a coincidence that the two NBAs for φ and $\neg\varphi$ are equal except for the opposite set of accepting states. The cause for this is the special form of φ, but in general the NBA for a formula and its negation may be of totally different shape.

In [DLS08] the idea of this construction is generalized and shown to produce an anticipatory and impartial monitor for every logic for which a finite automaton construction with emptiness check exists. In this context, an emptiness check is defined as a function assigning to every state whether acceptance is still possible from there or not. The work also presents the application to some LTL extensions which fulfill this property. It should be noted at this point that the generalized LTL_3 construction there could in principle also be used for LTL monitoring over finite words. This only requires a translation of LTL for finite traces into corresponding finite automata instead of Büchi automata.

In [DLT14] Decker et al. present a further result that generalizes this kind of monitor construction. They assume an arbitrary temporal logic over atomic propositions and a corresponding monitor construction. Based on this, they show that for the extension of the logic, where the atomic propositions in formulas are replaced by first-order logical expressions, one is still able to construct a monitor. However, the resulting monitor is in general not anticipatory, even if the monitor construction of the underlying temporal logic is.

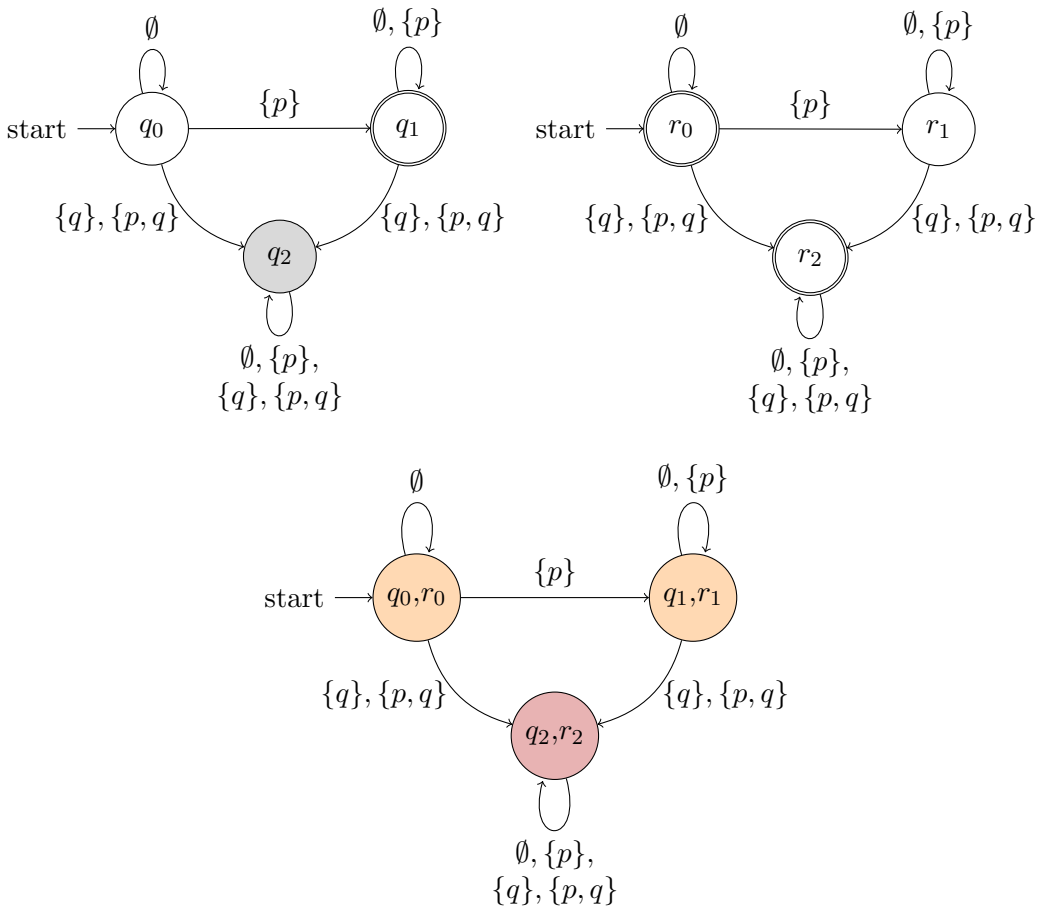


Figure 2.5.: Simplified NBAs for $\varphi = \mathcal{F}p \wedge \mathcal{G}\neg q$ (up left) and its negation (up right) with empty states marked gray and resulting LTL_3 monitor (below). Orange for ?, red for $\bot$ output.

Past LTL monitor construction

In 2002 Havelund and Roşu investigated the monitoring of the past-only fragment of LTL (past LTL) [HR02]. As mentioned earlier, it makes little sense to evaluate this fragment in terms of the initial semantics (definition 2.16), as one could only refer to the first position of the word due to the lack of future operators. Thus they developed a monitoring approach based on the pointed semantics (definition 2.15), which in construction and properties differs quite heavily from initial monitors, like the LTL_3 monitor from the previous section.

After desugaring, a past LTL formula may only contain the temporal operators $\mathcal{P}$ and $\mathcal{S}$ (note that in [HR02] a slightly different set of equivalent operators is used). The current value of $\mathcal{P}\varphi$ can be determined by evaluation of φ in the previous step

and memorizing the result (at the beginning of the trace it is simply `false`). For $\varphi \mathcal{S} \psi$ note the following equivalence:

$$\varphi \mathcal{S} \psi \equiv \psi \vee (\varphi \wedge \mathcal{P}(\varphi \mathcal{S} \psi))$$

Hence the current value of $\varphi \mathcal{S} \psi$ can be determined by the evaluation of φ and ψ and the value of $\mathcal{P}(\varphi \mathcal{S} \psi)$, i.e. the previous value of $\varphi \mathcal{S} \psi$. Altogether the value of all past operators can be computed based on the current input and the previous values of their sub-expressions including themselves. The monitoring algorithm presented in [HR02], formalized in an imperative language, makes use of this observation. It keeps an array data structure internally which assigns a truth value to all sub-expressions of the formula φ to be monitored. If for example $\varphi = \mathcal{P}((\neg p) \mathcal{S} p)$ the array would contain entries for the sub-formulas p, $\neg p$, $(\neg p) \mathcal{S} p$, and $\mathcal{P}((\neg p) \mathcal{S} p)$. Whenever a new input letter is received, the old array is backuped, i.e. copied in the memory. The new values of the sub-formulas are then successively determined from inside to outside based on the current input letter and – as described above – on the previous values from the old array. When the new array is fully determined the old one can be discarded and the output $\top$ or $\bot$ can be cast, according to the valuation of φ in the array.

As the state-space of this algorithm is finite (two boolean arrays of fixed size) it can likewise be understood as a Moore machine [KLSS22]: The states are boolean tuples assigning truth values to all sub-formulas plus one initial state where the automaton resides before receiving the first input letter. The transition function then results directly from the application of the procedure described above on a state with respect to a concrete input letter. The Moore machine finally outputs the verdict $\top$ in all states where the vector entry for the full formula is `true`, and $\bot$ in the other ones. The thus constructed monitor for the example formula $\mathcal{P}((\neg p) \mathcal{S} p)$ and input alphabet $\Sigma = 2^{\{p,q\}}$ is depicted in figure 2.6. The state names indicate the current valuation for the formulas p, $\neg p$, $(\neg p) \mathcal{S} p$, and $\mathcal{P}((\neg p) \mathcal{S} p)$ in this order. Number 1 stands for `true`, 0 for `false`. The grey states are those where the Moore machine casts the verdict $\top$, white states where $\bot$ is yielded as output.

Since the monitor has a finite state-space it may clearly be considered trace-length-independent. However, it is unclear whether the monitor is also impartial or anticipatory. This is because these terms, in their traditional definition (definition 2.13), do only apply to a setting where a monitor continuously tries to solve the same problem (if $w \in \mathcal{L}_\varphi$ holds for the full run w) and continuously receives more information. Here, however, a different question, namely if the formula is satisfied *at the current position* (pointed semantics), is conclusively decided in every step.

In this thesis we will further elaborate on this matter and discuss an extension of the traditional notions of impartiality and anticipation towards pointed semantics. Based on this, a monitoring technique for arbitrary properties on synchronous streams will be presented, which unifies the ideas of monitoring pointed semantics, as in the past LTL monitoring approach, and casting impartial and anticipatory

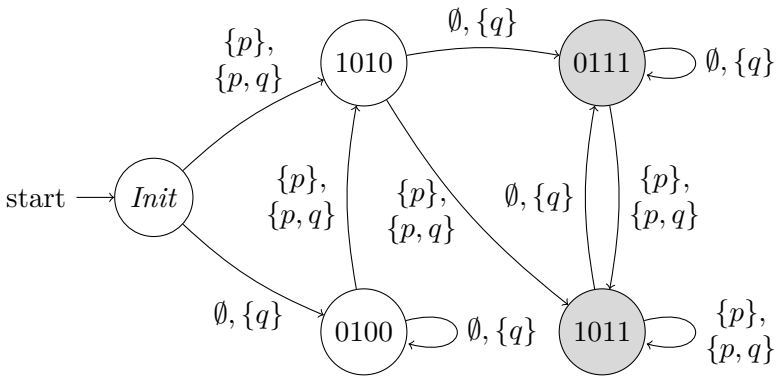


Figure 2.6.: Monitor resulting from past LTL construction for $\varphi = \mathcal{P}((\neg p)\ \mathcal{S}\ p)$. The state names indicate the current valuation of p, $\neg p$, $(\neg p)\ \mathcal{S}\ p$, and $\mathcal{P}((\neg p)\ \mathcal{S}\ p)$ in this order (1 means satisfied, 0 not). Gray states yield output $\top$, white states $\bot$.

verdicts from a lattice domain, as done by the LTL_3 monitor. These contents can be found in chapter 3 and chapter 4.

2.2.3. Metric (interval) and signal temporal logic

The semantics of LTL was given over finite or infinite words. Metric, metric interval and signal temporal logic, which will be introduced in this section, extend this concept with the presence of time and continuous input data values, and are interpreted over timed streams (aka metric models) and signals.

We start with the syntax and semantics of metric temporal logic [Koy90], which also builds the basis for the other two mentioned formalisms. Metric temporal logic (in the version commonly used today) uses the same operators as linear temporal logic but attaches time intervals to the since and until operator. To align with our definition of time distance measure (definition 2.5), we restrict to the special case of real intervals. In general, the intervals can be from any domain with an associated distance measure in the time domain (see [Koy90]).

Definition 2.21 (Metric temporal logic; based on [Koy90, OW06])**.**

The set of *metric temporal logic (MTL)* formulas over a finite set of atomic propositions $\mathcal{AP}$, $\text{MTL}_{\mathcal{AP}}$ is given by the following grammar:

$$\varphi ::= \texttt{true} \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi\, \mathcal{U}_I\, \varphi \mid \varphi\, \mathcal{S}_I\, \varphi$$

where $p \in \mathcal{AP}$ is an atomic proposition and $I \in \mathbb{I}_{\mathbb{R}_{\geq 0}}$ a possibly infinite interval.

The interval in the subscript of the until and since operator is intended to give a time range in which the second sub-formula has to be fulfilled. As for LTL we skip the $\mathcal{AP}$ and just write MTL if it is clear from the context. We do the same also for the other logics introduced in this section.

As before we first introduce a pointed MTL semantics over timed streams (aka metric models, see definition 2.6). Let again $\Sigma = 2^{\mathcal{AP}}$.

Definition 2.22 (Pointed MTL semantics; based on [Koy90, OW06])**.**

Let $\varphi \in \text{MTL}_{\mathcal{AP}}$ be an MTL formula, $w \in \mathcal{S}_\Sigma^{\mathbb{T}}$ a timed stream over time domain $(\mathbb{T}, <, 0, \ominus)$ with $\mathbb{R}^{\geq 0}$ distance measure and $t \in \mathbb{T}$ a position in the timed stream. The model relation $(w,t) \models_{\text{MTL}} \varphi$, which indicates that φ is satisfied at position t in w, is inductively defined as follows:

$$
\begin{aligned}
&(w,t) \models_{\text{MTL}} \texttt{true} \\
&(w,t) \models_{\text{MTL}} p && \text{iff} \quad p \in w(t) \\
&(w,t) \models_{\text{MTL}} \neg\varphi && \text{iff} \quad (w,t) \not\models_{\text{MTL}} \varphi \\
&(w,t) \models_{\text{MTL}} \varphi_1 \wedge \varphi_2 && \text{iff} \quad (w,t) \models_{\text{MTL}} \varphi_1 \text{ and } (w,t) \models_{\text{MTL}} \varphi_2 \\
&(w,t) \models_{\text{MTL}} \varphi_1 \,\mathcal{U}_I\, \varphi_2 && \text{iff} \quad \text{for some } t' \in t + I : (w,t') \models_{\text{MTL}} \varphi_2 \text{ and} \\
&&&\qquad\qquad \text{for all } t \leq t'' < t' : (w,t'') \models_{\text{MTL}} \varphi_1 \\
&(w,t) \models_{\text{MTL}} \varphi_1 \,\mathcal{S}_I\, \varphi_2 && \text{iff} \quad \text{for some } t' \in t - I : (w,t') \models_{\text{MTL}} \varphi_2 \text{ and} \\
&&&\qquad\qquad \text{for all } t' < t'' \leq t : (w,t'') \models_{\text{MTL}} \varphi_1
\end{aligned}
$$

where for $t \in \mathbb{T}$ and interval $I \in \mathbb{I}_{\mathbb{R}^{\geq 0}}$, $t + I = \{t' \in \mathbb{T} \mid t' \geq t \wedge (t' \ominus t) \in I\}$, $t - I = \{t' \in \mathbb{T} \mid t' \leq t \wedge (t \ominus t') \in I\}$.

Analogous to LTL, the common, initial MTL semantics can be derived in the following way:

Definition 2.23 (Initial MTL semantics; based on [OW06])**.**

Let $\varphi \in \text{MTL}_{\mathcal{AP}}$ be an MTL formula and $w \in \mathcal{S}_\Sigma^{\mathbb{T}}$ a timed stream over time domain $(\mathbb{T}, <, 0, \ominus)$ with $\mathbb{R}^{\geq 0}$ distance measure. The initial semantics of φ, is given as

$$w \models_{\text{MTL}} \varphi \quad \text{iff} \quad (w,0) \models_{\text{MTL}} \varphi.$$

These semantics apply to finite and infinite time domains. Likewise discrete, non-discrete and dense time domains are covered. Further note that the LTL operators $\mathcal{S}$ and $\mathcal{U}$ can be expressed in MTL by choosing $[0, +\infty[$ as interval for the corresponding MTL operators.

A drawback of MTL however is, that the satisfiability of a specific formula is undecidable if non-discrete time domains are allowed [AFH91, AH91, OW06]. In [AFH91] a fragment of MTL, so-called metric interval temporal logic (MITL), is introduced, which differs from MTL in that it is restricted to rational intervals and does not allow singular ones, i.e. those with the lower bound being identical to the upper bound. In this fragment the satisfiability problem is decidable [AFH91].

Definition 2.24 (Metric interval temporal logic; based on [AFH91])**.**

The set of *metric interval temporal logic (MITL)* formulas over a finite set of atomic propositions $\mathcal{AP}$, $\text{MITL}_{\mathcal{AP}}$, is the subset of $\text{MTL}_{\mathcal{AP}}$ formulas, where all intervals attached to a since or until operator are from $\mathbb{I}_{\mathbb{Q}^{\geq 0}}$ and non-singular (i.e. not of form $[a, a]$ for some $a \in \mathbb{Q}^{\geq 0}$).

To MITL the MTL semantics apply.

Signal temporal logic (STL) [MN04] finally extends the idea of MITL to real-valued signals. The signals considered there have $m \in \mathbb{N}^+$ real-valued channels, i.e. their data domain is $\mathbb{R}^m$. STL allows predicates in MITL formulas which depict values from the m-channel signal to boolean values.

Definition 2.25 (Signal temporal logic; based on [MN04])**.**

The set of *signal temporal logic (STL)* formulas, STL_U, over a finite set of predicates $U = \{\mu_1, \ldots, \mu_n\}$ with $\mu_i : \mathbb{R}^m \to \mathbb{B}$, is equal to MITL_U.

In the original definition from [MN04] the intervals in STL formulas are also restricted to be bounded and inclusive, which we do not require in our definition. Further the time domain of the signals is fixed as the set of reals between 0 and $t_{max} \in \mathbb{Q}^+$. In this thesis the definition is extended to arbitrary signals over data domain $\mathbb{R}^m$. The semantics of STL is based on the MITL semantics. The predicates are evaluated on the signal values, and the other operators are identical to MITL.

Definition 2.26 (Pointed STL semantics; based on [MN04])**.**

Let $\varphi \in \text{STL}_U$ be an STL formula over $U = \{\mu_1, \ldots, \mu_n\}$ with $\mu_i : \mathbb{R}^m \to \mathbb{B}$. Let further $s \in \mathcal{S}^{\mathbb{T}}_{\mathbb{R}^m}$ be a signal over time domain $(\mathbb{T}, <, 0, \ominus)$ with $\mathbb{R}^{\geq 0}$ distance measure.

The relation $(s, t) \models_{\text{STL}} \varphi$, which indicates that φ is satisfied at position t in s, is defined as

$$(s, t) \models_{\text{STL}} \varphi \quad \text{iff} \quad (\mu \circ s, t) \models_{\text{MTL}} \varphi$$

where $\mu : \mathbb{R}^m \to 2^U$ with $\mu(v) = \{\mu_i \mid \mu_i(v) = \texttt{true}\}$ is the transformation function from a signal value to a set of atomic propositions.

The initial semantics is given in the usual way:

Definition 2.27 (Initial STL semantics; based on [MN04])**.**

Let $\varphi \in \text{STL}_U$ be an STL formula over $U = \{\mu_1, \ldots, \mu_n\}$ with $\mu_i : \mathbb{R}^m \to \mathbb{B}$. Let further $s \in \mathcal{S}^{\mathbb{T}}_{\mathbb{R}^m}$ be a signal over time domain $(\mathbb{T}, <, 0, \ominus)$ with $\mathbb{R}^{\geq 0}$ distance measure.

The initial STL semantics of φ is given as

$$s \models_{\text{STL}} \varphi \quad \text{iff} \quad (s, 0) \models_{\text{STL}} \varphi.$$

In this thesis we will use the syntactic sugar constants and operators $\texttt{false}, \vee, \rightarrow, \leftrightarrow$ as defined for LTL also for MTL, MITL and STL. Further we introduce an interval version of finally, globally, once and historically:

$$\begin{aligned} \mathcal{F}_I \varphi &:= \texttt{true}\, \mathcal{U}_I\, \varphi \\ \mathcal{G}_I \varphi &:= \neg \mathcal{F}_I (\neg \varphi) \\ \mathcal{O}_I \varphi &:= \texttt{true}\, \mathcal{S}_I\, \varphi \\ \mathcal{H}_I \varphi &:= \neg \mathcal{O}_I (\neg \varphi) \end{aligned}$$

Having defined timed extensions of LTL, it is natural to ask for monitoring algorithms and automaton constructions for these logics. Several approaches to this have been presented in the literature. We will not discuss them in detail, as they will not play a significant role in the remainder of this thesis, but two prominent ones are outlined in the following.

An automaton construction for the initial MITL semantics is described in [MNP06]. By adding a component that transforms the input signal into a boolean signal like μ in definition 2.26, this approach should also be extendable to STL. The idea of the algorithm is to translate the MITL operators to timed signal transducers – a variant of Büchi automata with timed transitions and output generation – and to build compositions of them.

In [MN04] the authors present an offline monitoring algorithm for STL and MITL, restricted to a finite notion of piece-wise constant signals. The algorithm is mainly based on the idea of applying the formula's MITL operators to the finite input signals by compositionally merging and shifting them. Thereby one receives a timed boolean stream representing the truth value of the whole STL formula over time.

2.2.4. Monitoring under uncertainty and assumptions

We will now discuss two further relevant topics in runtime verification in this section, *uncertainty* and *assumptions*. We define these concepts in terms of traces, which

makes them also applicable to synchronous event streams and timed streams over discrete time domains, by encoding them as traces. The definitions can further be applied to signals (i.e. streams over dense time domain) at the cost of approximating them as traces as described in section 2.1.3.

Uncertainty

The term *uncertainty* describes the circumstance that some parts of the trace, that the monitor receives as input, are imprecise, i.e. some letters are either partially or completely unknown. In principle, one can consider an uncertain trace received by the monitor as the subset of Σ^* that contains all possible traces with respect to the uncertainty (c.f. [CTT19]).

Definition 2.28 (Uncertain trace)**.**

An *uncertain trace* over a data domain Σ is a set of finite traces $U \subseteq \Sigma^*$.

A trace $w \in \Sigma^*$ is called a *concrete trace of* U if and only if $w \in U$.

An uncertain trace $U \subseteq \Sigma^*$ is an *extension* of an uncertain trace $V \subseteq \Sigma^*$, denoted $V \preceq U$, if and only if

$$\forall u \in U.\, \exists v \in V, e \in \Sigma^*.\, u = v \circ e.$$

A monitor handling uncertainty can be understood as one which is receiving a sequence of uncertain traces, where each one is an extension of the previous one. Based on the considerations about the quality of a monitor in definition 2.13 we define a sound and perfect uncertain monitor in definition 2.29.

Definition 2.29 (Sound and perfect uncertain monitor; based on [KLSS22])**.**

Let M be a monitor receiving a sequence $U_1 \preceq U_2 \preceq \cdots \in (2^{\Sigma^*})^*$ of extending uncertain traces over Σ and subsequently yielding verdicts $v_1, v_2, \cdots \in \mathbb{V}$ over verdict domain $\mathbb{V} \supseteq \{\top, \bot\}$.

For a language $\mathcal{L}_\varphi \subseteq \Omega$ with $\Omega \in \{\Sigma^*, \Sigma^n, \Sigma^\omega, \Sigma^\infty\}$, M is called

- *Sound uncertain monitor* for $\mathcal{L}_\varphi$, if and only if for any input sequence $U_1 \preceq U_2 \preceq \cdots \preceq U_n$

$$\forall i \in \mathbb{N}^+.\, v_i = \top \Rightarrow \forall w \in U_i, wv \in \Omega.\, wv \in \mathcal{L}_\varphi \text{ and}$$
$$\forall i \in \mathbb{N}^+.\, v_i = \bot \Rightarrow \forall w \in U_i, wv \in \Omega.\, wv \notin \mathcal{L}_\varphi$$

- *Perfect uncertain monitor* for $\mathcal{L}_\varphi$, if and only if for any input sequence $U_1 \preceq U_2 \preceq \cdots \preceq U_n$

$$\forall i \in \mathbb{N}^+. v_i = \top \Leftrightarrow \forall w \in U_i, wv \in \Omega. wv \in \mathcal{L}_\varphi \text{ and}$$
$$\forall i \in \mathbb{N}^+. v_i = \bot \Leftrightarrow \forall w \in U_i, wv \in \Omega. wv \notin \mathcal{L}_\varphi$$

Thus a monitor is sound, if it casts the final verdicts $\top$ and $\bot$ only if all possible continuations of the concrete traces are in the language $\mathcal{L}_\varphi$. Further it is perfect if it also casts the final verdicts whenever possible. Note that this definition has strong parallels with the notions of impartiality and anticipation from definition 2.13. Indeed soundness corresponds to impartiality and perfectness to impartiality and anticipation.

It remains open how the sequence of uncertain traces, i.e. a sequence of sets of traces, can be passed to the monitor which usually receives a sequence of letters. In this thesis we bridge this gap by introducing an uncertain data domain Γ. Over this domain, an input trace is formed that encodes the uncertain input and is passed to the monitor letter by letter. Additionally we require an interpretation function that maps inputs over Γ to the corresponding uncertain traces.

Definition 2.30 (Uncertainty encoding)**.**

Let Σ, Γ be data domains, where Σ contains certain and Γ uncertain data values.

A function $\nu : \Gamma^* \to 2^{\Sigma^*}$ is called *uncertainty encoding* if and only if

$$\forall w, w' \in \Gamma^*. w \sqsubset w' \Rightarrow \nu(w) \prec \nu(w')$$

Hence, we require the encoding to map a strict continuation w' of an encoded trace w to a strict continuation of the represented uncertain trace. This guarantees that the monitor receives a consistent and growing amount of information about the uncertain input with each letter of Γ.

A most basic idea for the uncertainty encoding is to encode the possibilities for every letter individually, i.e. $\Gamma = 2^\Sigma$. We call this *letter-wise* or *timestamp/instant-immanent uncertainty*. However, not all uncertain inputs can be uniquely encoded this way. Consider for example the following uncertain trace over $\Sigma = \{a, b, c\}$:

$$\{a, b, c\}, \{aa, ab, bb, bc, cc\}, \{aaa, aab, abb, abc, bbb, bbc, bcc, ccc\}, \ldots$$

I.e. the exact input letters are fully unknown, but a is only followed by a or b, b by b or c and c by c. In this case the possible extension letters are always dependent on the previous letter and thus there is no letter-wise encoding.

In [KPD22] Kushwaha et al. present a more general method for abstracting uncertain traces. They represent sets of Σ-sequences (here called chunks) by letters of the uncertain data domain. Therefore they make use of a so-called loss model, which defines the relation between uncertain letters and the certain chunks they can represent. In their work, the loss model is meant as theoretical construct to model a partial trace, where information was lost compared to a certain trace. In this thesis, we take up the idea and define the chunk uncertainty model, which is basically equivalent to theirs, but relaxes some specific conditions that we do not require.

Definition 2.31 (Chunk uncertainty model; based on [KPD22])**.**

Let Σ, Γ be data domains, where Σ contains certain and Γ uncertain data values.

A relation $R \subseteq \Sigma^* \times \Gamma$ is called *chunk uncertainty model*.

Given a chunk uncertainty model, a corresponding uncertainty encoding is defined in definition 2.32.

Definition 2.32 (Chunk uncertainty encoding; based on [KPD22])**.**

Let $R \subseteq \Sigma^* \times \Gamma$ be a chunk uncertainty model and $w = w_1 w_2 \dots w_n \in \Gamma^*$ a trace over the uncertain data domain. The corresponding uncertainty encoding $\nu : \Gamma^* \to 2^{\Sigma^*}$ is defined as

$$\nu(w) = \begin{cases} \{\epsilon\} & \text{if } w = \epsilon \\ \{u \mid u = v \circ v' \wedge R(v, w_1) \wedge v' \in \nu(w_2 \dots w_n)\} & \text{else} \end{cases}$$

An example of a chunk uncertainty model (inspired by the one from [KPD22]) for $\Sigma = \{p, q\}$ and $\Gamma = \Sigma \cup \{1, 2, \dots, n\}$, could be

$$R(a, b) \Leftrightarrow (a = b \wedge a \in \Sigma) \vee (|a| \leq b \wedge b \in \{1, \dots, n\})$$

I.e. a number k in the input trace indicates a missing segment of length 0 to k in the concrete trace. Thus, for the input $p2q$, the corresponding uncertain trace is

$$\nu(p2q) = \{pq, ppq, pqq, pppq, ppqq, pqpq, pqqq\}.$$

The presented notion of abstracted traces and uncertainty models is able to express most uncertainty patterns, especially potential duplication and potential loss of events, noisy or fully unknown values and local value combinations. The model though is not able to express relations between uncertain events which are arbitrarily far from each other, e.g. the first letter is a, b or c, but it cannot be c if there is a further c somewhere in the trace. Thereby, however, an incremental reception of an

uncertain run is enabled. Consequently the presented uncertainty encoding can be seen as a general way to deal with uncertain inputs in runtime verification. In chapter 3 of this thesis we will build our monitoring theory on this kind of uncertainty encoding. For the monitoring algorithm which is presented in chapter 4, however, we will restrict it to letter-wise uncertainty again.

The traditional monitoring approach for uncertainty is to use power set constructions to express possible states in which the certain monitor can be and to combine the corresponding verdicts in a suitable way [KPD22, KLSS22]. The monitoring algorithm in chapter 4 is also based on this idea.

Another conceivable aspect in the context of uncertain input traces is the assignment of probabilities to the specific trace variations (e.g. a specific letter in the trace is with 70% probability an a, otherwise a b). This information could then be considered in probabilistic monitoring approaches, e.g. [SBS+11]. In this thesis however we will not consider this kind of uncertainty but focus on uncertainty without specific probabilities, as introduced above.

Assumptions

A specification φ in traditional runtime verification defines a language $\mathcal{L}_\varphi \subseteq \Omega$ with $\Omega \in \{\Sigma^*, \Sigma^n, \Sigma^\omega, \Sigma^\infty\}$ of traces that fulfill some property. Yet oftentimes the set of actually possible runs of the system is not Ω but a subset $\mathcal{A} \subseteq \Omega$, as some traces in Ω may never be taken by the system in reality (see [Leu12]). We call such a subset, restricting the possible system runs, an assumption.

Definition 2.33 (Assumption; based on [HS20])**.**

An *assumption* $\mathcal{A} \subseteq \Omega$ with $\Omega \in \{\Sigma^*, \Sigma^n, \Sigma^\omega, \Sigma^\infty\}$ is a set of potential runs of a system.

A run $w \in \Omega$ *meets* an assumption if $w \in \mathcal{A}$.

Formally an assumption is a language and may thus be given in different ways, e.g. grammars, automata, Kripke structures or transition systems.

In the presence of assumptions, the monitoring process can be adjusted to consider only input extensions that match the assumption. For this purpose, we restrict the containment operator $\in$ in the following way:

Definition 2.34 (Containment provided assumption)**.**

Let $\mathcal{A} \subseteq \Omega$ with $\Omega \in \{\Sigma^*, \Sigma^n, \Sigma^\omega, \Sigma^\infty\}$ be an assumption and φ a correctness property with corresponding language $\mathcal{L}_\varphi \subseteq \Omega$.

> A trace $w \in \Omega$ is *contained in* $\mathcal{L}_\varphi$ *provided* $\mathcal{A}$ ($w \in_{\mathcal{A}} \mathcal{L}_\varphi$) if and only if
>
> $$w \in \mathcal{A} \Rightarrow w \in \mathcal{L}_\varphi.$$
>
> and it is *not contained in* $\mathcal{L}_\varphi$ *provided* $\mathcal{A}$ ($w \notin_{\mathcal{A}} \mathcal{L}_\varphi$) if and only if
>
> $$w \in \mathcal{A} \Rightarrow w \notin \mathcal{L}_\varphi.$$

This definition allows us to adjust the notion of impartial and anticipatory monitoring (definition 2.13) as well as sound and perfect monitoring under uncertainty (definition 2.29) in the presence of assumptions. Therefore we simply have to replace $wv \in \mathcal{L}_\varphi$ and $wv \notin \mathcal{L}_\varphi$ in the corresponding definitions by $wv \in_{\mathcal{A}} \mathcal{L}_\varphi$ and $wv \notin_{\mathcal{A}} \mathcal{L}_\varphi$. Effectively this obligates sound and perfect monitors to cast final verdicts if the corresponding property holds or is violated for all input extensions that match the assumption.

If $\mathcal{A}$ is not the trivial assumption (i.e. $\mathcal{A} \neq \Omega$), monitoring under assumptions has several favorable implications on the monitoring process, that go beyond plain parallel monitoring of the the assumption and the property (see [Leu12, CTT19, HS20, KLSS22]):

- ***Early Conclusion*:** When a monitor receives an input, it may be capable of giving a final verdict earlier if an assumption is present, as the set of input extensions to be considered is restricted by $\mathcal{A}$.
- ***Extended Monitorability*:** If a monitor can reach a state, where it is not possible to cast a final ever again, because for the received prefix there is always an extension which leads to a trace in $\mathcal{L}_\varphi$ and one that leads to a trace not in $\mathcal{L}_\varphi$, then the corresponding property is considered as non-monitorable (see [PZ06]). The existence of assumptions may make such a property monitorable, as certain extensions of prefixes don't have to be considered anymore, because they are outside of the assumption.
- ***Uncertainty removal*:** When monitoring under uncertainty, an assumption may restrict the number of possible concrete runs, as some of them can actually not be extended to a trace which is compatible with the assumption. Consequently the monitor may be able to produce more precise verdicts.

Monitoring under assumptions is in general closely related to the field of model checking (c.f. [Leu12]). Consider an assumption $\mathcal{A}$ that is an exact model of the underlying system. Deciding whether $\top$ is the verdict of an anticipatory, impartial monitor for φ in the initial state is equal to model checking the property φ for the system. This already leads to the insight that anticipatory runtime verification under particular assumptions is in general undecidable due to Rice's theorem [Ric53].

Of course, if assumptions are present, it is also thinkable to check that the input trace handed to the monitor is coherent with the assumptions at all. In general this can

be done quite straight-forwardly, as long as the assumption allows for anticipatory and impartial monitoring. If the monitor has received trace w and for all $wv \in \Omega$, $wv \notin \mathcal{A}$, then w is an impossible input. However, reasoning with assumptions makes little sense if it is conceivable that traces are obtained that actually violate the assumption. For this reason, we will not consider this case in the rest of this thesis.

2.3. Stream runtime verification

A major part of this thesis deals with *stream runtime verification (SRV)*, which is considered as a generalization of the classical RV approaches (see [BS14]). In SRV special stream-based specification languages are used to describe the monitored properties of the system under scrutiny. The three most popular SRV languages are LOLA [DSS+05], including its extensions LOLA 2.0 [FFST16] and RTLola [FFST17, FFS+19], TeSSLa [CHL+18, KLS+22b] and Striver [GS18], which will be presented in the remainder of this section.

The central insight underlying SRV is that runtime verification can generally be understood as a stream transformation process. A monitor receives a sequence of observed values from the monitored system (i.e. streams of observed values) and transforms them into a stream of monitoring outputs, e.g. verdicts from some verdict domain. The approach of SRV is the direct description of the monitored property as a stream transformation function and the subsequent generation of a monitor out of it. In this general setting however, the operations and data types used for the stream transformation are not restricted in any way, which makes SRV a very powerful monitoring approach. Likewise the input events can have data of any type attached (e.g. variable or parameter values), and the monitor is not limited to issue verdicts from a truth domain but can cast output values from any domain to provide information about relevant system properties. For illustration of the approach, consider the following scenario.

Imagine a self-driving robot system. The RV monitor observes the robot's speed at a regular basis, additionally it receives the information whether the system is currently passing through critical terrain. Now the property to be observed is, that the speed of the robot never exceeds the threshold 10, if it is inside a critical section. The desired monitor output is 0, if everything is ok, otherwise the speed amount that the robot moves to fast.

From an SRV perspective such a monitor receives two input streams, `speed` of type real and `crit` of type boolean. These streams are transformed into an output stream again of type real. In SRV languages one describes such a stream transformation with the use of so-called *intermediate streams*. An intermediate stream results from the application of a basic stream operator on input or intermediate streams. Finally some of the intermediate streams which are formed this way are marked as output.

In the example from above one could e.g. specify an intermediate stream which holds the difference of `speed`'s current value to 10. Then one could define an additional intermediate stream, marked as output, which calculates the final result via an if expression, which evaluates to the value of `diff` if it is greater than 0 and `crit` is true and otherwise to 0. An illustration of this stream transformation can be found in figure 2.7.

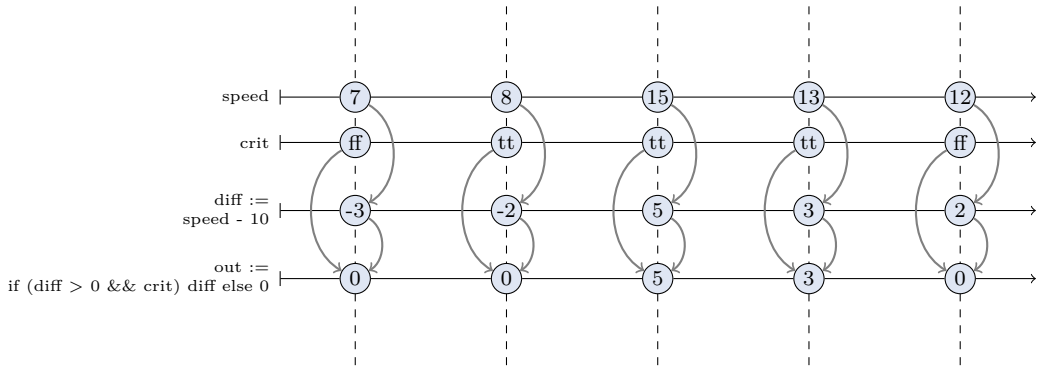


Figure 2.7.: Example of a stream transformation as used in stream runtime verification. The gray arrows mark the data flow among the individual stream events.

As the evaluation of stream operations is driven by the event flow between the streams, stream runtime verification languages follow the dataflow paradigm and thus show similarities to other languages of this family, especially Lustre [CPHP87] or SIGNAL [GL87], and functional reactive programming frameworks, see [EH97, Mag16]. However there are some aspects which usually cannot be found in these languages:

Traditional dataflow languages are programming languages, intended for a direct, causal description of a program behavior. SRV languages on the other hand have a more descriptive character, as is typical for specification languages [DSS+05]. These languages are intended for the specification of a (correctness) property, rather than the monitor execution. This difference is particularly noticeable in the fact that traditional dataflow languages are based on an immanently synchronous execution model, while in SRV there are typically language constructs with asynchronous or time-related character available (see [DSS+05, Sch24]):

- The languages often support a future operator, which allows to reference a stream event in the future. Such a feature may of course make the definite computation of an event depending on future events impossible until the future values are known. However from a specification rather than a programming standpoint it makes sense to have a future operator included, as some properties can be formulated more intuitively with such references.

- Likewise SRV languages often contain specific operators which allow the generation of output events deposed from the arrival time of input events (e.g. TeSSLa's `delay` operator or Striver's ticking expressions) or operators to calculate on events with asynchronous arrival (*signal semantics*, see [CHL+18]).
- Finally in SRV, the languages often also support features to handle a separation between system time and the monitor time [LSS+18], e.g. explicit timestamps attached to the events. Traditional dataflow languages usually do not require this distinction as for them only a unique execution time exists.

To handle theses features, evaluation algorithms for SRV languages often follow advanced, non-linear strategies.

2.3.1. LOLA

This section gives a formal introduction to the syntax and semantics of the pioneering SRV language *LOLA* (named after the 1998 movie "Run Lola Run"[2]). Subsequently a short outline of the related languages TeSSLa and Striver will be given, which, however, will play a subordinate role in the remainder of this work.

Syntax & semantics

From a formal perspective a LOLA specification is an equation system, where expressions, which define streams, are bound to stream identifiers. LOLA exclusively deals with finite, synchronous event streams (see definition 2.3), i.e. those without explicit timestamps attached to the events. We assume the instant domain of these streams to be globally fixed as $\mathbb{T} = \{0, 1, \ldots, t_{max}\}$. Furthermore, in this thesis we do not consider the very special case where $t_{max} = 0$, i.e. the streams are only of length 1.

The data type of streams is not restricted in any way, i.e. they may carry any value, e.g. booleans, numbers or more complex data structures like sets or maps. A LOLA specification however has to be type correct. Every stream is restricted to carry events from a specific data domain. If a stream may only bear events from data domain $\mathbb{D}$, we say this stream is of sort (or type) $\mathbb{D}$. For input streams, it is assumed that the type is fixed and known, while the type of intermediate streams, which are defined in the specification, is given implicitly by the type of the defining expression.

The syntax of typed LOLA expressions is defined in the following.

[2] Personal communication with César Sánchez, May 2023

Definition 2.35 (LOLA stream expressions; based on [DSS+05, KLS22a])**.**

The set of *LOLA stream expressions* of sort (or type) $\mathbb{D}$ over a set of stream identifiers S, $Exp^S_{\mathbb{D}}$, is defined as

$$Exp^S_{\mathbb{D}} ::= c \mid s[o|c] \mid f(Exp^S_{\mathbb{D}_1}, ..., Exp^S_{\mathbb{D}_n}) \mid ite(Exp^S_{\mathbb{B}}, Exp^S_{\mathbb{D}}, Exp^S_{\mathbb{D}})$$

where $c \in \mathbb{D}$ is a constant symbol of sort $\mathbb{D}$, $s \in S$ is a stream identifier of sort $\mathbb{D}$, $o \in \mathbb{Z}$ is an offset, f is a function symbol of sort $\mathbb{D}_1 \times \cdots \times \mathbb{D}_n \to \mathbb{D}$ and $\mathbb{B} = \{true, false\}$ is the boolean type.

Exp^S denotes the set of stream expressions over S of any sort.

A constant symbol describes the stream with this constant value at every position. The offset expression $s[o|c]$ denotes a stream which at position t carries the value of stream s at position $t + o$, if $t + o \in \mathbb{T}$, otherwise the constant default value c. In LOLA the offset expression is also used for accessing the current value of a stream, i.e. with offset $o = 0$. In this case, however, the default value c is not relevant, as a position outside of the instant domain $\mathbb{T}$ is never accessed. Consequently $s[now]$ may be used as shorthand for $s[0, d]$ with any constant d of correct type. Further, the expression $f(s_0, \ldots, s_n)$ is the stream whose events result from application of the function f on s_1 to s_n's current events. The term *ite* finally is the application of the special function symbol *ite* used for if-then-else expressions.

Based on definition 2.35, the syntax of a LOLA specification is defined as follows:

Definition 2.36 (LOLA syntax; based on [DSS+05, KLS22a])**.**

A *LOLA specification* $\varphi = (I, S, O, E)$ is a 4-tuple consisting of

- a finite set of input stream identifiers I,
- a finite set of intermediate stream identifiers S, s.t. $S \cap I = \emptyset$,
- a finite set of output stream identifiers $O \subseteq S$,
- a mapping $E : S \to Exp^{S \cup I}$, which assigns to every intermediate identifier $s \in S$ a stream expression $E(s)$ over input and intermediate stream identifiers.

The original LOLA definition from [DSS+05] also introduces a special `trigger` keyword, which can be applied to an intermediate stream of type $\mathbb{B}$. The semantics of this keyword is, that whenever the stream to which it is applied carries the value true, the monitor raises a notification. However such a keyword can be omitted in a theoretical consideration as trivially the affected stream can be marked as output and the generated monitor can be extended by a special alert layer implementing the desired behavior.

Throughout this thesis we will frequently require the specification to be in a *flattened* format. This means that it must contain only -1,0 and $+1$ offsets. In general,

any LOLA specification can be translated into such a flattened format by splitting higher and lower offsets into a chain of subsequently applied $+1$ or -1 offset operations on newly introduced helper streams. A definition `s = x[-3|0]` could e.g. be transformed into `s = s'[-1|0]`, `s' = s''[-1|0]`, `s'' = x[-1|0]` where `s'`, `s''` are fresh stream identifiers.

The algebra of all data domains and function symbols in a LOLA specification is called induced algebra of the specification. As every algebra in this thesis we also require it to include the boolean sort $\mathbb{B}$ with constants and the $\wedge$ operator.

Definition 2.37 (Induced algebra of LOLA specification)**.**

Let $\varphi = (I, S, O, E)$ be a LOLA specification. The *induced algebra of* φ, $\mathfrak{A}^{\varphi}$, is the algebra consisting of

- the sort $\mathbb{B}$,
- the sorts of all input and intermediate streams,
- all elements of the included sorts as constants,
- all functions used in φ,
- the operator $\wedge$ in its usual definition,
- the equality operator $=$ for all sorts.

As example for a LOLA specification consider the one in figure 2.8. The form of representation used there is quite intuitive and resembles the specification format used in existing LOLA implementations. On top of the specification all input streams are declared by the `input` keyword and the corresponding type information. In the latter part all intermediate streams are defined by equations with the stream identifier on the left hand side and LOLA stream expressions on the right hand side. For the application of common mathematical and logical functions $(+, >, \wedge, \dots)$ infix notation is used. In the last line the stream `out` is finally marked as output stream with help of the `output` keyword. This proposed way of writing LOLA specifications will also be utilized throughout the rest of this thesis.

```
1  input speed: ℝ
2  input crit: 𝔹
3
4  speed_avg := (speed[-1|0] + speed[now] + speed[+1|0]) / 3
5  diff := speed_avg[now] - 10
6  out := ite(diff[now] > 0 ∧ crit[now], diff[now], 0)
7
8  output out
```

Figure 2.8.: Example specification formalized in LOLA

The scenario specified in the example is an extension of the self-driving robot example discussed above (see figure 2.7). In difference to the property described there,

the specification from figure 2.8 first computes the average over three values from the input stream `speed` (the current, the last and next one) and keeps the result in the intermediate stream `speed_avg`. As in the mentioned example, the specification then computes a stream `diff` by subtracting the high speed (10) from the current value of `speed_avg`. Finally stream `out` is defined to take over this value, if it is positive and the robot is currently in a critical section (indicated by the current event of input stream `crit`), otherwise it is zero. The induced algebra of the specification contains the sorts real ($\mathbb{R}$) and bool ($\mathbb{B}$) with their values as constants and the operations $\wedge$, $=$, $+$, $-$, $/$, $>$, *ite*.

Next we will introduce the formal semantics of a LOLA specification. Therefore we start with defining the semantics of a LOLA expression and what a solution of a LOLA specification is.

Definition 2.38 (LOLA semantics; based on [DSS+05])**.**

Let $\varphi = (I = \{s_1, \ldots, s_n\}, S = \{s_{n+1}, \ldots, s_{n+m}\}, O = \{s_{i_1}, \ldots, s_{i_l}\}, E)$ with all $s_{i_j} \in S$ be a LOLA specification.

Given a tuple of streams $\Pi = (\pi_1, \ldots, \pi_{n+m}) \in \mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_{n+m}}$ corresponding to stream identifiers $s_1, \ldots, s_{n+m}$ over instant domain $\mathbb{T}$, the *semantics of a LOLA expression* w.r.t. φ, $[\![\cdot]\!]_{\Pi,\varphi} : Exp_{\mathbb{D}}^{S \cup I} \to \mathcal{S}_{\mathbb{D}}$, is iteratively defined as:

- $[\![c]\!]_{\Pi,\varphi}(t) = c$
- $[\![s_i[o|c]]\!]_{\Pi,\varphi}(t) = \begin{cases} \pi_i(t+o) & \text{if } t+o \in \mathbb{T} \\ c & \text{otherwise} \end{cases}$
- $[\![f(e_1, ..., e_n)]\!]_{\Pi,\varphi}(t) = f([\![e_1]\!]_{\Pi,\varphi}(t), \ldots, [\![e_n]\!]_{\Pi,\varphi}(t))$
- $[\![ite(e_1, e_2, e_3)]\!]_{\Pi,\varphi}(t) = \begin{cases} [\![e_2]\!]_{\Pi,\varphi}(t) & \text{if } [\![e_1]\!]_{\Pi,\varphi}(t) = \texttt{true} \\ [\![e_3]\!]_{\Pi,\varphi}(t) & \text{if } [\![e_1]\!]_{\Pi,\varphi}(t) = \texttt{false} \end{cases}$

A tuple of streams $\Pi = (\pi_1, \ldots, \pi_{n+m}) \in \mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_{n+m}}$ is called *solution of* φ for input streams $\Sigma = (\sigma_1, \ldots, \sigma_n) \in \mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_n}$ if and only if

$$\Pi = (\sigma_1, \ldots, \sigma_n, [\![E(s_{n+1})]\!]_{\Pi,\varphi}, \ldots, [\![E(s_{n+m})]\!]_{\Pi,\varphi})$$

The *semantics of* φ for input $\Sigma \in \mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_n}$ is defined as

$$\begin{array}{lcl} [\![\varphi]\!] & : & \mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_n} \to \mathcal{S}_{\mathbb{D}_{i_1}} \times \cdots \times \mathcal{S}_{\mathbb{D}_{i_l}} \\ [\![\varphi]\!](\Sigma) & = & (\pi_{i_1}, \ldots, \pi_{i_l}) \end{array}$$

if and only if for φ and Σ there is a unique solution $\Pi = (\pi_1, \ldots, \pi_{n+m})$ and otherwise undefined.

The semantics of the individual expressions are defined quite straightforwardly. Constants evaluate to the stream with the constant at each position, offsets take the corresponding value from the given tuple of all input and intermediate streams, or the default value if a position outside the instant domain would be accessed. The function application and if expressions are applied to the values of their sub-expressions. A solution of a specification, for a tuple of input streams, is defined in a recursive manner. It is each tuple of streams Π consisting of the input streams concatenated with the semantics of the defining expressions of all intermediate streams w.r.t. Π itself. Finally, the semantics of a LOLA specification, given a tuple of concrete input streams, consists of the output streams in the solution Π, if it exists and is unique.

Per se, the above definition does not guarantee that there is a solution of a LOLA specification (e.g. if a real stream is defined as $\mathtt{x} := \mathtt{x}[now] + 1$), nor that there are not multiple solutions (e.g. if a stream definition $\mathtt{x} := \mathtt{x}[now]$ is contained). In fact, a LOLA specification has a unique solution if no stream event at any given instant is dependent on itself. I.e. there is no cycle in the recursive evaluation [DSS+05]. In this work we only consider so-called *well-formed* LOLA specifications, which are defined on basis of the specification's dependency graph.

> **Definition 2.39** (LOLA dependency graph; based on [DSS+05])**.**
>
> Let $\varphi = (I, S, O, E)$ be a LOLA specification.
>
> The corresponding *dependency graph* is a directed, weighted graph $G_\varphi = (V, D)$ with nodes $V = I \cup S$ and weighted edges $D \subseteq V \times V \times \mathbb{N}$, s.t. $(u, v, o) \in D$ if and only if $v \in S$ and the definition of stream v ($E(v)$) contains the sub-expression $u[o|c]$ for any constant c.

The dependency graph contains the information which streams depend on which. The edge weight indicates the time offset between an influenced event and its influencing event. Clearly, if there is a cycle with an edge sum of zero the value of an event may depend on itself. In this case the LOLA specification is said not to be well-formed.

> **Definition 2.40** (LOLA well-formedness; based on [DSS+05])**.**
>
> A LOLA specification φ is *well-formed*, if and only if the corresponding dependency graph has no cycle with edge sum zero.

Well-formed LOLA specifications were shown to have a unique solution [DSS+05].

LOLA monitoring

Now we want to discuss how to monitor a LOLA specification. Here the original monitoring algorithm from [DSS+05] is shown (with slight modifications in presentation), commonly referred to as *universal monitoring algorithm*. The key idea is

that the monitor internally maintains a set A of active equations (i.e. boolean expressions) that describe the values of particular stream events. These expressions are over the induced algebra $\mathfrak{A}^\varphi$ and the set of so-called instant variables which represent a single event of an input or intermediate stream:

Definition 2.41 (LOLA instant variables and expressions; based on [DSS+05, KLS22a])**.**

Let $\varphi = (I, S, O, E)$ be a LOLA specification.

The set of *instant variables of* φ is given as

$$\mathbb{V}^\varphi = \{s^t \mid s \in S \cup I, t \in \mathbb{T}\}$$

where each s^t is of the same sort as stream s.

The set of *instant expressions from* φ *of type* $\mathbb{D}$ is defined as $\mathbb{E}^{\mathbb{D}}_{\varphi} := \mathbb{E}^{\mathbb{D}}_{\mathfrak{A}^\varphi, \mathbb{V}^\varphi}$.

The basic concept is that during monitoring the expressions in $A \subseteq \mathbb{E}^{\mathbb{B}}_{\varphi}$ are continuously refined with the given input readings and evaluated. The algorithm (depicted in figure 2.9) follows the subsequently listed steps:

1. Events from the input streams are received. Here it is assumed that the events appear in the correct order and synchronously. I.e. all input events at a certain instant have to arrive before further events from later instants are received. If input stream s at instant t has the value v, the equation

   $$s^t = v$$

   is added to the set of active equations A.

2. When all input events at a certain instant are received, the equations from the specification are instantiated for this instant (denoted as E^t) and added to A (function `Compute`). Therefore the offsets and left hand side stream identifiers in the equations are replaced by instant variables of the corresponding stream events or default values. E.g. for instant 0 the equation

   ```
   speed_avg := (speed[-1|0] + speed[now] + speed[+1|0]) / 3
   ```

   from the specification in figure 2.8 would be instantiated to

   $$\mathit{speed\_avg}^0 = (0 + \mathit{speed}^0 + \mathit{speed}^1)/3.$$

3. Subsequently all equations in A are evaluated as far as possible: Whenever an equation of form $s^k = v$ is contained in A where v is a constant value, the corresponding output for this instant is generated if s is an output stream (function `Output`) and all further occurrences of s^k in equations from A are

replaced by v. Function applications (incl. `ite` expressions) are resolved as far as possible. This partial evaluation is repeated until no further simplifications can be performed.

4. Finally all equation of form $s^k = v$ with a constant value v are removed from A, if there is no $s[o|c]$ expression in φ with $o < k - t$ in φ with t being the current instant (function `Prune`). This removal is possible, because if there is an equation $s^k = v$ in A, all occurrences of s^k were already replaced by v in step 3 and in future equations s^k cannot occur anymore as there is no offset expression with a sufficiently high negative offset in φ.

5. If the end of the trace is not yet reached (i.e. $t \leq t_{max}$), the procedure is repeated for the next instant from step 1 on.

We will now take a look at the resources required by this monitoring algorithm. Consider a fixed specification φ. For the following considerations we make some assumptions: First, we require the data domains used in φ to be storable in constant size, e.g. a single register. We also assume the evaluation time of the function symbols used there to be constant. These assumptions are necessary and common for resource estimation of monitoring algorithms. Without it (e.g. when complex data structures like sets or maps are involved in the specification) no resource bounds could be found, as runtime and memory consumption of the algorithm can always be dominated by the storage of and the evaluation time of function symbols on large input values.

Secondly we consider the maximal timestamp $t_{max} \in \mathbb{T}$ to be part of the monitor input (e.g. as first letter or by a special letter indicating the trace end) without explicitly denoting it. A monitor is thus trace-length-independent (definition 2.13) if its resource demand per instant is constant for any $\mathbb{T}$. Considering $\mathbb{T}$ to be given globally or to be part of the specification would make little sense, as in this case every monitor would be trace-length-independent because the actual monitoring would only have to be performed for a finite set of instances and thus automatically be bounded by a constant.

With these assumptions, which we will also uphold throughout the rest of this thesis, it is easy to see, that the monitoring time per instant (runtime of one execution of the while body) is polynomial in terms of the number of equations in A (note, that length of these equations is bounded by a constant, as they are generated from φ). With an advanced simplification strategy the runtime can further be reduced to be linear in the size of A. The memory consumption, is also linear in $|A|$.

A consequence of this complexity analysis is that the monitoring algorithm is not trace-length-independent as the set of active equations A, may grow with the monitoring time. This is caused by the positive offset expressions (future offsets) allowed in LOLA. With them, it is possible that every newly instantiated expression contains future offsets to intermediate streams, and thus no expression can be simplified to a value and be removed from A. However, this trace-length-dependence is not due

```
1  t ← 0;
2  A ← ∅;
3  while t ≤ t_max do
4      foreach i ∈ I do
5          Read input value v for stream i at instant t;
6          A ← A ∪ {i^t = v};
7      A ← Compute(t, A);
8      Output(A);
9      A ← Prune(t, A);
10     t ← t + 1 ;
11 Function Compute(t, A) is
12     A ← A ∪ E^t;
13     Simplify all equations in A;
14     while new (s^k = v) ∈ A do
15         Replace all other s^k in A by v;
16         Simplify modified equations in A;
17     return A;
18 Function Output(A) is
19     foreach s ∈ O ∧ (s^k = v) ∈ A do
20         if s^k is not yet printed then
21             Output s^k = v;
22 Function Prune(t, A) is
23     foreach (s^k = v) ∈ A do
24         if no s[o, c] in φ with o < (k − t) then
25             A ← A\{s^k = v};
26     return A;
```

Figure 2.9.: Monitoring algorithm for LOLA specification $\varphi = (I, S, O, E)$. E^t denotes the instantiated equations from E for instant t, v a constant value.

to a non-optimal evaluation algorithm. Under the premise that the monitor should report the exact value of each output stream event, LOLA specifications are generally not monitorable with trace-length-independent resources [DSS+05]. Consider e.g. the specification in figure 2.10.

In this specification there are two boolean input streams, `x` and `y`. Two further streams are defined on basis of these inputs:

- Stream `z` is true whenever `x` is true now or at any position in the future.
- Stream `v` is true at all instants where also `y` and `z` are true.

```
1  input x: 𝔹
2  input y: 𝔹
3
4  z := z[+1|false] ∨ x[now]
5  v := y[now] ∧ z[now]
6
7  output v
```

Figure 2.10.: Example of a LOLA specification not monitorable with trace-length-independent resources if the value of all output events shall be cast.

One can easily see, that an evaluation of this specification requires the monitor to store all positions at which stream `y` was true until the first event with value true occurs on stream `x`. This is because at the moment when the first true event is present on stream `x`, but not earlier, all previous events on stream `z` are also known to be true, and likewise the events of `v` at instants when `y` was also true. Obviously, any monitoring strategy for this LOLA specification which yields the values of all output events, must store exactly these positions, the number of which can grow linearly with the trace length.

As pointed out before it is often desirable for RV tasks, that the resources demanded by the monitor are bounded by a constant. Thus [DSS+05] identifies a so-called efficiently monitorable LOLA fragment, for which trace-length-independent monitors can be synthesized:

> **Definition 2.42** (Efficiently monitorable LOLA fragments; based on [DSS+05, Gor22])**.**
>
> A LOLA specification φ is called *efficiently monitorable* if and only if there are no cycles with a positive edge sum in its dependency graph.
>
> Further φ is called *very efficiently monitorable* if it does not contain positive offset expressions.

A monitor following the evaluation strategy from above is trace-length-independent for efficiently (and thus also very efficiently) monitorable LOLA specifications. Recall that an edge from a stream identifier x to y labeled with number k in the dependency graph means that the value of an event on stream y may depend on an event of stream x, k instants in the future. Clearly, a path in the dependency graph expresses a (mediate) dependency between events on two not necessarily directly connected streams and the sum of the path's edge weights indicates the temporal relation. Thus, a positive cycle connotes that the events of a certain stream depend on later events of the same stream, which in turn depend on later events of the stream themselves. Hence, such a chain of dependencies may never be resolved unless the end of the stream is reached. On the other hand, if the dependency

graph does not contain such positive cycles, there is a constant c, such that all paths in the dependency graph have an edge sum lower than c. Consequently, every equation can be evaluated at the latest c instants after being added to A. Additionally, a fully evaluated equation is removed at most d instants after being added, where $-d$ is the (constant) lowest negative offset accessing its corresponding stream. Thus, for specification $\varphi = (S, I, O, E)$ the number of equations in A is limited by $(|S| + |I|) \cdot \max\{c, d\} = \mathit{const}$. The maximal size of an equation is determined by the corresponding stream definition in the specification and thus also constant. The very efficiently monitorable fragment simplifies the monitoring process even more. All equations can be evaluated directly at the instant they are added because they depend only on the values of current and previous stream events. As a result, it is not necessary to store the complete equations, but only the values of the instant variables.

Further LOLA fragments

In the later sections of this thesis, we will also examine additional LOLA fragments which are restricted with respect to the supported types, function symbols, and shape of stream definitions, rather than the dependency graph. Specifically, we deal with the following fragments, for which we will show special properties with respect to their trace-length-independent monitoring:

Definition 2.43 (LOLA fragments; based on [KLS22a])**.**

The following syntactic fragments of LOLA are defined:

- **Boolean fragment** ($\text{Lola}_{\mathbb{B}}$) is the LOLA fragment, where the data domain of all streams is the boolean domain $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$ and the available functions are $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$.
- **Linear algebra fragment** ($\text{Lola}_{\mathcal{LA}}$) is the LOLA fragment, where the data domain of all streams are real numbers $\mathbb{R}$ and every stream definition has the form $c_0 + c_1 \cdot s_0[o_1|d_1] + \cdots + c_n \cdot s_n[o_n|d_n]$ where $c_i, d_i \in \mathbb{R}$ are constants.
- **Linear real arithmetic fragment** ($\text{Lola}_{\mathcal{LRA}}$) is the LOLA fragment, where the data domain of all streams is $\mathbb{B}$ or $\mathbb{R}$. Every stream definition is either contained in the linear algebra fragment or in the boolean fragment extended by the functions $<$, $>$, $\leq$, $\geq$ and $=$ for reals.

These fragments are of course combinable with the efficient monitorability property from above and consequently also an efficiently monitorable boolean fragment etc. exists. Observe that the induced algebras of specifications over these fragments correspond to (subsets of) the algebras introduced in section 2.1.5.

LOLA 2.0 and RTLola

As the first and pioneering formalism in the field of stream runtime verification, traditional LOLA [DSS+05] as presented in the previous section was basis for two strongly related languages or dialects, Lola 2.0 [FFST16] and RTLola [FFS+19, FFST17], which shall briefly be discussed in this section.

Lola 2.0 extends the traditional LOLA syntax by a notion of parameterized streams, so-called *stream template* expressions. Compared to normal streams they bring two additional features:

- The stream templates represent a dynamic, potentially infinite number of homogeneous instance streams, which are parameterized in the values of another stream. A stream template could for example define an infinite set of streams where each one counts the number of occurrences of a specific value in another stream.
- The event rate of each instance stream spawned by a template expression is determined by a specific boolean stream. If this boolean stream carries the value `true` the template stream has an event, at other instants it has not. Hence template expressions allow for asynchronous streams in the sense that not all streams have events at all instants.

However, from a formal perspective these features provided by Lola 2.0 do not increase the expressiveness of LOLA [Sch22]. Since the data types of LOLA are not restricted, a stream of type map can be used to represent all streams generated by a template expression. Therefore the map can store the current value for each stream with the template parameter as key. Always when the computation of an instance stream is triggered, the corresponding value in the map can be updated (see [Sch22]).

Clearly the resulting monitor would not be trace-length-independent in terms of its memory requirement anymore, because the map could grow arbitrarily large and consequently not be stored in a single memory cell (as demanded before). Yet this is a general problem. Lola 2.0 can principally not be monitored in a trace-length-independent manner as long as there is no bound on the number of invoked instance streams and future references are allowed [FFST16]. Otherwise, if there is a bound on the number of instance streams, all possible instance streams can also directly be defined in a LOLA specification (though this is less comfortable of course).

RTLola, short for real-time Lola, is based on Lola 2.0 and extends it with real-time features [FFS+19, FFST17]. To do so it first and foremost attaches real-valued timestamps from a global clock to events, thus operates on timed streams (definition 2.6). This way input streams may have arbitrary and various event rates. Streams can either be computed in an event-driven manner (i.e. whenever other streams they depend on have an event) or on a regular basis (denoted by

a frequency annotation at the stream definition) [FFS+19]. Additionally RTLola provides a special syntax for offsets referring to events a certain amount of time in the past and for aggregating a bunch of events in a past-only sliding window of a fixed time duration. Note that such a sliding window may contain arbitrarily many events (depending on the event rate of the aggregated stream). Finally, as extension of Lola 2.0, RTLola also supports the concept of template streams as described above.

From a theoretical point of view these features can also be mimicked by a traditional LOLA specification [Sch22]. The timestamps assigned to the events can be expressed by a global `time` stream which carries the current wall-clock time. The stream is externally filled with events for all timestamps, where either an input event appears or where the computation of an output stream is triggered by a frequency annotation. Also the aggregations and timed offsets can be implemented in standard LOLA by the use of map data structures where the values and timestamps within the corresponding time window are maintained. This map can then be used for aggregation or to retrieve a stream event some time in the past.

The implementation of RTLola, StreamLAB [FFS+19], also contains some additional useful features, especially a powerful type system which ensures runtime and memory of the monitor to be bounded (i.e. it enforces trace-length-independence). Therefore it prevents the user from writing specifications where the number of events in a sliding window is unlimited. The StreamLAB implementation is further also capable of providing a worst-case estimation of the monitor's storage requirements to enable a safe execution on systems with restricted memory and computation capabilities.

In this thesis we will not further consider the languages Lola 2.0 and RTLola. Since both can be mimicked by traditional LOLA specifications, the monitoring algorithm presented in chapter 4 is in principle also suitable for them, but might suffer from performance issues. The question of how to directly and efficiently adapt the monitoring algorithm to Lola 2.0 and RTLola is left for future work.

2.3.2. TeSSLa

TeSSLa, short for temporal stream-based specification language, may be considered as an extension of LOLA towards asynchronicity. TeSSLa was in its basic form introduced by Leucker et al. in [LSS+18], and revised to its current version in [CHL+18]. The corresponding tool chain implementation based on the language is available as open source project and described in [KLS+22b]. Again, as TeSSLa will not be used directly in the rest of this thesis, only a brief overview of the TeSSLa features will be given.

In difference to LOLA, TeSSLa is based on infinite timed streams (see definition 2.6), i.e. every event of the stream has a timestamp from a (possibly non-discrete) time domain $\mathbb{T}$, e.g. $\mathbb{R}$, attached.

Note that previously it was argued in section 2.1.3 that timed streams over non-discrete time domain (e.g. signals), can be approximated by ($\mathbb{T} \times \mathbb{D}$) sequences (i.e. timed traces or discrete streams), which TeSSLa makes use of. TeSSLa by default interprets a received input sequence as timed stream with gaps. In fact the standard TeSSLa implementation [KLS+22b] also contains special syntactic sugar (*signal lift* [CHL+18]) to interpret discrete (input) streams as piece-wise constant signals. This way TeSSLa can also be used for specification and computation on piece-wise constant signals.

As in the case of LOLA, a TeSSLa specification transforms a set of typed input streams to a set of typed output streams. Therefore it is given as an equation system, where each output stream has a defining expression of five core operators assigned.

In TeSSLa, the defined output streams can have events (i.e. letters in the representing timed trace) at an irregular selection of timestamps that differs from those of the input streams (by use of the so-called `delay` operator). In this sense, it differs from LOLA, where the specification only allows output events to be synchronous with input events, and is thus considered more expressive than LOLA in this respect [Sch22] and is counted as an asynchronous stream runtime verification language [GS21b]. Besides, the stream and semantics definitions of TeSSLa also allow asynchronous evaluation in the sense that not all input streams need to be known up to the same timestamp (see [LSS+18, Sch24]).

However, with TeSSLa's ability to spawn events with fresh timestamps comes the possibility of *zeno behavior* [CHL+18]. This describes the circumstance that in TeSSLa one can specify an output stream with infinitely many events, which accumulate before a specific timestamp. Obviously, such a stream is not fully computable in practice and a corresponding monitor would never advance over the corresponding timestamp.

Unlike LOLA, TeSSLa is not able to refer to future events, but a corresponding extension is discussed in [Sch22]. In general every computable, (according to the prefix order of timed streams) continuous and future-independent stream transformation can be expressed in TeSSLa [CHL+18]. TeSSLa can be evaluated with an algorithm similar to (past-only) LOLA in a trace-length-independent manner (due to the missing future references) if only constant-sized data types are utilized.

2.3.3. Striver

Striver [GS18, GS21b], presented in 2018 by Gorostiaga et al. is an asynchronous SRV language comparable to TeSSLa, but with the support of future references. Additionally, provided non-zeno inputs, Striver also guarantees non-zeno output behavior by defining a maximal rate on which output streams may have events, when there are no input events. Thus, Striver can be seen as an extension of LOLA

for timed streams (comparable to RTLola). In fact Striver can, like LOLA 2.0 and RTLola, also be simulated in LOLA [GDS20]. Like the other SRV languages besides LOLA, Striver will only be dealt with marginally in the remainder of this work.

2.4. Fixed point computation and abstract interpretation

In this final section of the preliminaries chapter, the basics of fixed point computation and the field of abstract interpretation will be introduced. Fixed points play a fundamental role in the formal consideration of recursively defined problems. As theoretical framework for stream monitoring they will also be utilized in the remainder of this thesis.

Given a function with identical domain and co-domain $f : A \to A$, a fixed point is defined as an element of A which f maps to itself.

Definition 2.44 (Fixed point; based on [DP90])**.**

Let $f : A \to A$ be a function over domain A.

The set of *f's fixed points* $\text{fix}(f)$ is defined as

$$\text{fix}(f) = \{a \in A \mid f(a) = a\}.$$

An element $a \in \text{fix}(f)$ is called *fixed point* of function f.

Depending on f, $\text{fix}(f)$ can consist of a single fixed point, be empty or any other subset of f's domain. If f's domain A is a partially ordered set, it may (but does not have to) be the case, that a unique least or greatest fixed point exists.

Definition 2.45 (Least and greatest fixed point; based on [DP90])**.**

Let $f : A \to A$ be a function over partially ordered set $(A, \sqsubseteq)$.

The fixed point $\mu(f) \in \text{fix}(f)$ is called *least fixed point of f* if and only if

$$\forall a \in \text{fix}(f).\, \mu(f) \sqsubseteq a.$$

The fixed point $\nu(f) \in \text{fix}(f)$ is called *greatest fixed point of f* if and only if

$$\forall a \in \text{fix}(f).\, a \sqsubseteq \nu(f).$$

Least and greatest fixed points play a role as uniquely identifiable fixed points when $\text{fix}(f)$ contains multiple elements. Several fixed point theorems can be found in literature on existence and computation of extreme (i.e. least and greatest) fixed points. In this thesis we will use the fixed point theorems of Knaster and Tarski

and the one of Kleene. They are applicable if domain A is a complete lattice (see definition 2.8) and f is monotonic:

Definition 2.46 (Monotonic function; based on [DP90])**.**
Let $f : A \to B$ be a function over partial orders $(A, \sqsubseteq^A)$, $(B, \sqsubseteq^B)$.

Function f is called *monotonic* if and only if

$$\forall a, b \in A.\, a \sqsubseteq^A b \Rightarrow f(a) \sqsubseteq^B f(b).$$

The fixed point theorem of Knaster and Tarski states that if $f : A \to A$ is a monotonic function over a lattice domain then its fixed points also form a complete lattice.

Theorem 2.47 (Fixed point theorem of Knaster and Tarski; based on [Tar55])**.**
Let $f : A \to A$ be a monotonic function over complete lattice $(A, \sqsubseteq)$.

The ordered set $(\mathrm{fix}(f), \sqsubseteq)$ is a complete lattice.

If $(\mathrm{fix}(f), \sqsubseteq)$ is a complete lattice, this implies that f has a (unique) least and greatest fixed point.

Next we consider Kleene's fixed point theorem. Therefor f is required to be continuous (sometimes also referred to as *Scott-continuous*).

Definition 2.48 (Continuity; based on [DP90, Cou21])**.**
Let $f : A \to B$ be a function over complete lattices $(A, \sqsubseteq^A)$, $(B, \sqsubseteq^B)$.

Function f is called *upper continuous* if and only if for every upward-directed subset $D \subseteq A$

$$f\left(\bigsqcup\nolimits^A D\right) = \bigsqcup\nolimits^B \{f(d) \mid d \in D\}$$

and *lower continuous* if and only if for every downward-directed subset $D \subseteq A$

$$f\left(\bigsqcap\nolimits^A D\right) = \bigsqcap\nolimits^B \{f(d) \mid d \in D\}\,.$$

Observe that any upper or lower continuous function is monotonic. If a function is upper and lower continuous we call it continuous.

Kleene's fixed point theorem gives us a useful characterization of minimal and maximal fixed points of a continuous function.

Theorem 2.49 (Fixed point theorem of Kleene; based on [DP90, Cou21])**.**

Let $f : A \to A$ be a continuous[3] function over complete lattice $(A, \sqsubseteq)$.

The least and greatest fixed points of f are given as

$$\mu(f) = \bigsqcup_{n \geq 0} f^n(\bot^A) \qquad \text{and} \qquad \nu(f) = \bigsqcap_{n \geq 0} f^n(\top^A).$$

Recall that $\bot^A$ denotes the least and $\top^A$ the greatest element in A.

Considering the so-called ascending Kleene chain

$$s_0 = \bot^A \;\sqsubseteq\; s_1 = f(\bot^A) \;\sqsubseteq\; s_2 = f^2(\bot^A) \;\sqsubseteq\; \ldots$$

it follows directly from theorem 2.49 that if for any $i \in \mathbb{N}$, $s_i = s_{i+1}$ then $s_i = \mu(f)$ and the analogous for the greatest fixed point. This means the least (greatest) fixed point can be computed, starting with the least (greatest) element of A and iteratively applying the fixed point function. However note that the existence of such an i where the chain stabilizes is not guaranteed and thus the Kleene chain might be infinite, i.e. the iterative computation will not terminate. We call the resulting chain of function applications fixed point iteration.

Definition 2.50 (Fixed point iteration)**.**

Let $(A, \sqsubseteq)$ be a complete lattice and $f : A \to A$ a continuous function.

The sequences $(f^i(\bot^A))_{i \in \mathbb{N}}$ and $(f^i(\top^A))_{i \in \mathbb{N}}$ are called *lower* and *upper fixed point iteration* of f. A fixed point iteration $(s_i)_{i \in \mathbb{N}}$ is called *finite* if for some $j \in \mathbb{N}$ $s_j = s_{j+1}$.

As mentioned, for a lower fixed point iteration $(s_i)_{i \in \mathbb{N}}$ of a continuous function f, $\sqcup_{i \in \mathbb{N}} s_i = \mu(f)$ and $\sqcap_{i \in \mathbb{N}} s_i = \nu(f)$ for an upper one. If the iteration is finite, $\mu(f)$ and $\nu(f)$ are equal to the element stabilizing the iteration.

The requirement of Kleene's fixed point theorem that function f is continuous is relatively strong. Yet, if the limit of a fixed point iteration is a fixed point, then it is still the least or greatest fixed point even if the function is just monotonic and not continuous. This relaxed version of Kleene's fixed point theorem is formulated in theorem 2.51. We will later use this theorem instead of theorem 2.49, as due to the restriction to finite stream lengths, our fixed point computations will be guaranteed to be finite.

[3] If one is only interested in the least fixed point it is sufficient that the function is upper continuous; for the greatest fixed point f must be lower continuous.

Theorem 2.51 (Relaxed fixed point theorem; based on [DP90])**.**
Let $f : A \to A$ be a monotonic function over complete lattice $(A, \sqsubseteq)$.

$$\text{If } \bigsqcup_{n\geq 0} f^n(\bot^A) \in \text{fix}(f) \text{ then } \mu(f) = \bigsqcup_{n\geq 0} f^n(\bot^A) \text{ and}$$

$$\text{if } \bigsqcap_{n\geq 0} f^n(\top^A) \in \text{fix}(f) \text{ then } \nu(f) = \bigsqcap_{n\geq 0} f^n(\top^A).$$

2.4.1. Recursive computations as fixed point equations

Fixed point equations can be considered as canonical description of recursively defined problems and structures. The fixed points are consequently solutions to these problems and fixed point theorems may yield algorithmic procedures for their computation or approximation. In this subsection two typical examples of problems which are characterized via fixed point equations are presented.

One of the canonical examples is the Fibonacci sequence (a similar example for the factorial function can be found in [DP90]). This is a sequence where the first element is 0, the second one is 1 and all other elements are the sum of their two predecessors. We can phrase this condition as fixed point equation $F_{fib} : \mathbb{N}^\omega \to \mathbb{N}^\omega$:

$$F_{fib}(s)(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ s(n-1) + s(n-2) & \text{else} \end{cases}$$

Note that any fixed point of F_{fib} (i.e. some $s \in \mathbb{N}^\omega$ s.t. $F_{fib}(s) = s$) is a valid Fibonacci sequence. In fact F_{fib} only has a unique fixed point, $s = 0, 1, 1, 2, 3, 5, 8, \ldots$.

Note that in a similar way, an SRV specification (see section 2.3) can also be understood as a fixed point equation on the involved streams, which are essentially equivalent to (potentially) recursively defined sequences. A fixed point-based semantics for the SRV language LOLA will be presented in chapter 4.

Another typical use case of fixed point equations is the definition of program semantics, especially for their formal analysis e.g. in the field of static analysis. Consider the following imperative algorithm for computation of the greatest common divider (Euclidean algorithm[4]) where a, b are integer variables:

[4] Version from `https://en.wikipedia.org/w/index.php?title=Euclidean_algorithm&oldid=1192334861`, based on [Knu97]

```
1  a = userInput; //no negative input allowed
2  b = userInput; //no negative input allowed
3
4  while a ≠ b do
5      if a > b then
6        a = a − b;
7      else
8        b = b − a;
9      fi;
10 od;
11
12 Output a;
```

If one wants to reason about possible values of variables at certain code locations, e.g. to proof that the value of a at line 12 cannot be negative, one is usually interested in a *static* (also called *collecting*) semantics of the program [CC77, CC92]. The static semantics of a program can be a map assigning to every program location the set of possible variable valuations before execution of the statement at this location (see [CC77]). In the case of the example above such a semantics is an element $S \in \mathbf{Sem} := (\mathbb{L} \to 2^{\mathbb{Z}_\perp \times \mathbb{Z}_\perp})$ where $\mathbb{Z}_\perp = \mathbb{Z} \cup \{\perp\}$ extends the integers with element $\perp$ to indicate that the corresponding variable is not yet set and $\mathbb{L} = \{1, 2, 4, 5, 6, 8, 12\}$ is the set of program locations (we use the line numbers where a statement starts). A semantics with $S(4) = \{(\perp, 2), (\perp, 3)\}$ would for example indicate that before the execution of line 4, variable a is not set and b is either 2 or 3.

For an imperative program like the one above, the possible valuations at each location can be obtained by application of all possible predecessor operations (under consideration of their conditions in case of if and while) on the valuations at their location. E.g. the possible valuations at line 4 are those resulting from the assignment in line 2 and the subtractions in line 6 and 8. The static semantics can be derived from the fixed point equation $sem : \mathbf{Sem} \to \mathbf{Sem}$ which encodes the source code of the program:

$$
\begin{array}{lcl}
sem(S)(1) & = & \{(\perp, \perp)\} \\
sem(S)(2) & = & \{(z, y) \mid z \in \mathbb{N}, (x, y) \in S(1)\} \\
sem(S)(4) & = & \{(x, z) \mid z \in \mathbb{N}, (x, y) \in S(2)\} \cup \{(x - y, y) \mid (x, y) \in S(6)\} \cup \\
 & & \{(x, y - x) \mid (x, y) \in S(8)\} \\
sem(S)(5) & = & \{(x, y) \mid (x, y) \in S(4) \wedge x \neq y\} \\
sem(S)(6) & = & \{(x, y) \mid (x, y) \in S(5) \wedge x > y\} \\
sem(S)(8) & = & \{(x, y) \mid (x, y) \in S(5) \wedge x \leq y\} \\
sem(S)(12) & = & \{(x, y) \mid (x, y) \in S(4) \wedge x = y\}
\end{array}
$$

For this concrete example, *sem* has the following fixed point S:

$$\begin{array}{lcl} S(1) & = & \{(\bot, \bot)\} \\ S(2) & = & \{(x, \bot) \mid x \in \mathbb{N}\} \\ S(4) & = & \{(x, y) \mid x, y \in \mathbb{N}\} \\ S(5) & = & \{(x, y) \mid x, y \in \mathbb{N} \wedge x \neq y\} \\ S(6) & = & \{(x, y) \mid x, y \in \mathbb{N} \wedge x > y\} \\ S(8) & = & \{(x, y) \mid x, y \in \mathbb{N} \wedge x < y\} \\ S(12) & = & \{(x, x) \mid x \in \mathbb{N}\} \end{array}$$

However, this is not the only fixed point of *sem*. In fact, one could allow that either a or b is from $\mathbb{Z}$ instead of $\mathbb{N}$ in the valuations of all locations inside the loop. The additional valuations would then condition themselves through the loop and thus also lead to a fixed point. However, for these assignments no valuations outside of the loop exist that would cause them to occur during real program execution. Thus, for the static semantics we are always interested in the least fixed point w.r.t. the order $\sqsubseteq$ where $S \sqsubseteq S' \Leftrightarrow \forall l \in \mathbb{L} : S(l) \subseteq S'(l)$, i.e. the one which contains no superfluous valuations. The fixed point given above is the least fixed point of *sem*.

2.4.2. Abstract fixed point computation

A standard approach in computer science for finding least an greatest fixed point is to compute the corresponding Kleene chain until the desired fixed point is reached. As mentioned before this is not always possible, as for some functions the Kleene chain is infinite (e.g. in the Fibonacci example above).

An alternative approach, when fixed point computation does not terminate or is too costly, is to to apply the principle of *abstraction*, i.e. to perform the computation inside another domain that is easier to handle because some information is abstracted away.

As an example consider again the static semantics of the Euclidean algorithm from above, where the solution of the fixed point equation was an element of $\mathbf{Sem} := (\mathbb{L} \to 2^{\mathbb{Z}_\bot \times \mathbb{Z}_\bot})$, assigning a set of possible variable valuations to each program location. On the one hand the representation of elements from $\mathcal{S}$ in memory is complicated. On the other hand, the Kleene chain of the static semantics is not necessarily finite for all programs and the solution cannot be computed iteratively then.

An alternative abstract domain would be $\mathbf{Sem}^\sharp := (\mathbb{L} \to \mathbb{I}_\mathbb{Z} \times \mathbb{I}_\mathbb{Z})$ where every program variable at each location gets an interval of possible values assigned. It is easy to see that this domain is easier to represent in memory and to calculate on: All operations only have to be performed on the interval's borders. However, this ease of use comes at the expense of accuracy: In this domain, it would no longer be possible to conclude that all values of a and b in line 12 are positive as it is the case in the original domain . This is because the relation $a > b$ or $b > a$ before

execution of lines 6,8 respectively cannot be represented in this abstract domain and thus the resulting interval for a or b after the execution would be $]-\infty,+\infty[$. In other examples, however, the use of this interval domain might be sufficient to infer certain program properties. The art of fixed point abstraction is thus to find a suitable abstract domain that is precise enough for the desired purpose and easy to compute with.

Key entities of abstraction are the concretization and abstraction function, translating between the involved concrete and abstract domains and the abstract fixed point equation. If certain conditions on them are met, it is guaranteed that the abstract fixed point is an approximation of the concrete one and the abstraction is precise enough to decide certain properties on. In 1977, Patrick and Radhia Cousot presented a formal framework called *abstract interpretation* [CC77] which enables the construction of sound or perfect fixed point approximations over abstract domains. The approach is mainly used for the static analysis of programs, but due to its generality it is not limited to this. The monitoring algorithm which will be discussed in chapter 4 is also based on the idea of abstract specification computation and utilizes concepts from abstract interpretation.

While abstract interpretation can be defined as a very general framework (see [CC92]), we rely in this thesis on the traditional Galois connection framework [CC77, CC92], which generally imposes too strict requirements on the involved operations and domains, but is suitable for our purposes.

Abstract interpretation is a well studied subject in literature. As such, a lot of useful abstract domains for certain purposes have already been investigated and implemented. Prominent ones are e.g. interval domains [CC77, Cou21], linear restraint sets (convex polyhedra) [CH78] and octagons [Min01, Cou21]. While the first only provides information about each program variable individually, the latter ones are also able to represent their relation.

Abstract interpretation framework

In the following let $(C, \sqsubseteq^C)$ and $(A, \sqsubseteq^A)$ be two complete lattices called the concrete and abstract domain. Let further $\alpha : C \to A$ be the abstraction and $\gamma : A \to C$ the concretization function translating between both domains and $f : C \to C$, $f^\sharp : A \to A$ the fixed point equations in the concrete and abstract.

In order to prove the soundness (i.e. over-approximation) or perfection of the abstract fixed point computation w.r.t. the concrete one, we have to make some assumptions about the translation functions.

In the Galois connection framework we require α, γ to form a Galois connection.

Definition 2.52 (Galois connection; based on [Ore44, Cou21])**.**

Let $(C, \sqsubseteq^C)$ and $(A, \sqsubseteq^A)$ be two partially ordered sets.

A pair of functions $\alpha : C \to A$, $\gamma : A \to C$ is called *Galois connection* (denoted $C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A$), if and only if for all $a \in A$ and $c \in C$

$$\alpha(c) \sqsubseteq^A a \Leftrightarrow c \sqsubseteq^C \gamma(a)$$

A Galois connection ensures monotonicity among two partially ordered sets: If an element a from domain A is greater than or equal to the picture of an element $c \in C$ in A, then the picture of a in C is also greater than or equal to c. This means that translating an element $c \in C$ to domain A, increasing it and translating it back to domain C leads to an element greater than or equal to c. This implies that $c \sqsubseteq^C \gamma(\alpha(c))$ for any $c \in C$, i.e. the direct translation to the abstract domain and back may not lead to a smaller element in the concrete domain. Likewise $\alpha(\gamma(a)) \sqsubseteq^A a$ holds for any $a \in A$ in a Galois connection.

In abstract interpretation it is a common approach to form an abstract domain as Cartesian product of several abstract domains. This can either be done to abstract several components of a fixed point computation or to perform several abstract computations in parallel. An abstract domain which is crafted this way inherits some of the properties from the sub-domains, as shown in lemma 2.53.

Lemma 2.53 (Cartesian abstract domains; based on [Cou21])**.**

Let $(A_1, \sqsubseteq^1), \ldots, (A_n, \sqsubseteq^n)$ be complete lattices which are connected to complete lattices $(C_1, \preceq^1), \ldots, (C_n, \preceq^n)$ by the Galois connections $C_i \underset{\alpha_i}{\overset{\gamma_i}{\leftrightarrows}} A_i$ for all $i \in \{1, \ldots, n\}$.

The domains $A = A_1 \times \cdots \times A_n$ and $C = C_1 \times \cdots \times C_n$ are complete lattices with orders $\sqsubseteq^A \in A \times A$ and $\preceq^C \in C \times C$, s.t.

$$\begin{array}{lcl} (a_1, \ldots, a_n) \sqsubseteq^A (a'_1, \ldots, a'_n) & \Leftrightarrow & \forall i \in \{1, \ldots, n\}.\, a_i \sqsubseteq^i a'_i \\ (c_1, \ldots, c_n) \preceq^C (c'_1, \ldots, c'_n) & \Leftrightarrow & \forall i \in \{1, \ldots, n\}.\, c_i \preceq^i c'_i \end{array}$$

The translation functions $\alpha : C \to A$ and $\gamma : A \to C$, s.t.

$$\begin{array}{lcl} \alpha((c_1, \ldots, c_n)) & = & (\alpha_1(c_1), \ldots, \alpha_n(c_n)) \\ \gamma((a_1, \ldots, a_n)) & = & (\gamma_1(a_1), \ldots, \gamma_n(a_n)) \end{array}$$

form a Galois connection which connects A and C.

When it comes to abstract fixed point computation, we require – besides consistency of the translation functions – also the fixed point equations f and $f^\sharp$ to be related.

Since we (usually) want to over-approximate $\mu(f)$ or $\nu(f)$, we demand that for any abstract element $a \in A$, $f^\sharp(a)$ translated back to the concrete domain does not under-approximate the result of f applied to $\gamma(a)$. This property is visualized on the left side of figure 2.11.

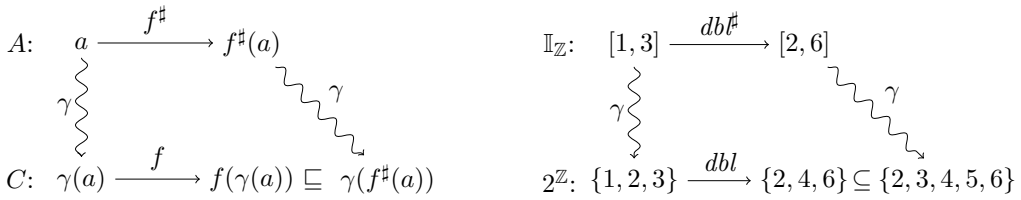


Figure 2.11.: General scheme of abstraction (left) and concrete example (right).

Formally we define this relation in definition 2.54.

Definition 2.54 (Abstraction; based on [CC77])**.**

Let $(C, \sqsubseteq^C)$ and $(A, \sqsubseteq^A)$ be complete lattices.

For a pair of abstraction and concretization functions $\alpha : C \to A$, $\gamma : A \to C$, that form a Galois connection $C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A$, a monotonic function $f^\sharp : A \to A$ is called *(sound) abstraction* of monotonic function $f : C \to C$ if and only if

$$\forall a \in A.\, f(\gamma(a)) \sqsubseteq^C \gamma(f^\sharp(a))$$

An example of a sound abstraction is illustrated on the right side of figure 2.11: Imagine the abstraction of a single program variable of type integer at a specific program location (the example can easily be generalized for more variables and locations) and an operation *dbl* which doubles the value. In the example let the concrete domain be the power set domain reflecting all possible values of the variable. The abstract domain is chosen to be the interval domain, where a set of values is abstracted by the interval between its infimum and supremum. Note that the domains and corresponding translations form a Galois connection.

Consider the abstract element $[1, 3]$ which corresponds to the concrete set $\{1, 2, 3\}$. The operation *dbl*, operating in the power set domain, multiplies all possible values in the set with two. An abstract double function for intervals would instead multiply the boundaries of the interval with two and obtain $[2, 6]$. The concretization of this interval however, $\{2, 3, 4, 5, 6\}$, would be an over-approximation of the actual value $\{2, 4, 6\}$ which results from the application of *dbl*, i.e. the abstraction would introduce the non possible variable values 3 and 5.

The literature on abstract interpretation provides a large number of theorems how the least and greatest fixed points in the abstract and concrete domain are related under which circumstances (see [Cou21]). Later in this thesis however we will aim at

over-approximation of the least fixed point, thus we concentrate on the results about perfect and over-approximative abstraction of the least fixed point in theorem 2.55.

Theorem 2.55 (Fixed point abstractions; based on [CC77, Cou21])**.**

Let $(C, \sqsubseteq^C)$, $(A, \sqsubseteq^A)$ be complete lattices connected by a Galois connection $C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A$. Let further $f : C \to C$ be a monotonic function and $f^\sharp : A \to A$ an abstraction of f.

It holds that

- $\alpha(\mu(f)) \sqsubseteq^A \mu(f^\sharp)$,
- $\alpha(\mu(f)) = \mu(f^\sharp)$ if for all $c \in C$, $\alpha(f(c)) = f^\sharp(\alpha(c))$.

Consequently computing the fixed point of $f^\sharp$ yields us an over-approximation or even the prefect representation of $\mu(f)$ in the abstract lattice. We call the second case a *perfect abstraction*.

If $f^\sharp$ is upper continuous we additionally can compute $\mu(f^\sharp)$ with a fixed point iteration. However $\mu(f^\sharp)$ is like $\mu(f)$ not guaranteed to be computable in finitely many steps. Yet it can be the case if the abstract domain is chosen wisely. For example domains without infinite ascending chains (called *ascending chain condition* [Cou21]) would guarantee a finite lower fixed point iteration for $f^\sharp$ (even if $f^\sharp$ is not continuous).

For the common case of over-approximation of least fixed points, abstract interpretation delivers two strategies called *widening* and *narrowing* to accelerate the fixed point computation [CC77, Cou21]. Widening aims at shorter fixed point iterations by extrapolating from the changes of the previous iterations until computation surpasses the least fixed point. Narrowing, which is often applied after the widening, tries to approach the least fixed point again from above without stepping over it.

2.4.3. Usage of abstractions in runtime verification

Abstract interpretation is a widely used technique in static analysis, yet, due to its generality not limited to this area. In [LSS+19] it is used in runtime verification to handle uncertainties on the input streams. Due to the close connection to the topic of this thesis, this section provides a brief overview of the core idea behind the approach.

Specifically, [LSS+19] presents an abstraction-based technique for handling uncertain inputs in the stream runtime verification language TeSSLa. The semantics of a TeSSLa specification φ is defined as the stream transformation given by the fixed point equation

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

There, $s_1, \ldots, s_k$ are the specification's input streams of types $\mathcal{S}_{\mathbb{D}_1}, \ldots, \mathcal{S}_{\mathbb{D}_k}$ and $e_1, \ldots, e_n$ denote the defining expressions of φ's intermediate streams of types $\mathcal{S}_{\mathbb{D}'_1}$, $\ldots$, $\mathcal{S}_{\mathbb{D}'_n}$ with the corresponding recursive semantics $[\![\cdot]\!]_{s_1,\ldots,s_k} : \mathcal{S}_{\mathbb{D}'_1} \times \ldots \times \mathcal{S}_{\mathbb{D}'_n} \to \mathcal{S}_{\mathbb{D}'_1} \times \ldots \times \mathcal{S}_{\mathbb{D}'_n}$ for fixed input streams $s_1, \ldots, s_k$ [CHL+18].

The idea used in the mentioned approach is to introduce a notion of abstract streams which may contain gaps (i.e. time intervals where the stream is fully unknown) and uncertain events. Further a Galois connection for the translation between sets of concrete and an abstract stream is defined. Based on this, perfect abstractions of all TeSSLa operators are given. By principles of abstract interpretation, this allows the computation of the specification's output in the abstract, i.e. uncertain, domain.

Contrary to traditional abstract interpretation, the approach from [LSS+19] renounces to apply widening or other methods to force a finite fixed point computation. Instead it performs iterations until a fixed point is reached (the same holds for concrete TeSSLa). This is due to the special usage of the fixed points in the monitoring setting: The monitor receives a finite prefix of the input streams up to a certain timestamp (called *progress*) and, based on this, determines the output streams (via fixed point iterations) up to the same timestamp. If new events on the input streams are read and their progress increases, the fixed point computation can be commenced from the previously computed fixed point. This is possible as TeSSLa specifications always express monotonic stream transformations, i.e. such where a continuation of the input stream prefixes also leads to a continuation of the output stream prefixes. The monitoring process itself is thus equal to the iterative computation of a fixed point.

However, as the abstract domain of the approach is basically a Cartesian combination of the different abstract stream domains, the abstract computation of the whole specification is sound but not perfect anymore. The reason for this is, that due to the individual abstraction of each stream, the interconnections between events on different streams get lost.

3

A generalized monitoring theory

In chapter 2 several well-known monitoring approaches for different kinds of logics and specification formalisms have been presented. In this chapter, we will now define a unified framework for synchronous online monitoring, i.e. where the monitor casts outputs whenever it receives inputs, including the handling of uncertainty and assumptions. This framework forms a common basis for all synchronous monitoring approaches from the previous section, including synchronous stream runtime verification which will even be shown to be a general formalism for this kind of properties. In this context also a translation of LTL, M(I)TL and STL for discrete time domains to LOLA will be presented. The following two chapters will then discuss and develop a generic and a concrete, symbolic implementation of the proposed monitoring framework for LOLA.

3.1. Monitoring

In the following we distinguish between synchronous and asynchronous monitoring (see [GS21b]). *Synchronous monitoring* describes the setting where the monitor casts outputs only at instants, where inputs are fed to the monitor. Yet in *asynchronous monitoring* the outputs of the monitor can be produced at arbitrary locations without inputs as well. In this thesis we focus on synchronous monitoring, although at some points we will discuss how approaches would need to be adapted for asynchronous monitoring, to hint at possible future work.

A synchronous monitor in our setting basically resembles a Moore machine from definition 2.18. It receives a sequence of inputs and synchronously produces a sequence of outputs, determined by the current state. However, unlike a Moore machine, we do not require general synchronous monitors to have a finite state-space, nor a finite in- and output domain.

Definition 3.1 (Synchronous monitor; based on [KLSS22])**.**

A *synchronous monitor* $\mathcal{M} = (S, \Sigma, \mathbb{V}, \delta, \gamma, s_0)$ is a 6-tuple of

- a non-empty set of states S,
- a non-empty input domain Σ,
- a non-empty output domain $\mathbb{V}$,
- a deterministic transition function $\delta : S \times \Sigma \to S$,
- an output function $\gamma : S \to \mathbb{V}$,
- an initial state $s_0 \in S$

A sequence of states $s_0, s_1, \ldots, s_n$ with $s_i \in S$ is called a *run* of $\mathcal{M}$ for $w \in \Sigma^*$, if and only if

$$s_{i+1} \in \delta(s_i, w(i)) \text{ for all } i \in \mathbb{N}.$$

The sequence $\gamma(s_0), \gamma(s_1), \ldots \gamma(s_n)$ associated to this run is called *output* of $\mathcal{M}$. The final output of a monitor for w is denoted with $\mathcal{M}(w) := \gamma(s_n)$.

The extension to infinite domains and state space is necessary for monitoring formalisms with higher expressivity, such as LOLA, but for example also for the support of real-valued (multi-channel) signals as used in STL.

We will now deal with the question under which circumstances a monitor's output can be considered sound or perfect in terms of the monitored property. Therefore we will distinguish between initial and pointwise properties. Initial properties assign a single value to a full trace while point-wise properties assign a value to every location in the trace. While traditional RV approaches focused mainly on monitoring initial properties, pointwise properties have received increasing attention in recent times.

In the following sections, an extension of the results from [KLSS22, HKLS24] is presented, where different (partly novel) approaches for monitoring both kinds of properties are compared and their advantages and disadvantages are discussed. Thereby in particular, the notion of impartial and anticipatory monitoring, as known from the LTL_3 approach, is extended to the synchronous monitoring of initial and pointwise properties in general.

3.2. Initial monitoring

We start with the discussion of initial properties. We define such a property as a function assigning a value (called *valuation*) to every finite or infinite trace over a given input domain Σ. Again we keep the definition very general by not restricting input or value domain to be finite.

Definition 3.2 (Initial property)**.**

A function $\mathcal{P} : \Omega \to \mathbb{D}$ with $\Omega \in \{\Sigma^*, \Sigma^n, \Sigma^\omega, \Sigma^\infty\}$ where Σ is an input domain and $\mathbb{D}$ a value domain is called *initial property*.

We call $\mathcal{P}$ a *finite initial property* if and only if $\Omega \subseteq \Sigma^*$ and an *infinite initial property* if and only if $\Omega = \Sigma^\omega$.

Let φ be an LTL, MTL, MITL or STL formula, then the initial semantics of the corresponding logics are initial properties according to definition 3.2 for any time domain $\mathbb{T}$, as the following table shows:

Logic	Semantics	$\Sigma =$	$\Omega =$	$\mathbb{D} =$
LTL (def. 2.14)	Def. 2.16	2^{AP}, AP finite	$\Sigma^{*/n/\omega/\infty}$	$\mathbb{B}$
MTL (def. 2.21)	Def. 2.23	2^{AP} or $\mathbb{T} \times 2^{\mathrm{AP}}$, AP finite	$\Sigma^{*/n/\omega/\infty}$	$\mathbb{B}$
MITL (def. 2.24)	Def. 2.23	2^{AP} or $\mathbb{T} \times 2^{\mathrm{AP}}$, AP finite	$\Sigma^{*/n/\omega/\infty}$	$\mathbb{B}$
STL (def. 2.25)	Def. 2.27	$\mathbb{R}^m$ or $\mathbb{T} \times \mathbb{R}^m$	$\Sigma^{*/n/\omega/\infty}$	$\mathbb{B}$

For M(I)TL and STL we thereby assume an encoding of the timed stream or signal as finite or infinite (timed) trace. While this is no problem for discrete time domains, non-discrete ones require some kind of representation strategy. For details on the encoding of streams as timed traces, see section 2.1.3.

In general for a set $M \subseteq \Sigma^{*/n/\omega/\infty}$, the membership problem is an initial property with $\mathbb{D} = \mathbb{B} = \{\texttt{true}, \texttt{false}\}$. However, more advanced functions like $\#_a : \Sigma^* \to \mathbb{N}$, which counts the number of occurrences of $a \in \Sigma$ in w, also fit the definition of initial properties.

Monitoring initial properties now consists in the task of subsequently receiving input letters (thus prefixes of the full trace) and casting outputs which yield information about possible property valuations for the full trace. LTL_3 monitoring (see section 2.2.2) for example does so by giving verdicts from $\mathbb{B}_3 = \{\top, \bot, ?\}$, which express potential outcomes of the language containment problem ($\top \mathrel{\widehat{=}} \{\texttt{true}\}$, $\bot \mathrel{\widehat{=}} \{\texttt{false}\}$, $? \mathrel{\widehat{=}} \{\texttt{true}, \texttt{false}\}$).

We now generalize this idea by requiring an output interpretation function $\gamma_{\mathbb{V}} : \mathbb{V} \to 2^{\mathbb{D}}$, which translates a monitor output from $\mathbb{V}$ to the set of corresponding property values in $\mathbb{D}$ it represents. Based on such a mapping we can define a notion of sound and perfect monitoring of an initial property:

Definition 3.3 (Initial monitoring; based on [KLSS22])**.**

Let $\mathcal{P} : \Omega \to \mathbb{D}$ (with $\Omega \in \{\Sigma^*, \Sigma^n, \Sigma^\omega, \Sigma^\infty\}$) be an initial property. Let further $\mathcal{M} = (S, \Sigma, \mathbb{V}, \delta, \gamma, s_0)$ be a synchronous monitor with output interpretation $\gamma_{\mathbb{V}} : \mathbb{V} \to 2^{\mathbb{D}}$.

$\mathcal{M}$ is called

- *sound initial monitor* for $\mathcal{P}$ if and only if

$$\forall w \in \Sigma^* : \gamma_{\mathbb{V}}(\mathcal{M}(w)) \supseteq \{\mathcal{P}(wv) \mid wv \in \Omega\}$$

- *perfect initial monitor* for $\mathcal{P}$ if and only if

$$\forall w \in \Sigma^* : \gamma_{\mathbb{V}}(\mathcal{M}(w)) = \{\mathcal{P}(wv) \mid wv \in \Omega\}$$

Thus an initial monitor is said to be sound, if it only produces outputs that over-approximate the set of actually possible property valuations with respect to all conceivable continuations of the input. I.e. it does not embezzle possible outcomes. Further, a monitor is perfect if its outputs reflect exactly the possible property valuations. Note that the definition of soundness coincides with the definition of impartiality for the traditional RV word problem (see definition 2.13). Moreover a monitor is perfect in this setting if and only if it is impartial and anticipatory (see definition 2.13).

3.3. Pointwise monitoring

Initial properties and the corresponding monitoring are not perfectly suited for certain purposes. Therefore the notion and several variants of so-called recurrent monitoring will be introduced in this section. In the following we will first motivate the use of these monitoring techniques and subsequently consider the belonging formal definitions.

3.3.1. Motivation

An initial property assigns a single value to the whole (finite or infinite) trace. This way initial monitoring is not capable of identifying certain locations in the trace where the observed property is breached or satisfied. Likewise, it is not possible to detect all faults if the supervised property was violated more than once, which is for example interesting in offline RV, as used for log analysis.

For illustration consider the following example: Imagine a self-driving robot system with an extensible crane on its top. The ways on which the robot system is driving lead through underpasses of different height. To prevent potential damage to the crane, we want to be sure that the crane is always retracted when the system passes through an underpass. In (futrue) LTL we can specify this property in the following way:

$$\varphi = \mathit{extendCrane} \rightarrow (((\neg \mathit{underpass})\, \mathcal{U}\ \mathit{retractCrane}) \vee \neg \mathcal{F}\, \mathit{underpass})$$

I.e. it may not be the case that whenever the crane gets extended, there is an underpass before the crane is retracted again. Note at this point that the $\mathcal{U}$ would by definition require a *retractCrane* event to follow, so we explicitly add that never passing an underpass again for the rest of the trace ($\neg \mathcal{F}\, \mathit{underpass}$) is fine as well.

Since we want to enforce this policy over the whole run, the LTL formula of the initial property can be formulated as

$$\varphi' = \mathcal{G}(\mathit{extendCrane} \rightarrow (((\neg \mathit{underpass})\, \mathcal{U}\ \mathit{retractCrane}) \vee \neg \mathcal{F}\, \mathit{underpass})).$$

Figure 3.1 illustrates a sample run of the (initial) LTL_3 monitor (see section 2.2.2) for this property over $\Sigma = 2^{\{ec, rc, up\}}$, abbreviating the atomic propositions from the formula.

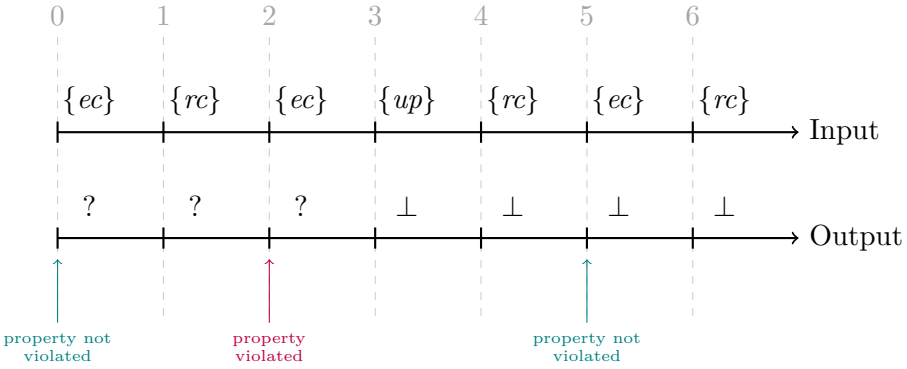


Figure 3.1.: Example run of initial LTL_3 monitor for $\mathcal{G}(ec \rightarrow (((\neg up)\, \mathcal{U}\, rc) \vee \neg \mathcal{F}\, up))$.

Note that the monitor yields the verdict ? until at instant 3 the input $\{up\}$ occurs, then the output switches to the final verdict $\bot$. From then on the monitor stays with this final verdict for all subsequent inputs it receives (which is typical for sound and perfect initial monitoring). Yet we actually want to know in this example at which locations in the trace the fault was caused, i.e. where the crane was extended despite a following underpass. Hence we actually want to check, at which locations of the trace φ (not φ') is satisfied and at which not, i.e. we want to evaluate the trace with respect to the pointed LTL semantics (definition 2.15). At position 0 this semantics would for example yield `true`, which indicates that the extension of the crane there

was safe. For instant 2 the value would be `false` and for instant 5 `true` again, which matches our desired monitor outputs. The outputs of the LTL_3 monitor however do not reflect these valuations at all, instead the outputs for instant 0 and 2 are exactly the same, and likewise all verdicts after position 2.

The given example suggests that initial monitoring as presented in the previous section is mainly suitable for incrementally checking a global property of a trace. Though for purposes like log analysis, bug tracking and explainability of faults or the continuation of monitoring, after a property violation was detected, a point-wise evaluation of a property might be more reasonable.

In the following section we will extend the definition of a property from definition 3.2 to the point-wise case and consider several monitoring strategies for such properties.

3.3.2. Pointwise properties and their monitoring

In difference to initial properties which assign a single data value to a finite or infinite trace, a pointwise property can assign such a value to every location in the trace:

> **Definition 3.4** (Pointwise property; based on [HKLS24])**.**
>
> A partial function $\mathcal{P} : \Omega \times \mathbb{N} \rightharpoonup \mathbb{D}$ with $\Omega \in \{\Sigma^*, \Sigma^n, \Sigma^\omega, \Sigma^\infty\}$ where Σ is an input domain and $\mathbb{D}$ a value domain is called a *pointwise property*.
>
> $\mathcal{P}$ is called a *finite pointwise property* if and only if $\Omega \subseteq \Sigma^*$ and an *infinite pointwise property* if and only if $\Omega = \Sigma^\omega$.

Note that a pointwise property may also be undefined for some positions, e.g. those behind the maximum trace length in the case of $\Omega = \Sigma^n$.

Again, the pointed semantics of LTL, MTL, MITL and STL match this definition for any discrete time domain $\mathbb{T}$:

Logic	Semantics	$\Sigma =$	$\Omega =$	$\mathbb{D} =$
LTL (def. 2.14)	Def. 2.15	2^{AP}, AP finite	$\Sigma^{*/n/\omega/\infty}$	$\mathbb{B}$
MTL (def. 2.21)	Def. 2.22	2^{AP} or $\mathbb{T} \times 2^{\text{AP}}$, AP finite	$\Sigma^{*/n/\omega/\infty}$	$\mathbb{B}$
MITL (def. 2.24)	Def. 2.22	2^{AP} or $\mathbb{T} \times 2^{\text{AP}}$, AP finite	$\Sigma^{*/n/\omega/\infty}$	$\mathbb{B}$
STL (def. 2.25)	Def. 2.26	$\mathbb{R}^m$ or $\mathbb{T} \times \mathbb{R}^m$	$\Sigma^{*/n/\omega/\infty}$	$\mathbb{B}$

For the initial properties of M(I)TL and STL we could allow any time domain (including non-discrete ones), as long as timed streams in this domain can be encoded as timed traces, i.e. sequences of data values and timestamps (see section 2.1.3). This is no longer possible for pointwise properties as they do not assign a single value

to the whole trace, but to every position instead. However, when encoding timed streams over non-discrete domain as timed traces, some of the timestamps from the stream's time domain are skipped in the timed trace and e.g. pretended to be the same as at the previous position (in case of piece-wise constant streams). Yet the semantics of M(I)TL and STL which is defined over streams and not timed words still provides values for all timestamps, including those which do not appear in the timed trace. Consequently the semantics of M(I)TL and STL on these words can no longer be considered as pointwise properties, as they assign valuations to timestamps for which no input letter exists. Pointwise properties, as defined in definition 3.4 only assign values to every trace location.

For the case of non-discrete time domains a further extension of definition 3.4 to *continuous* or *asynchronous properties*, i.e. those receiving any timed trace and assigning a value to every timestamp ($\mathcal{P} : \mathcal{S}_{\mathbb{D}}^{\mathbb{T}} \times \mathbb{T} \to \mathbb{D}$), would be thinkable. In this thesis however, we concentrate on *synchronous properties* and monitoring and leave continuous properties for future work.

Note here that synchronous stream runtime verification languages (e.g. LOLA) also describe pointwise properties. This connection will be considered separately in detail at the end of the chapter.

Also, initial properties can generally be considered a special case of pointwise properties, which are ignorant regarding the position parameter. I.e. if $\mathcal{P}$ is an initial property, it can be transformed to the pointed property $\mathcal{P}'(w, i) = \mathcal{P}(w)$.

Besides the LTL_3 monitoring algorithm, which served us as a role-model for the concept of initial monitoring in the previous section, section 2.2.2 also discussed the approach from Havelund and Roşu [HR02] for monitoring past LTL w.r.t. its pointed semantics (definition 2.15). In their approach they generate a monitor which receives an input word incrementally and casts a verdict at every input. It reflects the pointed LTL semantics of a given specification at the point up to which the input has been received (in the following called *current instant*). We call such a monitoring approach *recurrent*, because the same property is evaluated recurrently (i.e. over and over again) by the monitor, each time a new input is received. Unlike in initial monitoring, where the evaluation of the property is rather continued with an additional piece of input.

We can generalize the mentioned strategy for past LTL monitoring to arbitrary pointwise properties, by giving definitions for sound and perfect recurrent monitoring, based on definition 3.3. Thereby we again follow the idea from LTL_3 to consider all potential input continuations and give outputs which yield the possible valuations or over-approximate them in case of sound monitoring. We consequently unify the approaches from the two algorithms.

Definition 3.5 (Recurrent monitoring; based on [KLSS22, HKLS24])**.**

Let $\mathcal{P} : \Omega \times \mathbb{N} \rightharpoonup \mathbb{D}$ with $\Omega \in \{\Sigma^*, \Sigma^n, \Sigma^\omega, \Sigma^\infty\}$ be a pointwise property. Let further $\mathcal{M} = (S, \Sigma, \mathbb{V}, \delta, \gamma, s_0)$ be a synchronous monitor with output interpretation $\gamma_{\mathbb{V}} : \mathbb{V} \to 2^{\mathbb{D}}$.

$\mathcal{M}$ is called

- *sound recurrent monitor* for $\mathcal{P}$ if and only if

$$\forall w \in \Sigma^*. \gamma_{\mathbb{V}}(\mathcal{M}(w)) \supseteq \{\mathcal{P}(wv, |w| - 1) \mid wv \in \Omega \wedge \mathcal{P}(wv, |w| - 1) \text{ def.}\}$$

- *perfect recurrent monitor* for $\mathcal{P}$ if and only if

$$\forall w \in \Sigma^*. \gamma_{\mathbb{V}}(\mathcal{M}(w)) = \{\mathcal{P}(wv, |w| - 1) \mid wv \in \Omega \wedge \mathcal{P}(wv, |w| - 1) \text{ def.}\}$$

Hence a sound recurrent monitor casts an over-approximation of all possible valuations for the current position in the input word. A perfect one provides an exact representation of all such valuations.

The monitor construction from [HR02] yields a perfect recurrent monitor for past LTL. Since past LTL's pointed semantics never depends on input letters at later instants than the considered one, this monitor only casts the final verdicts $\top \mathrel{\widehat{=}} \{\texttt{true}\}$ and $\bot \mathrel{\widehat{=}} \{\texttt{false}\}$. These verdicts give full information about satisfaction or violation of the property at this position. However, for future or full LTL the situation is different. There even a perfect recurrent monitor might oftentimes only be able to cast $? \mathrel{\widehat{=}} \{\texttt{true}, \texttt{false}\}$ verdicts, as the pointwise semantics at some instant is usually dependent on future letters. Consider for example the LTL formula $(\mathcal{P}q) \vee (\mathcal{X}\mathcal{X}p)$ and the input/output traces of a perfect recurrent monitor in figure 3.2.

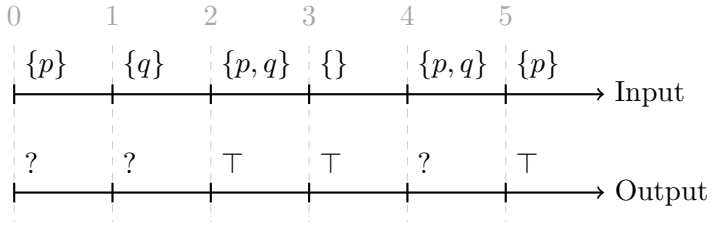


Figure 3.2.: Perfect recurrent monitor for $(\mathcal{P}q) \vee (\mathcal{X}\mathcal{X}p)$.

While at some instants (2,3,5) it is possible to give a final verdict, because the truth-value of the formula is fully determined by the already received letters, this is not the case at the other positions. There, the satisfaction of the property depends on the $\mathcal{X}\mathcal{X}p$ part and thus on inputs not yet received. The perfect monitor output is consequently the uninformative verdict ?. However, if the verdicts for this particular specification were to be cast for two positions prior to the current instant instead for

the current one, a perfect monitor would provide conclusive verdicts for all positions. This is because the *look-ahead*, i.e. the number of future positions the formula $(\mathcal{P}q) \vee (\mathcal{X}\mathcal{X}p)$ depends on, is two, conditioned by the two nested next operators. This observation motivates to adjust the definition of recurrent monitoring to support a constant offset between the instant until which the input is known and the instant for which the monitor output is given. As above we do not restrict this notion of k-offset recurrent monitoring to LTL or boolean verdicts, but define it for arbitrary pointwise properties.

Definition 3.6 (k-offset recurrent monitoring; based on [KLSS22])**.**

Let $\mathcal{P} : \Omega \times \mathbb{N} \rightharpoonup \mathbb{D}$ with $\Omega \in \{\Sigma^*, \Sigma^n, \Sigma^\omega, \Sigma^\infty\}$ be a pointwise property. Let further $\mathcal{M} = (S, \Sigma, \mathbb{V}, \delta, \gamma, s_0)$ be a synchronous monitor with output interpretation $\gamma_{\mathbb{V}} : \mathbb{V} \to 2^{\mathbb{D}}$.

For a fixed $k \in \mathbb{Z}$, $\mathcal{M}$ is called

- *sound k-offset recurrent monitor* for $\mathcal{P}$ if and only if

  $$\forall w \in \Sigma^*. \gamma_{\mathbb{V}}(\mathcal{M}(w)) \supseteq \{\mathcal{P}(wv, |w|{-}1{-}k) \mid wv \in \Omega \wedge \mathcal{P}(wv, |w|{-}1{-}k) \text{ def.}\}$$

- *perfect k-offset recurrent monitor* for $\mathcal{P}$ if and only if

  $$\forall w \in \Sigma^*. \gamma_{\mathbb{V}}(\mathcal{M}(w)) = \{\mathcal{P}(wv, |w|{-}1{-}k) \mid wv \in \Omega \wedge \mathcal{P}(wv, |w|{-}1{-}k) \text{ def.}\}$$

Note, that a fundamental concept behind k-offset recurrent monitoring is a separation between *monitored time*, i.e. the position to which the monitor output belongs, and *monitoring time*, i.e. the position until which the monitor has received the input.

In k-offset recurrent monitoring these two times are related by a constant offset. While this is a satisfying solution for the case of monitoring $(\mathcal{P}q) \vee (\mathcal{X}\mathcal{X}p)$ as discussed above and depicted in figure 3.2, it might not be the optimal choice in other specific scenarios, e.g. for 2-offset monitoring of $q \vee (p \,\mathcal{U}\, q)$. A depiction of this case can be found in figure 3.3.

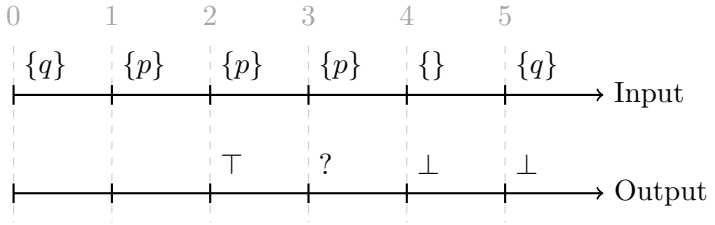


Figure 3.3.: Perfect 2-offset recurrent monitor for $q \vee (p \,\mathcal{U}\, q)$. The output verdicts refer to the pointwise semantics of the formula two instants before.

After receiving four input letters the monitor there yields the verdict ?, referring to position 1. In fact the monitor would have required another input letter to cast a final verdict ($\bot$), i.e. a 3-offset recurrent monitor would have been required in this case. Yet, for any chosen k, there could have been a trace with $k+1$ $\{p\}$ input letters in a row, which would have lead to an inconclusive ? verdict.

On the other hand the drawback of choosing a (too) high value for k can also be seen in the example: The $\top$ verdict for instant 0 is cast when the third input letter is read, however this verdict has already been clear, when the first letter was received. In real world scenarios this delayed availability of final verdicts, though they are already inevitable, is unfavorable.

In conclusion this leads to the insight that a constant offset between monitoring and monitored time is often not advantageous. Thus we define a further variant of recurrent monitoring which allows for a full separation of them. We call this kind of monitoring *random access (recurrent) monitoring*. In advance of the definition we first introduce the notion of a *query domain*, that is a function $\mathbb{N} \to 2^{\mathbb{N}}$ defining the positions for which the monitor has to provide property valuations with respect to the currently received input position. The query domain $\mathcal{Q}(t) = \{t-1, t, t+1\} \cap \mathbb{N}$ would e.g. require the monitor to provide information about the current, the previous and the next position. It thus represents a sliding window of size three around the current instant. We call a query domain $\mathcal{Q}$ with $\mathcal{Q}(t) = \emptyset$ *trivial for instant* t.

As output domain of a random access monitor we consequently do not choose $\mathbb{V}$ as before, which encodes different property valuations, but the set of all partial functions from $\mathbb{N}$ to $\mathbb{V}$, which depict instants from the query domain to the corresponding output there. The monitor can thus be considered as kind of question answering machine: It receives the input word letter by letter and after each letter returns a function object which provides a sound (or perfect) representation of the possible property valuations at the instants in the query domain.

Definition 3.7 (Random access (recurrent) monitoring; based on [KLSS22])**.**

Let $\mathcal{P} : \Omega \times \mathbb{N} \rightharpoonup \mathbb{D}$ be a pointwise property with $\Omega \in \{\Sigma^*, \Sigma^n, \Sigma^\omega, \Sigma^\infty\}$, $\mathcal{Q} : \mathbb{N} \to 2^{\mathbb{N}}$ a query domain and $\mathcal{M} = (S, \Sigma, \mathbb{N} \rightharpoonup \mathbb{V}, \delta, \gamma, s_0)$ a synchronous monitor with output interpretation $\gamma_{\mathbb{V}} : \mathbb{V} \to 2^{\mathbb{D}}$.

$\mathcal{M}$ is called

- *sound random access (recurrent) monitor* for $\mathcal{P}, \mathcal{Q}$ if and only if

$$\forall w \in \Sigma^+. \forall i \in \mathcal{Q}(|w|-1). \gamma_{\mathbb{V}}(\mathcal{M}(w)(i)) \supseteq \{\mathcal{P}(wv, i) \mid wv \in \Omega \wedge \mathcal{P}(wv, i) \text{ def.}\}$$

- *perfect random access (recurrent) monitor* for $\mathcal{P}, \mathcal{Q}$ if and only if

$$\forall w \in \Sigma^+. \forall i \in \mathcal{Q}(|w|-1). \gamma_{\mathbb{V}}(\mathcal{M}(w)(i)) = \{\mathcal{P}(wv, i) \mid wv \in \Omega \wedge \mathcal{P}(wv, i) \text{ def.}\}$$

The principle of random access recurrent monitoring is illustrated in figure 3.4 which depicts a perfect random access monitor for $\mathcal{Q}(t) = \mathbb{N}$, again for the LTL formula $q \vee (p \,\mathcal{U}\, q)$ and the trace from figure 3.3.

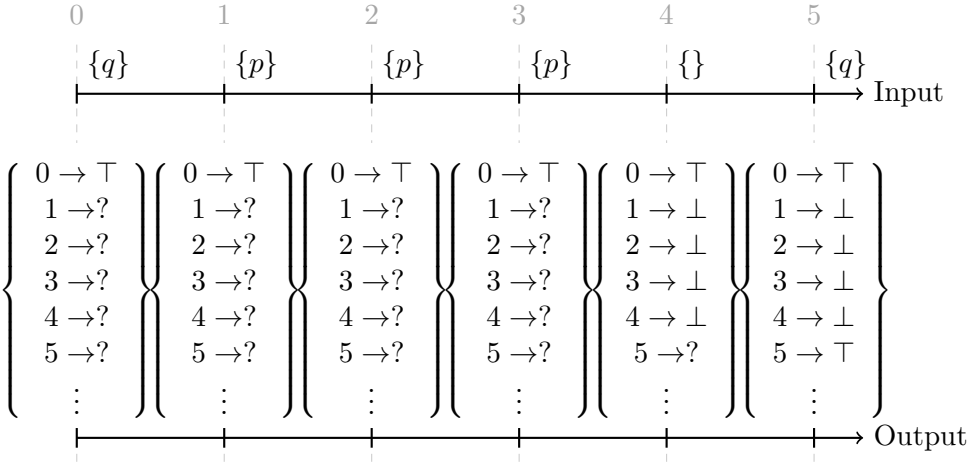


Figure 3.4.: Perfect random access monitor for $q \vee (p \,\mathcal{U}\, q)$.

It is easy to see that a random access monitor subsumes all other monitoring models from this chapter. E.g. k-offset recurrent monitoring is equal to random access monitoring with query domain $\mathcal{Q}(t) = \{t - k\} \cap \mathbb{N}$. Initial monitoring on the other hand equals random access monitoring with query domain $\mathcal{Q}(t) = \{0\}$.

Thus, a perfect random access monitor is a powerful tool for online monitoring, but – depending on the chosen query domain – not suitable for practical use. For $\mathcal{Q}(t) = \mathbb{N}$ for example an efficient implementation of such a monitor is in general not possible. It would have to store information about every single letter in the received word, to give valuations for all positions. This is even the case for very simple properties like the LTL formula p. To yield the output function assigning a verdict to every instant, the monitor would have to record all instants of the received word which contained the proposition p. In general this makes such a monitor trace-length-dependent, which is an issue for the usage in runtime verification.

One could argue that the requirements for an implementation can be relaxed such that the monitor output only has to contain values for positions that were not conclusive (i.e. mapping to a singleton valuation set) in a previous output. This is because these values are fully known and don't change anymore until the end of the monitoring. They can thus can be reported uniquely at their first occurrence. In general, however, the monitor would still have to store an increasing amount of information, even for simple properties. For example, consider the LTL formula $p \rightarrow \mathcal{F}q$. In this case the monitor would have to memorize all positions in which p held until a subsequent q arrives, because only then the verdict for these locations

would flip from ? to $\top$. Overall, these examples show that perfect random access monitoring is generally incompatible with trace-length-independent monitoring.

Possibilities to counter this issue are either to restrict the query domain in some way or to deliver abstractions of the actual outputs (the functions). An example for such an abstraction is e.g. discussed in [KLSS22], called – slightly contrary to the wording used in this thesis – anticipatory recurrent monitoring. There the monitor only casts an interval which indicates the distance to the next instant where a monitored LTL property may be satisfied.

3.3.3. Extensions

In order to provide a general theory for synchronous monitoring, we will discuss in this section how to enrich the notion of random access monitoring with the presence of uncertain input readings and assumptions.

Assumptions

As outlined in section 2.2.4 assumptions restrict the set of possible input traces that can actually occur in the observed system and which the monitor has to consider. This does not only restrict the prefixes which have been received by the monitor but also the set of possible continuations, i.e. the "search space" of the monitor. We can quite easily extend our notion of sound and perfect random access monitoring to support assumptions.

Definition 3.8 (Random access (recurrent) monitoring under assumptions; based on [HKLS24])**.**

Let $\mathcal{P} : \Omega \times \mathbb{N} \rightharpoonup \mathbb{D}$ be a pointwise property with $\Omega \in \{\Sigma^*, \Sigma^n, \Sigma^\omega, \Sigma^\infty\}$ and $A \subseteq \Omega$ an assumption. Let further $\mathcal{Q} : \mathbb{N} \to 2^{\mathbb{N}}$ be a query domain and $\mathcal{M} = (S, \Sigma, \mathbb{N} \rightharpoonup \mathbb{V}, \delta, \gamma, s_0)$ a synchronous monitor with output interpretation $\gamma_{\mathbb{V}} : \mathbb{V} \to 2^{\mathbb{D}}$.

$\mathcal{M}$ is called

- *sound random access (recurrent) monitor* for $\mathcal{P}, \mathcal{Q}$ and A if and only if

$$\forall w \in \Sigma^+. \forall i \in \mathcal{Q}(|w|{-}1). \gamma_{\mathbb{V}}(\mathcal{M}(w)(i)) \supseteq \{\mathcal{P}(wv, i) \mid wv \in A \wedge \mathcal{P}(wv, i) \text{ def.}\}$$

- *perfect random access (recurrent) monitor* for $\mathcal{P}, \mathcal{Q}$ and A if and only if

$$\forall w \in \Sigma^+. \forall i \in \mathcal{Q}(|w|{-}1). \gamma_{\mathbb{V}}(\mathcal{M}(w)(i)) = \{\mathcal{P}(wv, i) \mid wv \in A \wedge \mathcal{P}(wv, i) \text{ def.}\}$$

Compared to the previous definition of random access monitoring, a perfect monitor now only has to care about extensions of the received input that also match assumption A. Likewise the output of a sound monitor only has to over-approximate property valuations of runs that fit the assumption.

In case the input trace has already breached the assumption, there would be no extension of w which is contained in A and thus a perfect random access monitor would have to yield a representation of the empty value set. As discussed earlier, we will not consider this case further in this thesis, because the monitor should never receive an input which violates the assumption.

Uncertainty

In a similar straight-forward fashion we can extend our concept with the capability of handling uncertainty. The problem of monitoring languages in $\Sigma^{*/n/\omega/\infty}$ under uncertainty has already been discussed in section 2.2.4. There we had introduced a notion of sound and perfect uncertain monitoring of language containment, which we will now extend to the case of monitoring general pointwise properties.

We use the idea from there, that an uncertain input over Σ is essentially a subset $U \subseteq \Sigma^*$ containing all potential concrete inputs. Our monitors, as defined in definition 3.1, receive their input piece-wise, s.t. every received input letter reveals additional information about the monitored run. To transfer this to the uncertain setting, we require an abstract alphabet Γ and uncertainty encoding $\nu : \Gamma^* \to 2^{\Sigma^*}$ (see definition 2.30) which provides the mapping between the monitor input and the corresponding uncertain input word. As suggested, this could e.g. be a letter- or chunk-wise encoding of the uncertainty (see definition 2.32).

Based on this considerations we can extend the definition of perfect random access monitors by assumptions and uncertainty.

Definition 3.9 (Random access monitoring under uncertainty and assumptions)**.**

Let $\mathcal{P} : \Omega \times \mathbb{N} \rightharpoonup \mathbb{D}$ with $\Omega \in \{\Sigma^*, \Sigma^n, \Sigma^\omega, \Sigma^\infty\}$ be a pointwise property and $A \subseteq \Omega$ an assumption and $\mathcal{Q} : \mathbb{N} \to 2^{\mathbb{N}}$ a query domain. Let further $\mathcal{M} = (S, \Gamma, \mathbb{N}^+ \to \mathbb{V}, \delta, \gamma, s_0)$ be a synchronous monitor with output interpretation $\gamma_{\mathbb{V}} : \mathbb{V} \to 2^{\mathbb{D}}$ and uncertainty encoding $\nu : \Gamma^* \to 2^{\Sigma^*}$.

$\mathcal{M}$ is called

- *sound uncertain random access (recurrent) monitor* for $\mathcal{P}, \mathcal{Q}$ under A if and only if

$$\forall w \in \Gamma^+. \forall i \in \mathcal{Q}(|w| - 1). \gamma_{\mathbb{V}}(\mathcal{M}(w)(i)) \supseteq \{\mathcal{P}(w'v, i) \mid w' \in \nu(w) \wedge w'v \in A \wedge \mathcal{P}(w'v, i) \text{ def.}\}$$

- *perfect uncertain random access (recurrent) monitor* for $\mathcal{P}, \mathcal{Q}$ under A if and only if

$$\forall w \in \Gamma^+. \forall i \in \mathcal{Q}(|w|-1). \gamma_{\mathbb{V}}(\mathcal{M}(w)(i)) = \{\mathcal{P}(w'v, i) \mid w' \in \nu(w) \wedge w'v \in A \wedge \mathcal{P}(w'v, i) \text{ def.}\}$$

Note that with the presence of uncertainty, $w \in \Gamma^*$ can generally represent a set of possible words of different lengths. This leads to a peculiarity of definition 3.9 compared to the previous definitions. First the query domain now receives the length of the encoded uncertain input but delivers positions in the decoded word for which the monitor has to answer. In case of a non-linear relation between the length of the encoded uncertain input and the represented certain inputs this makes e.g. the encoding of sliding windows around the current position via the query domain impossible. Likewise k-offset recurrent monitoring can no longer be considered as special case of random access monitoring by "asking" the monitor for $\mathcal{M}(w)(|w|-1-k)$ as there is no unique current input length anymore. The corresponding definition of a k-offset recurrent monitoring under uncertainty and assumptions would therefore be as given in definition 3.10.

Definition 3.10 (k-offset rec. monitoring under uncertainty and assumptions)**.**

Let $\mathcal{P} : \Omega \times \mathbb{N} \rightharpoonup \mathbb{D}$ be a pointwise property with $\Omega \in \{\Sigma^*, \Sigma^n, \Sigma^\omega, \Sigma^\infty\}$ and $A \subseteq \Omega$ an assumption. Let further $\mathcal{M} = (S, \Gamma, \mathbb{V}, \delta, \gamma, s_0)$ be a synchronous monitor with output interpretation $\gamma_{\mathbb{V}} : \mathbb{V} \to 2^{\mathbb{D}}$ and uncertainty encoding $\nu : \Gamma^* \to 2^{\Sigma^*}$.

$\mathcal{M}$ is called

- *sound uncertain k-offset recurrent monitor* for $\mathcal{P}, \mathcal{Q}$ and A if and only if

$$\forall w \in \Gamma^*. \gamma_{\mathbb{V}}(\mathcal{M}(w)) \supseteq \{\mathcal{P}(w'v, |w'|-1-k) \mid w' \in \nu(w) \wedge w'v \in A \wedge \mathcal{P}(w'v, |w'|-1-k) \text{ def.}\}$$

- *perfect uncertain k-offset recurrent monitor* for $\mathcal{P}, \mathcal{Q}$ and A if and only if

$$\forall w \in \Gamma^*. \gamma_{\mathbb{V}}(\mathcal{M}(w)) = \{\mathcal{P}(w'v, |w'|-1-k) \mid w' \in \nu(w) \wedge w'v \in A \wedge \mathcal{P}(w'v, |w'|-1-k) \text{ def.}\}$$

Consequently, if the uncertain input encodes traces of different lengths, a sound or perfect k-offset recurrent monitor reveals information about the instant k positions away from the end of the different represented certain input traces. Thus in its output the monitor combines the valuations for different absolute positions.

For monitoring under uncertainty (with the possibility of unevenly long concrete words), neither a random access nor a k-offset monitor is strong enough to entail

the outputs of the other, and thus it is up to the user to decide which monitoring model is appropriate for the concrete use case. However, in the subsequent chapter we will restrict the handled uncertainty to the case where only traces of equal length are encoded by the uncertain input. In particular we will only consider letter-wise uncertainty, where the encoded input has the same length as the represented inputs. In this case the random access monitor is again powerful enough to perform k-offset recurrent monitoring, even for a sliding window.

3.4. Connection to stream runtime verification

The definitions given so far provide a general concept of sound and perfect monitoring of arbitrary synchronous properties. We will now discuss stream runtime verification approaches in terms of this framework and investigate which properties it can express. In doing so, we will pay special attention to the language LOLA and in particular show how other formalisms can be unified by embedding them in LOLA. In the following chapter we will then present a generic implementation of a sound and perfect recurrent monitor for LOLA, using abstract interpretation techniques, which in turn will be able to generalize recurrent RV under uncertainty and assumptions for several existing formalisms.

3.4.1. LOLA specifications as pointwise properties

As outlined in section 2.3.1, a well-defined LOLA specification describes a transformation from a fixed number of input streams to a fixed number of output streams over arbitrary data domains. This conceptually fits to our notion of pointwise properties (definition 3.4) which receive a sequence of arbitrary data type and assign a property of arbitrary datatype to every point of the sequence. The connection becomes clear, if we consider the input streams of the LOLA specification as a single stream/sequence of tuples consisting of all stream values at this instant. Likewise we can interpret the output streams as a single sequence of stream value tuples. Remember at this point that a finite synchronous stream (definition 2.3) is equivalent to a finite sequence of data values (i.e. a trace).

Technically we describe the above translation with the functions

$$\begin{aligned} com &: \mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_n} \to (\mathbb{D}_1 \times \cdots \times \mathbb{D}_n)^{t_{max}+1} \\ com((s_1, \ldots, s_n)) &= \langle (s_1(0), \ldots, s_n(0)), \ldots, (s_1(t_{max}), \ldots, s_n(t_{max})) \rangle \end{aligned}$$

composing a tuple of streams into a single sequence of value tuples and

$$\begin{aligned} dec &: (\mathbb{D}_1 \times \cdots \times \mathbb{D}_n)^{t_{max}+1} \to \mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_n} \\ dec(w) &= (\langle \pi_1(w(0)), \ldots, \pi_1(w(t_{max})) \rangle, \ldots, \langle \pi_n(w(0)), \ldots, \pi_n(w(t_{max})) \rangle) \end{aligned}$$

decomposing a sequence of value tuples into a tuple of streams. Thereby $\pi_i(t)$ denotes the i^{th} component of tuple t.

Based on this translation, we can explicitly define the induced (pointwise) property of a LOLA specification.

Definition 3.11 (Induced property of LOLA specification; based on [HKLS24])**.**
Let φ be a LOLA specification with input stream types $\mathbb{D}_1, \ldots, \mathbb{D}_n$ and output stream types $\mathbb{D}'_1, \ldots, \mathbb{D}'_m$.

The function $\mathcal{P}_\varphi : (\mathbb{D}_1 \times \cdots \times \mathbb{D}_n)^{t_{max}+1} \times \mathbb{N} \rightharpoonup \mathbb{D}'_1 \times \cdots \times \mathbb{D}'_n$ with

$$\mathcal{P}_\varphi(w,t) = \begin{cases} com([\![\varphi]\!](dec(w)))(t) & \text{if } t \in \mathbb{T} \\ undef. & \text{else} \end{cases}$$

where $[\![\varphi]\!] : (\mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_n}) \to (\mathcal{S}_{\mathbb{D}'_1} \times \cdots \times \mathcal{S}_{\mathbb{D}'_m})$ is the semantics of φ is called the *induced (pointwise) property of* φ.

3.4.2. Embedding of pointwise properties in LOLA

Because of its generic nature and lack of restriction to specific data domains, LOLA is a very powerful formalism for the description of stream transformations. In fact every fixed-sized pointwise property (i.e. over $\Omega = \Sigma^{t_{max}+1}$) as defined in definition 3.4 is expressible in LOLA.

Thereby the restriction to fixed trace lengths is not a grave one for practical applications. While for offline monitoring the trace length of the input is known a priori anyway, for online monitoring one usually just wants to have a trace length which is long enough such that the trace end is not reached during monitoring. Consequently t_{max} can simply be chosen as a large number, e.g. 2^{64} which would be enough to monitor for over 500 million years at an event rate of one input instant per millisecond.

The proposition about encoding pointwise properties in LOLA specifications is formalized in theorem 3.12.

Theorem 3.12.
For every finite pointwise property $\mathcal{P} : \Sigma^{t_{max}+1} \times \mathbb{N} \rightharpoonup \mathbb{D}$ there is a LOLA specification φ, s.t.

$$\forall w \in \Sigma^{t_{max}+1}, t \in \mathbb{T} = \{0, 1, \ldots, t_{max}\}.\, \mathcal{P}_\varphi(w,i) = \mathcal{P}(w,i).$$

Proof. Let $\mathcal{P} : \Sigma^{t_{max}+1} \times \mathbb{N} \rightharpoonup \mathbb{D}$ be a finite pointwise property. Consider the following LOLA specification:

```
1  input in: Σ
2
3  n := n[-1|0] + 1
4  past := past[-1|⟨⟩]∘⟨in[now]⟩
5  future := ⟨in[now]⟩∘future[+1|⟨⟩]
6  out := 𝒫(past[-1|⟨⟩]∘future[now], n[now]-1)
7
8  output out
```

The specification contains three intermediate streams: n, which is counting the length of the prefix up to the current position. The streams *past* and *future* which are of type Σ^* and maintain the received input up to/from the current position on. They do this by converting the current input letter from stream *in* of type Σ into a 1-element sequence (function $\langle\cdot\rangle$) and concatenating it (function $\cdot \circ \cdot$) with the last and next value of themselves. Thus, *past* contains the prefix of the stream from the beginning up to the current position and *future* the postfix from this position to the trace end. Hence, for every timestamp, $\mathit{past}[-1|\langle\rangle] \circ \mathit{future}[\mathit{now}]$ evaluates to the full input trace. Consequently for arbitrary input trace $w \in \Sigma^{t_{max}+1}$ the single output stream *out* at position i ($0 \leq i \leq t_{\max}$) is by definition equivalent to $\mathcal{P}(w, i)$. □

The "trick" of this construction basically consists in collecting all past input values and all future input values and aggregating them in a sequence data structure. This construction thus requires storage of all past and future inputs in a complex data structure (list or sequence) to compute the output for a single instant. Yet not all properties do actually need this unlimited amount of information, rather for some it is enough to just reference a fixed number of past and future instances, or to aggregate the past and future information in a compact value. This is e.g. the case for properties expressed in LTL, M(I)TL and STL (with a discrete time domain). In the following the (generic) translation from these logics to LOLA will be shown. We start with LTL.

Let $\varphi \in \text{LTL}_{\mathcal{AP}}$ be an LTL formula over atomic propositions $\mathcal{AP}$. The main idea (which is also used for the M(I)TL and STL translation) is that each subformula of φ is represented by its own stream of type boolean (c.f. [HKLS24, Gor22, DSS+05]). The stream for such an expression is in turn defined by a composition of the streams of its sub-expressions. Finally we assume a single input stream `in` with sets of atomic propositions from $\mathcal{AP}$. The concrete translation of the different expression types is given by the following table:

$\varphi =$	LOLA stream definition of s_φ
`true`	$s_{\texttt{true}}$ `:= true`
p	s_p `:=` $(p \in$ `in[now])`
$\neg\psi$	$s_{\neg\psi}$ `:=` $\neg s_\psi$`[now]`
$\psi_1 \wedge \psi_2$	$s_{\psi_1\wedge\psi_2}$ `:=` s_{ψ_1}`[now]` $\wedge$ s_{ψ_2}`[now]`
$\mathcal{P}\psi$	$s_{\mathcal{P}\psi}$ `:=` s_ψ`[-1|false]`
$\psi_1 \mathcal{S} \psi_2$	$s_{\psi_1\mathcal{S}\psi_2}$ `:=` s_{ψ_2}`[now]` $\vee$ `(`s_{ψ_1}`[now]` $\wedge$ $s_{\psi_1\mathcal{S}\psi_2}$`[-1|false])`
$\mathcal{X}\psi$	$s_{\mathcal{X}\psi}$ `:=` s_ψ`[+1|false]`
$\psi_1 \mathcal{U} \psi_2$	$s_{\psi_1\mathcal{U}\psi_2}$ `:=` s_{ψ_2}`[now]` $\vee$ `(`s_{ψ_1}`[now]` $\wedge$ $s_{\psi_1\mathcal{U}\psi_2}$`[+1|false])`

The translation is mostly straight-forward, $s_{\texttt{true}}$ is the stream with `true` events at all positions, and s_p for $p \in \mathcal{AP}$ is true whenever p is contained in the current input value. The propositional logic operators $\neg$, $\wedge$ are applied to the current value(s) of the sub-expression(s). $\mathcal{P}$ ($\mathcal{X}$) takes the previous (next) value from the stream of its sub-expression and is false at the first (last) position. $\mathcal{S}$ and $\mathcal{U}$ are unrolled according to the well-known equalities

$$\varphi \,\mathcal{U}\, \psi \equiv \psi \vee (\varphi \wedge \mathcal{X}(\varphi \,\mathcal{U}\, \psi)) \text{ and } \varphi \,\mathcal{S}\, \psi \equiv \psi \vee (\varphi \wedge \mathcal{P}(\varphi \,\mathcal{S}\, \psi)).$$

Note, that for practical applications it is often more convenient to consider s_a for each $a \in \mathcal{AP}$ as separate input streams of type boolean instead of one stream *in* carrying sets of atomic propositions. In this case the input word would consist of boolean vectors indicating which atomic propositions currently hold. In the following we will make use of this equivalent version.

For $\varphi = \mathcal{F}((\mathcal{P}p)\,\mathcal{U}\,q) = (\texttt{true}\,\mathcal{U}\,((\mathcal{P}p)\,\mathcal{U}\,q))$ we would for example get the following specification:

```
1  input s_p: 𝔹
2  input s_q: 𝔹
3
4  s_true := true
5  s_Pp := s_p[-1|false]
6  s_(Pp)Uq := s_q[now] ∨ (s_Pp[now] ∧ s_(Pp)Uq[+1|false])
7  s_φ := s_(Pp)Uq[now] ∨ (s_true[now] ∧ s_φ[+1|false])
8
9  output s_φ
```

Similarly other formalisms like LTL with regular expressions [LS07] or linear dynamic logic (LDL) [GV13] can be translated to LOLA following a similar scheme. Corresponding constructions are outlined in [KLS23].

For translation of M(I)TL we can also use the strategy of defining a stream for every sub-expression of the formula to be monitored. For the timed streams on which M(I)TL is defined, we use the encoding as a timed trace, i.e. a trace of timestamp/value tuples (see section 2.1.3). For the LOLA specification we thus assume - besides a stream for every atomic proposition - also an input stream t,

which carries the current timestamp and is of type $\mathbb{T}$. As mentioned earlier, for the M(I)TL-to-LOLA translation we restrict ourselves to discrete time domains $\mathbb{T}$. Such time domains are characterized by the fact that there is a smallest possible distance d_{min} between two timestamps. Yet, it is not required by definition 2.4 that all subsequent timestamps have exactly this minimal distance, i.e. the time domain may be *irregular*.

If we have the common case of a regular time domain like e.g. $\mathbb{T} = \mathbb{N}^{t_{max}}$ (i.e. all natural numbers from 0 to t_{max}), where for every $t \in \mathbb{T}$, either $t + d_{min} \in \mathbb{T}$ or $t = t_{max}$, we can choose a strategy similar to the LTL translation above (also comparable to the one from [Gor22]). In this case we define the LOLA specification s.t. at instant $k \in \mathbb{N}$ it evaluates the M(I)TL formula for timestamp $k \cdot d_{min} \in \mathbb{T}$.

The operators $\neg$, $\wedge$ and the constant `true` can be handled exactly in the same way as in the LTL translation before. The operators until and since in M(I)TL however carry an interval which restricts the time period in which the second argument has to hold and thus require a special translation. First note, that for a discrete domain $\mathbb{T}$, we can restrict to inclusive intervals (except for ∞), as non-inclusive bounds can simply be replaced by inclusive bounds of the next and previous instant. We slightly abuse notation and write $[a, b]$ with $a \in \mathbb{R}^{\geq 0}$ and $b \in \mathbb{R}^{\geq 0} \cup \{\infty\}$ to denote the interval between a and b, where b is exclusive if and only if it is ∞. In the following we will consider the case for the since operator; the until operator is analogous.

Recall that expression $\psi \, \mathcal{S}_I \, \varphi$ is satisfied whenever φ held some time ago which is within the interval I from now and ever since ψ was satisfied at all points in time. This means, it is necessary to memorize if φ held at a certain interval in the past to determine the current value of $\psi \, \mathcal{S}_I \, \varphi$. Since we know that no events can occur outside of the time domain, the following rules for $a \in \mathbb{R}^{>0}$ and $b \in \mathbb{R}^{>0} \cup \{\infty\}$ hold (where $\varphi[-d_{min}]$ denotes the value of φ at the previous timestamp):

$$\begin{aligned}
\varphi \, \mathcal{S}_{[0,0]} \, \psi &\equiv \psi \\
\varphi \, \mathcal{S}_{[0,b]} \, \psi &\equiv \psi \vee (\varphi \wedge (\varphi \, \mathcal{S}_{[0,b-d_{min}]} \, \psi)[-d_{min}]) \\
\varphi \, \mathcal{S}_{[a,b]} \, \psi &\equiv \varphi \wedge (\varphi \, \mathcal{S}_{[a-d_{min},b-d_{min}]} \, \psi)[-d_{min}].
\end{aligned}$$

Thus we can encode $\psi \, \mathcal{S}_I \, \varphi$ and likewise $\psi \, \mathcal{U}_I \, \varphi$ in LOLA by "unrolling" the intervals into immediate streams:

$\varphi =$	LOLA stream definition of s_φ
$\varphi \, \mathcal{S}_{[0,0]} \, \psi$	$s_{\varphi \mathcal{S}_{[0,0]} \psi}$ `:=` s_ψ`[now]`
$\varphi \, \mathcal{S}_{[0,b]} \, \psi$	$s_{\varphi \mathcal{S}_{[0,b]} \psi}$ `:=` s_ψ`[now]` $\vee$ (s_φ`[now]` $\wedge$ $s_{\varphi \mathcal{S}_{[0,b-d_{min}]} \psi}$`[-1|false]`)
$\varphi \, \mathcal{S}_{[a,b]} \, \psi$	$s_{\varphi \mathcal{S}_{[a,b]} \psi}$ `:=` s_φ`[now]` $\wedge$ $s_{\varphi \mathcal{S}_{[a-d_{min},b-d_{min}]} \psi}$`[-1|false]`
$\varphi \, \mathcal{U}_{[0,0]} \, \psi$	$s_{\varphi \mathcal{U}_{[0,0]} \psi}$ `:=` s_ψ`[now]`
$\varphi \, \mathcal{U}_{[0,b]} \, \psi$	$s_{\varphi \mathcal{U}_{[0,b]} \psi}$ `:=` s_ψ`[now]` $\vee$ (s_φ`[now]` $\wedge$ $s_{\varphi \mathcal{U}_{[0,b-d_{min}]} \psi}$`[+1|false]`)
$\varphi \, \mathcal{U}_{[a,b]} \, \psi$	$s_{\varphi \mathcal{U}_{[a,b]} \psi}$ `:=` s_φ`[now]` $\wedge$ $s_{\varphi \mathcal{U}_{[a-d_{min},b-d_{min}]} \psi}$`[+1|false]`

For illustration, the M(I)TL formula $\varphi = \mathcal{F}(p\,\mathcal{U}_{[2,\infty]}\,q) = (\texttt{true}\,\mathcal{U}_{[0,\infty]}\,(p\,\mathcal{U}_{[2,\infty]}\,q))$ with time domain $\mathbb{T} = \mathbb{N}^{t_{max}} = \{0, 1, \ldots, t_{max}\}$ and thus $d_{min} = 1$, would yield the following specification:

```
1  input t: ℕ^{t_max}
2  input s_p: 𝔹
3  input s_q: 𝔹
4
5  s_true := true
6  s_{pU_[0,∞]q} := s_q[now] ∨ (s_p[now] ∧ s_{pU_[0,∞]q}[+1|false])
7  s_{pU_[1,∞]q} := s_p[now] ∧ s_{pU_[0,∞]q}[+1|false]
8  s_{pU_[2,∞]q} := s_p[now] ∧ s_{pU_[1,∞]q}[+1|false]
9  s_φ := s_{pU_[2,∞]q}[now] ∨ (s_true[now] ∧ s_φ[+1|false])
10
11 output s_φ
```

Unfortunately this translation is not suited for the case of an irregular time domain where subsequent timestamps have no equal distance. It also introduces a high number of intermediate streams, especially when until and since operators use large interval bounds. This has in general negative influence on the evaluation time. The reason for the large number of streams, is, that for $\varphi\,\mathcal{S}_{[a,b]}\,\psi$ or $\varphi\,\mathcal{U}_{[a,b]}\,\psi$ with $b \neq \infty$, $\frac{b}{d_{min}}$ variations of the formula with adjusted interval are iteratively evaluated and stored in a dedicated stream. However information about all these individual points is actually not necessary.

Indeed, the following observation can be made (for any time domain even dense ones): Consider the M(I)TL formula $\varphi = \psi_1\,\mathcal{S}_{[a,b]}\,\psi_2$ and a timed stream $s \in \mathcal{S}_\Sigma^{\mathbb{T}}$ over arbitrary time domain with $\mathbb{R}^{\geq 0}$ distance measure, where ψ_2 holds at timestamp t_1 and t_2 which are closer to each other than the interval range $b - a$ and ψ_1 holds at all positions between t_1 and t_2. In this case φ is satisfied at all instants t which are at least a from t_1 and at most b from t_2 and up to which ψ_1 holds from t_2 (and thus also t_1) on (see visualization in figure 3.5). A formalisation and proof of this statement can be found in the following lemma, where we abbreviate the model relation for MTL, $\models_{\text{MTL}}$, with $\models$.

Lemma 3.13.
Let $\varphi = \psi_1\,\mathcal{S}_{[a,b]}\,\psi_2$ and $s \in \mathcal{S}_\Sigma^{\mathbb{T}}$ be a timed stream over time domain $(\mathbb{T}, <, 0, \ominus)$ with $\mathbb{R}^{\geq 0}$ distance measure. Let further $t_1, t_2 \in \mathbb{T}$ be two timestamps with $t_1 < t_2, (t_2 \ominus t_1) \leq (b - a)$ s.t.

$$(s, t_1) \models \psi_2 \quad \text{and} \quad (s, t_2) \models \psi_2 \quad \text{and} \quad \forall t' \in \mathbb{T}.\, t_1 < t' < t_2 \rightarrow (s, t') \models \psi_1.$$

Then for any $t \in \mathbb{T}$ with $t > t_2$, $t \ominus t_1 \geq a$ and $t \ominus t_2 \leq b$:

$$(s, t) \models \varphi \quad \text{if and only if} \quad \forall t' \in \mathbb{T}.\, t_2 \leq t' \leq t \rightarrow (s, t') \models \psi_1$$

Proof. Let $t \in \mathbb{T}$ with $t > t_2$, $t \ominus t_1 \geq a$ and $t \ominus t_2 \leq b$ and $(s, t_1) \models \psi_2$, $(s, t_2) \models \psi_2$. Further $\forall t_1 < t' < t_2 . (s, t') \models \psi_1$ and also $\forall t_2 \leq t' \leq t . (s, t') \models \psi_1$, thus $\forall t_1 < t' \leq t . (s, t') \models \psi_1$. Consequently $(s, t) \models \varphi$ as either t_1 or t_2 is within the interval $[a, b]$ from t. □

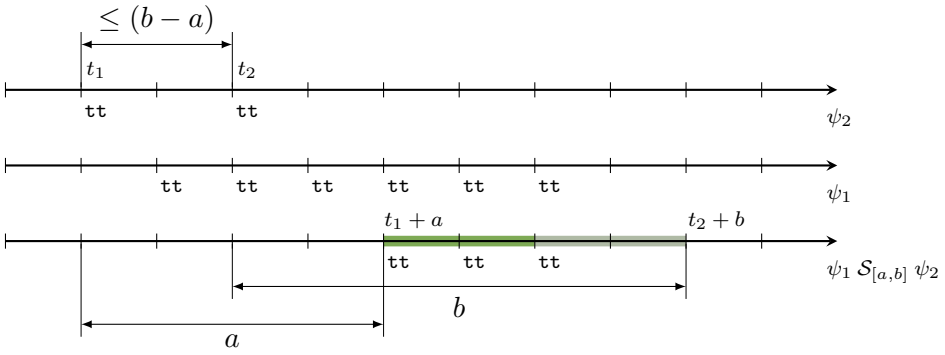


Figure 3.5.: Visualization of lemma 3.13: Since ψ_1 is known to be true at t_1 and t_2 which are closer to each other than $b - a$, the valuation of ψ_2 between t_1, t_2 does not matter. $\psi_1 \, \mathcal{S}_{[a,b]} \, \psi_2$ is true at all instants t more than a from t_1 and less than b from t_2 (marked area) iff ψ_1 is true all the way between t_1 (exclusively) and t. This is because for all such t either $t \ominus t_1 \in [a, b]$ or $t \ominus t_2 \in [a, b]$.

The symmetrical observation can be made for the until operator.

Lemma 3.14.
Let $\varphi = \psi_1 \, \mathcal{U}_{[a,b]} \, \psi_2$ and $s \in \mathcal{S}_\Sigma^{\mathbb{T}}$ a timed stream over time domain $(\mathbb{T}, <, 0, \ominus)$ with $\mathbb{R}^{\geq 0}$ distance measure. Let further $t_1, t_2 \in \mathbb{T}$ be two timestamps with $t_1 < t_2, (t_2 \ominus t_1) \leq (b - a)$ s.t.

$$(s, t_1) \models \psi_2 \quad \text{and} \quad (s, t_2) \models \psi_2 \quad \text{and} \quad \forall t' \in \mathbb{T} . t_1 < t' < t_2 \rightarrow (s, t') \models \psi_1 .$$

Then for any $t \in \mathbb{T}$ with $t < t_1$, $t_2 \ominus t \geq a$ and $t_1 \ominus t \leq b$:

$$(s, t) \models \varphi \quad \text{if and only if} \quad \forall t' \in \mathbb{T} . t_2 \leq t' \leq t \rightarrow (s, t') \models \psi_1$$

Proof. Analogous to proof of lemma 3.13. □

A consequence of these lemmas is, that for evaluating $\psi_1 \, \mathcal{S}_{[a,b]} \, \psi_2$ (or $\psi_1 \, \mathcal{U}_{[a,b]} \, \psi_2$) it is enough to memorize a subset of instants where ψ_2 did hold. Particularly if ψ_2 holds at timestamps t_1 and t_2 closer to each other than $b - a$ and ψ_1 holds at all locations in between, it is not necessary to store further information about instants

between t_1 and t_2. This is because all locations that could be influenced by these instants between are already determined by ψ_2 holding at t_1 and t_2.

This motivates the following strategy for an M(I)TL to LOLA translation over discrete time domains.

For translation of $\varphi = \psi_1 \, \mathcal{S}_{[a,b]} \, \psi_2$ ($\varphi = \psi_1 \, \mathcal{U}_{[a,b]} \, \psi_2$) we create a helper stream s_φ^{set} which stores a set of timestamps at which ψ_2 has been (will be) true. Always when ψ_1 is not true at some instant, s_φ^{set} is reset to the empty set. Using the lemmas from above we remove such timestamps from the set, which are between two other timestamps in the set that are closer than $b-a$. Furthermore timestamps which are further in the past (future) than b are also removed from the set. Technically we can define these stream as follows:

$$
\begin{aligned}
s^{pre}_{\varphi\mathcal{S}_{[a,b]}\psi} \quad &:= \quad \texttt{ite}(s_\psi\texttt{[now]}, \\
&\qquad \texttt{filter}^{\mathcal{S}_{[a,b]}}(s^{set}_{\varphi\mathcal{S}_{[a,b]}\psi}\texttt{[-1|}\emptyset\texttt{]}, \texttt{t[now]}) \cup \{\ \texttt{t[now]}\ \} \\
&\qquad s^{set}_{\varphi\mathcal{S}_{[a,b]}\psi}\texttt{[-1|}\emptyset\texttt{]})
\end{aligned}
$$

$$
s^{set}_{\varphi\mathcal{S}_{[a,b]}\psi} \quad := \quad \texttt{ite}(s_\varphi\texttt{[now]},\ s^{pre}_{\varphi\mathcal{S}_{[a,b]}\psi}\texttt{[now]},\ \texttt{ite}(s_\psi\texttt{[now]},\ \{\ \texttt{t[now]}\ \},\ \emptyset))
$$

$$
\begin{aligned}
s^{pre}_{\varphi\mathcal{U}_{[a,b]}\psi} \quad &:= \quad \texttt{ite}(s_\psi\texttt{[now]}, \\
&\qquad \texttt{filter}^{\mathcal{U}_{[a,b]}}(s^{set}_{\varphi\mathcal{U}_{[a,b]}\psi}\texttt{[+1|}\emptyset\texttt{]}, \texttt{t[now]}) \cup \{\ \texttt{t[now]}\ \} \\
&\qquad s^{set}_{\varphi\mathcal{U}_{[a,b]}\psi}\texttt{[+1|}\emptyset\texttt{]})
\end{aligned}
$$

$$
s^{set}_{\varphi\mathcal{U}_{[a,b]}\psi} \quad := \quad \texttt{ite}(s_\varphi\texttt{[now]},\ s^{pre}_{\varphi\mathcal{U}_{[a,b]}\psi}\texttt{[now]},\ \texttt{ite}(s_\psi\texttt{[now]},\ \{\ \texttt{t[now]}\ \},\ \emptyset))
$$

where the functions $\texttt{filter}^{\mathcal{S}_{[a,b]}} : 2^{\mathbb{T}} \times \mathbb{T} \to 2^{\mathbb{T}}$ and $\texttt{filter}^{\mathcal{U}_{[a,b]}} : 2^{\mathbb{T}} \times \mathbb{T} \to 2^{\mathbb{T}}$ are defined as follows:

$$
\begin{aligned}
\texttt{filter}^{\mathcal{S}_{[a,b]}}(S,t) &= \{t' \in S \mid t' \ominus t \leq b \wedge \nexists t'' \in S.\, t'' < t' < t \wedge (t \ominus t'') < (b-a)\} \\
\texttt{filter}^{\mathcal{U}_{[a,b]}}(S,t) &= \{t' \in S \mid t' \ominus t \leq b \wedge \nexists t'' \in S.\, t < t' < t'' \wedge (t \ominus t'') < (b-a)\}
\end{aligned}
$$

I.e. all elements are filtered away which are further from the current position than b or where the current position and another element in the set are closer than $b-a$. Based on these auxiliary streams we can define $s_{\varphi\mathcal{S}_{[a,b]}\psi}$, $s_{\varphi\mathcal{U}_{[a,b]}\psi}$ by checking all stored timestamps.

$\varphi =$	LOLA stream definition of s_φ
$\varphi \, \mathcal{S}_{[a,b]} \, \psi$	$s_{\varphi\mathcal{S}_{[a,b]}\psi} \ \texttt{:=}\ \exists t \in s^{set}_{\varphi\mathcal{S}_{[a,b]}\psi}\texttt{[now]}.\, (\texttt{t[now]} \ominus t) \in [a,b]$
$\varphi \, \mathcal{U}_{[a,b]} \, \psi$	$s_{\varphi\mathcal{U}_{[a,b]}\psi} \ \texttt{:=}\ \exists t \in s^{set}_{\varphi\mathcal{U}_{[a,b]}\psi}\texttt{[now]}.\, (t \ominus \texttt{t[now]}) \in [a,b]$

Note at his place that the set we utilize to store the timestamps for the evaluation of $\psi_1 \mathcal{S}_{[a,b]} \psi_2$ and $\psi_1 \mathcal{U}_{[a,b]} \psi_2$ will never have more entries than $\frac{b}{b-a} + 1$ if $b \neq a$. This is because all timestamps further away from the current one than b are removed and for each $b - a$ time distance at most one timestamp is contained in the set. Further the size of the set is limited by $\frac{b}{d_{min}} + 1$ where d_{min} is the smallest distance between two timestamps in our time domain. Thus for MTL and MITL monitoring over discrete time domain we have a constant size bound for the involved set data structures. Consequently, the LOLA specification can be transformed into one that represents the sets of at most n elements by n separate streams and applies all set operations directly to these streams. Note that such a specification would not require complex data structures and could be implemented within LOLA's linear arithmetic fragment (see definition 2.43) except from the function `ite`. The translation becomes especially simple if the time domain is regular, then $\frac{b}{b-a}$ streams can be introduced which continuously shift information about the inputs in every $\lfloor b - a \rfloor$ long trace section.

The translation of M(I)TL can now easily be extended towards STL. In the case of STL we have an n channel input signal, which is represented by n input streams $s_1, \ldots, s_n$ of type $\mathbb{R}$ in the LOLA specification. For each STL predicate μ_i we can define a stream s_{μ_i} `:=` μ_i`(`s_1`[now], ... ,`s_n`[now])` which is evaluating the predicate at every timestamp. These streams can then directly be utilized in the M(I)TL translation.

3.4.3. LOLA monitoring

With the insight that a LOLA specification essentially describes a synchronous pointwise property, it appears natural to classify the traditional LOLA monitoring algorithm from [DSS+05] described in section 2.3.1 in terms of the introduced monitoring approaches from the previous section.

Therefore, we first give a definition of a LOLA monitor according to definition 3.1, reflecting the traditional monitoring algorithm from figure 2.9. Note that the algorithm there is casting output events up to the current instant as soon as they can be evaluated into a concrete value. Consequently we cannot rely on the monitor to directly provide us all output values for the current instant. Yet we can think of it to give us an output map containing a concrete value for a random number of streams and instants.

In our classification we would refer to such a device as random access monitor, as it provides information about the monitored property at different instants. Yet in definition 3.7 we model such monitors as kind of question answering machine which can be asked for a specific instant, while the algorithm from figure 2.9 rather suggests push semantics, yielding those values which have newly been computed. We can bridge this gap by adding a final stage to the monitor which is collecting all

the outputs and returning a map of output stream values that have been computed so far. For all streams where the monitor has not cast outputs yet, the map contains ? entries, representing any value from the domain. Thus, formally this mapping is a function $m : \mathbb{T} \to V_{\mathbb{D}_1,\ldots,\mathbb{D}_n}$ with $V_{\mathbb{D}_1,\ldots,\mathbb{D}_n} = \mathbb{D}_1 \cup \{?\} \times \cdots \times \mathbb{D}_n \cup \{?\}$ being the set of output vectors where $\mathbb{D}_i$ for $i \in \{1, \ldots, n\}$ denotes the type of the specification's i^{th} output stream. Such an output function fits to the definition of a random access recurrent monitor. The corresponding output interpretation for the elements from $V_{\mathbb{D}_1,\ldots,\mathbb{D}_n}$ is given as

$$\begin{aligned} \gamma_{V_{\mathbb{D}_1,\ldots,\mathbb{D}_n}} &: V_{\mathbb{D}_1,\ldots,\mathbb{D}_n} \to 2^{\mathbb{D}_1 \times \cdots \times \mathbb{D}_n} \\ \gamma_{V_{\mathbb{D}_1,\ldots,\mathbb{D}_n}}((a_1, \ldots, a_n)) &= \{(c_1, \ldots, c_n) \in \mathbb{D}_1 \times \cdots \times \mathbb{D}_n \mid \forall i \in \{1, \ldots, n\}. \\ &\qquad a_i \neq ? \to a_i = c_i\}. \end{aligned}$$

For illustration consider the LOLA specification in figure 3.6.

```
1  input x: ℝ
2
3  y := (x[now] = 11) ∨ y[-1|false]
4  z := y[+1|true]
5
6  output y
7  output z
```

Figure 3.6.: Example LOLA specification with future offset.

The specification there is receiving a numeric input stream x. Stream y is checking if x did have an event with value 11 up to now. Stream z is true if and only if y is true at

Input sequence	$x = 5$	$x = 7$	$x = 11$	$x = 8$	$\cdots$
Output of algorithm from figure 2.9	$0 : y = \texttt{false}$	$1 : y = \texttt{false}$ $0 : z = \texttt{false}$	$2 : y = \texttt{true}$ $1 : z = \texttt{true}$	$3 : y = \texttt{true}$ $2 : z = \texttt{true}$	$\cdots$
Interpretation as function $\mathbb{N}^+ \to V_{\mathbb{B},\mathbb{B}}$	$\left\{ \begin{array}{l} 0 \to \begin{pmatrix} \texttt{ff} \\ ? \end{pmatrix} \\ 1 \to \begin{pmatrix} ? \\ ? \end{pmatrix} \\ 2 \to \begin{pmatrix} ? \\ ? \end{pmatrix} \\ 3 \to \begin{pmatrix} ? \\ ? \end{pmatrix} \\ \vdots \end{array} \right\}$	$\left\{ \begin{array}{l} 0 \to \begin{pmatrix} \texttt{ff} \\ \texttt{ff} \end{pmatrix} \\ 1 \to \begin{pmatrix} \texttt{ff} \\ ? \end{pmatrix} \\ 2 \to \begin{pmatrix} ? \\ ? \end{pmatrix} \\ 3 \to \begin{pmatrix} ? \\ ? \end{pmatrix} \\ \vdots \end{array} \right\}$	$\left\{ \begin{array}{l} 0 \to \begin{pmatrix} \texttt{ff} \\ \texttt{ff} \end{pmatrix} \\ 1 \to \begin{pmatrix} \texttt{ff} \\ \texttt{tt} \end{pmatrix} \\ 2 \to \begin{pmatrix} \texttt{tt} \\ ? \end{pmatrix} \\ 3 \to \begin{pmatrix} ? \\ ? \end{pmatrix} \\ \vdots \end{array} \right\}$	$\left\{ \begin{array}{l} 0 \to \begin{pmatrix} \texttt{ff} \\ \texttt{ff} \end{pmatrix} \\ 1 \to \begin{pmatrix} \texttt{ff} \\ \texttt{tt} \end{pmatrix} \\ 2 \to \begin{pmatrix} \texttt{tt} \\ \texttt{tt} \end{pmatrix} \\ 3 \to \begin{pmatrix} \texttt{tt} \\ ? \end{pmatrix} \\ \vdots \end{array} \right\}$	$\cdots$

Figure 3.7.: Example run of the standard LOLA evaluation algorithm and corresponding LOLA monitor output for the specification from figure 3.6.

the next instant or the trace ends after the current event. Figure 3.7 shows the output of the monitoring algorithm from figure 2.9 for input stream $x = \langle 5, 7, 11, 8, \dots \rangle$. Below is the interpretation of the output as a function from an instant to a vector of output stream values (y, z).

Based on the considerations made, we now give a formalization of the LOLA monitoring algorithm from figure 2.9 as a synchronous monitor from definition 3.1.

Definition 3.15 (Synchronous LOLA Monitor)**.**

Let $\varphi = (I = \{s_1, \dots, s_n\}, S, O = \{s'_1, \dots, s'_m\}, E)$ be a LOLA specification where stream s_i is of type $\mathbb{D}_i$ and s'_i of type $\mathbb{D}'_i$. The corresponding synchronous monitor is given as $\mathcal{M}_\varphi = (Q, \Sigma, \mathbb{V}, \delta, \gamma, q_0)$ with

- state space $Q = \mathbb{N} \times 2^{\mathbb{E}^{\mathbb{B}}_\varphi} \times (\mathbb{T} \to V_{\mathbb{D}'_1,\dots,\mathbb{D}'_m})$, consisting of the instant of the next inputs, the set of symbolic equations maintained internally and the current output function.
- Input alphabet $\Sigma = \mathbb{D}_1 \times \cdots \times \mathbb{D}_n$ encoding a vector of input readings,
- Output alphabet $\mathbb{V} = (\mathbb{T} \to V_{\mathbb{D}'_1,\dots,\mathbb{D}'_m})$.
- Transition function $\delta : Q \times \Sigma \to Q$ computing the successor state, based on the current state and inputs. For state $q = (t, E, o)$ and input $i = (i_1, \dots, i_n) \in \Sigma$, $\delta(q, i) = (t+1, E', o')$ where E' results from E by adding the input readings to E and execution of the procedures `Compute` and `Prune` from figure 2.9, i.e.

  $$E' = \texttt{Prune}(t, \texttt{Compute}(t, E \cup \{s_j = i_j \mid 1 \leq j \leq n\})).$$

  Output o' results from incorporating all equations from $\texttt{Compute}(t, E \cup \{s^t_j = i_j \mid 1 \leq j \leq n\})$ of form $s'^t_j = v_j$ into o.
- An output function $\gamma((n, U, o)) = o$ returning the current output stored in the state,
- Initial state $q_0 = (0, \emptyset, o_?)$ with $o_?(i) = (?, \dots, ?)$ for all $i \in \mathbb{T}$.

For the following considerations we distinguish between specifications that do not contain future references (very efficiently monitorable fragment of LOLA, see definition 2.42) and unrestricted LOLA specifications.

For the very efficiently monitorable fragment we are able to build perfect recurrent monitors.

Theorem 3.16.
For every very efficiently monitorable LOLA specification φ there is a perfect recurrent monitor.

Proof. Let $\varphi = (I, S, O, E)$ be a flat, very efficiently monitorable LOLA specification and $\mathcal{M}_\varphi = (Q, \Sigma, \mathbb{V}, \delta, \gamma, q_0)$ the corresponding synchronous monitor according to definition 3.15. Let further $w \in \Sigma^{\leq t_{max}+1}$ be a monitor input. Based on this we can construct the monitor $\mathcal{M}'_\varphi = (Q, \Sigma, \mathbb{V}, \delta, \gamma', q_0)$ with $\gamma'((t, E, o)) = o(t-1)$, which is directly casting the output vector for the current instant.

Since φ is very efficiently monitorable and flat, all contained offsets in φ are -1 or 0 offsets. The monitor $\mathcal{M}'_\varphi$ starts in state $(0, \emptyset, o_?)$. For $s[-1, d]$ expressions it uses the default value d in this state. The expression $s[now]$ may either refer to an input stream and can thus be resolved by the corresponding input reading or to an intermediate stream. Besides offset expressions, a defining expression in E may only contain constants and function symbols, which can directly be evaluated by `Compute(`t`,`E`)`. As we require LOLA specifications to be well-formed, there may not be a cyclic dependency between stream events. The value of all intermediate streams can thus be determined iteratively. For all streams s which are referenced as $s[-1, d]$ in φ the computed equations are not pruned and kept in the set of equations during monitoring. Hence in the subsequent step these $s[-1, d]$ expressions can also be resolved and again values for all streams can be determined.

Therefore, after receiving w, $\mathcal{M}'_\varphi$ yields the output $com([\![\varphi]\!](dec(ww')))(|w| - 1)$ for any continuation w' of the received input trace w s.t. $|w| + |w'| = t_{max} + 1$. Note that this output is equal for all $w' \in \Sigma^{t_{max}-|w|-1}$, since φ contains no future references. Consequently $\mathcal{M}'_\varphi$ is a perfect recurrent monitor for φ's induced property, $\mathcal{P}_\varphi$. □

Since a perfect recurrent monitor only has to prepare outputs for the current instant and discards past outputs and intermediate streams which are not referenced by -1 offsets from the state, such a monitor can also be considered trace-length-independent.

However, as the example from figure 3.6 already suggests, the algorithm from definition 3.15 is not perfect anymore, if the specification also contains future offsets.

Theorem 3.17.
The monitor construction from definition 3.15 yields a sound, but in general imperfect random access monitor over any query domain $\mathcal{Q} : \mathbb{N} \to 2^{\mathbb{T}}$ which is non-trivial for at least one instant in $\mathbb{T} \backslash \{t_{max}\}$[a].

[a] We exclude t_{max}, as a monitor which only answers at t_{max} has the full trace available and is thus an offline monitor. Further we don't allow the query domain to yield locations which are out of the instant domain $\mathbb{T}$.

Proof. First we will proof the soundness. Let φ be a LOLA specification with output stream types $\mathbb{D}_1, \ldots, \mathbb{D}_n$ and $\mathcal{M}_\varphi$ the corresponding synchronous monitor according to definition 3.15. Let further $w \in \Sigma^*$ be an arbitrary input trace prefix, $w' \in \Sigma^+$ with $|w| + |w'| = t_{max} + 1$ an input extension and $t \in \{0, 1, \ldots, t_{max}\}$ an instant.

The vector $v = (v_1, \ldots, v_n) = \mathcal{P}_\varphi(ww', t)$ is the valuation of φ at t for w continued with w'. Let $o = (o_1, \ldots, o_n) = \mathcal{M}_\varphi(w)(t)$ be the output of the monitor for instant t after receiving w. We now have to show that $v \in \gamma_{V_{\mathbb{D}_1,\ldots,\mathbb{D}_n}}(o)$, i.e. the result vector v is represented by the output of $\mathcal{M}_\varphi$ for instant t. We consider two cases for each entry o_i of o.

- Either $o_i = ?$, which represents every value at this instant.
- Or $o_i = c$ for a constant c. In this case we also have $v_i = c$, because $o_i = c$ followed from a finite application of equivalence transformations (procedure `Compute`) of defining expressions in φ.

By definition of $\gamma_{V_{\mathbb{D}_1,\ldots,\mathbb{D}_n}}$ it follows that $v \in \gamma_{V_{\mathbb{D}_1,\ldots,\mathbb{D}_n}}(o)$ and $\mathcal{M}_\varphi$ is a sound recurrent monitor for any query domain $\mathcal{Q}$.

Now a counterexample for the perfectness of the random access monitor will be given. W.l.o.g. let $t' \in \mathcal{Q}(t)$ where $t \in \mathbb{T}\backslash\{t_{max}\}$ will serve us as current instant. Let φ be the specification from figure 3.6. If $t' < t$ we slightly modify the specification s.t. stream `z` is defined as `y[`$1 + t - t'$`|true]`.

Consider the random access monitor M_φ for this specification and assume it receives the input trace prefix $w = \langle 0, \ldots, 0, 11\rangle \in \mathbb{R}^{t+1}$. In this case streams z and y evaluate to `true` at all instants after and including t. Thus for instant t' we have $\mathcal{P}_\varphi(ww', t') = (\texttt{true}, \texttt{true})$ if $t' \geq t$ and $\mathcal{P}_\varphi(ww', t') = (\texttt{false}, \texttt{true})$ if $t' < t$ for all input extensions $w' \in \mathbb{R}^{t_{max}-t}$, i.e. $\{\mathcal{P}_\varphi(ww', t') \mid w' \in \mathbb{R}^{t_{max}-1}\} = \{(\texttt{true}, \texttt{true})\}$ or $\{\mathcal{P}_\varphi(ww', t') \mid w' \in \mathbb{R}^{t_{max}-1}\} = \{(\texttt{false}, \texttt{true})\}$, respectively.

Yet, by construction, the monitor M_φ does not add constraints about future events to its internal state and thus yields ? output for z. For y it yields ? if $t' > t$, `true` if $t' = t$ and `false` otherwise. Consequently, the output of the monitor concerning instant t', $M_\varphi(w)(t')$, is either $(?, ?)$ if $t < t'$ or $(\texttt{true}, ?)$ if $t = t'$ or $(\texttt{false}, ?)$ if $t' < t$. By definition of $\gamma_{V_{\mathbb{B},\mathbb{B}}}(M_\varphi(w)(t'))$, non of them matches the perfect output for t', $\{(\texttt{false}, \texttt{true})\}$ or $\{(\texttt{true}, \texttt{true})\}$, respectively. Hence M_φ does not yield a perfect output for $t' \in \mathcal{Q}(t)$. □

As discussed earlier in this chapter, a consequence of theorem 3.17 is that the traditional LOLA monitoring algorithm does also not yield a perfect initial nor a perfect ($k-$offset) recurrent monitor, as they can be emulated by a random access monitor with corresponding query domain.

> **Corollary 3.18.**
>
> The monitor construction from definition 3.15 yields in general neither a perfect initial, nor a perfect k-offset recurrent monitor for any $k \in \mathbb{Z}$ with $|k| < t_{max}$.

Further note, that the algorithm from definition 3.15 is also not able to cope with uncertain input readings and the presence of assumptions.

3.4.4. Other SRV languages

Besides standard LOLA, two extensions of it, namely LOLA 2.0 and RTLola, and two further SRV languages, TeSSLa and Striver, were touched upon in section 2.3. While in this thesis we mainly focus on LOLA, here is a quick overview how these formalisms can be considered in terms of our monitoring classification.

As discussed in section 2.3.1 LOLA 2.0 and RTLola can principally (with overhead) be emulated in LOLA. As such, specifications in these languages can also be understood as or represented by pointwise properties. Yet in the case of RTLola one might have to prepare "dummy" inputs during monitoring at instants of the time domain where no actual inputs are available (see [Sch22]).

This is pretty similar to the case of Striver. While it can be considered an asynchronous SRV formalism because the outputs can appear at other non-regular positions than the inputs, it is restricted to non-zeno output behavior. Therefore, any Striver specification can still be mimicked by a LOLA specification with the same strategy as used for RTLola (see [Sch22, GDS20]).

The language TeSSLa, on the other hand differs quite heavily from LOLA. TeSSLa specifications can describe the values of output streams at arbitrary timestamps from a dense time domain, independent from the input timestamps; there is no notion of a global pulse, at which events occur. This concept however does neither fit to the definition of pointwise properties (definition 3.4) nor to synchronous monitoring (definition 3.1) on which this chapter is largely based. A TeSSLa specification rather expresses a continuous property, i.e. a function assigning a value to every instant of a dense time domain. Such properties would require a different kind of monitor, which generates an output that yields information about a continuous time range or can be „asked“ about a specific timestamp. A monitor for this scenario could be based on the model of a random access recurrent monitor, but would require an advanced output data structure.

However, it should be noted at this point that TeSSLa without its `delay` operator has synchronous character as well and can thus be expressed in LOLA [Sch22]. Yet in this case, LOLA is even more expressive than TeSSLa, due to the missing ability of future references in TeSSLa. Thus, despite LOLA, TeSSLa is not able to express arbitrary pointwise (synchronous) properties, but only future-independent, i.e. very efficiently monitorable, ones.

In the remainder of this thesis we restrict ourselves to synchronous properties and LOLA as universal formalism for them. A further elaboration on Striver, TeSSLa and other asynchronous formalisms will be left for future work.

3.5. Summary

In this chapter, we have examined different kinds of synchronous properties, initial and pointwise ones, and classified the corresponding monitoring approaches. We have defined a notion of soundness and perfectness for these monitors, which can be considered as continuation or generalization of the concepts anticipation and impartiality, as known from the LTL_3 monitoring approach. Further we have seen how synchronous stream runtime verification languages fit with the definition of pointwise properties and exemplary showed how several other formalisms can be embedded in these highly expressive languages. Yet the standard monitoring algorithm for LOLA, as presented in chapter 2 bears two drawbacks in terms of our general view on monitoring

- It does not provide utmost precise outputs for specifications with future offsets.
- It is not able to handle uncertain inputs and assumptions (besides simple monitoring of them).

In the following chapter, the main contribution of this thesis will be presented. We will overcome the limitations mentioned above by introduction of a recurrent monitoring algorithm for LOLA, based on abstraction, which is able to handle uncertainty and assumptions. Further this monitor will be shown to be perfect under particular circumstances.

4

A LOLA monitoring framework

In the previous chapter, we have examined the process for synchronous monitoring of pointwise properties in general (which also subsumes the special case of initial monitoring). It was found that the stream runtime verification language LOLA can be used as a general formalism to specify this kind of properties. However, the traditional LOLA monitoring algorithm [DSS+05] is not able to deliver a perfect (i.e. anticipatory and impartial) recurrent monitor for properties which involve future references and is likewise not capable of handling the presence of uncertain inputs and assumptions.

In this chapter we will discuss approaches for sound and perfect monitoring of LOLA specifications under presence of uncertainties and assumptions with consideration of the future. To this end we will first extend the LOLA semantics from the preliminaries chapter to a so-called monitoring semantics, capable of handling input prefixes and input uncertainties.

Based on this semantics, we will develop a generic theory to build sound and perfect recurrent LOLA monitors. Therefore, we will introduce an abstraction-based framework utilizing ideas and insights from the field of abstract interpretation. In addition to the presented recurrent monitoring approach, there will be demonstrated how several common advanced monitoring tasks can actually be reduced to recurrent monitoring in LOLA.

In the subsequent chapter there will be a detailed description of a symbolic implementation, which instantiates the generic framework.

The results of this and the following chapter are mainly based on [HKLS24, KLS22a].

4.1. Basic notations

For convenience, we will use some common notations throughout the following sections.

Let $\varphi = (\{s_1, \ldots, s_n\}, \{s_{n+1}, \ldots, s_{n+m}\}, \{s'_{o_1}, \ldots, s'_{o_l}\}, E)$ be a LOLA specification with input stream identifiers $s_1, \ldots, s_n$, intermediate stream identifiers $s_{n+1}, \ldots, s_{n+m}$ and output stream identifiers $s_{o_1}, \ldots, s_{o_l}$ for $o_1, \ldots, o_l \in \{n+1, \ldots, n+m\}$.

With

$$\mathbf{D}_{\varphi}^{\mathbf{in}} = \mathbb{D}_1 \times \cdots \times \mathbb{D}_n$$

we denote the product of all input stream types corresponding to $s_1, \ldots, s_n$, with

$$\mathbf{D}_{\varphi}^{\mathbf{int}} = \mathbb{D}_{n+1} \times \cdots \times \mathbb{D}_{n+m}$$

the product of all intermediate stream types corresponding to $s_{n+1}, \ldots, s_{n+m}$ and we use

$$\begin{aligned} \mathbf{D}_{\varphi}^{\mathbf{out}} &= \mathbb{D}_{o_1} \times \cdots \times \mathbb{D}_{o_l}, \\ \mathbf{D}_{\varphi} &= \mathbf{D}_{\varphi}^{\mathbf{in}} \times \mathbf{D}_{\varphi}^{\mathbf{int}}. \end{aligned}$$

Analogously for tuples of streams we use

$$\begin{aligned} \mathbf{S}_{\varphi}^{\mathbf{in}} &= \mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_n}, \\ \mathbf{S}_{\varphi}^{\mathbf{int}} &= \mathcal{S}_{\mathbb{D}_{n+1}} \times \cdots \times \mathcal{S}_{\mathbb{D}_{n+m}}, \\ \mathbf{S}_{\varphi}^{\mathbf{out}} &= \mathcal{S}_{\mathbb{D}_{o_1}} \times \cdots \times \mathcal{S}_{\mathbb{D}_{o_l}} \text{ and} \\ \mathbf{S}_{\varphi} &= \mathbf{S}_{\varphi}^{\mathbf{in}} \times \mathbf{S}_{\varphi}^{\mathbf{int}}. \end{aligned}$$

Additionally, we assume for this and the subsequent chapter that all LOLA specifications are in flattened form.

4.2. LOLA semantics revisited

The solution of a LOLA specification is defined as a tuple of streams, satisfying the corresponding equation system of the specification (see definition 2.38). In the case of well-defined LOLA specifications, to which we restrict in this thesis, the solution of the specification is unique and can thus be considered as stream transformation, from a tuple of completely known input streams from instant 0 to t_{max} to a tuple of completely known output streams.

For illustration consider the LOLA specification φ in figure 4.1.

Stream s sums up all values of input stream i from the current instant to the stream end. Assume $\mathbb{T} = \{0, 1, 2\}$, and $\sigma \in \mathcal{S}_{\mathbb{R}}$ as input stream for identifier i. Let further $\Pi = (\sigma, \pi) \in \mathcal{S}_{\mathbb{R}} \times \mathcal{S}_{\mathbb{R}}$ be the tuple of all (input and output) streams of the

```
1  input i: ℕ
2
3  s := s[+1|0] + i[now]
4
5  output s
```

Figure 4.1.: Example LOLA specification

specification. The semantics of s's defining expression `s[+1|0] + i[now]` would be given as

$$\begin{array}{lll} [\![s[+1|0] + i[now]]\!]_{\Pi,\varphi}(0) & = [\![s[+1|0]]\!]_{\Pi,\varphi}(0) + \sigma(0) & = \pi(1) + \sigma(0) \\ [\![s[+1|0] + i[now]]\!]_{\Pi,\varphi}(1) & = [\![s[+1|0]]\!]_{\Pi,\varphi}(1) + \sigma(1) & = \pi(2) + \sigma(1) \\ [\![s[+1|0] + i[now]]\!]_{\Pi,\varphi}(2) & = [\![s[+1|0]]\!]_{\Pi,\varphi}(2) + \sigma(2) & = 0 + \sigma(2) \end{array}$$

Thus the only output stream π for identifier s, which is conformant with the semantics of its defining expression `s[+1|0] + i[now]` and input streams σ is

$$\pi = [\![\varphi]\!](\sigma) = \langle \sigma(2) + \sigma(1) + \sigma(0), \sigma(2) + \sigma(1), \sigma(2) \rangle.$$

Following [CHL+18], we will consider this kind of recursively defined LOLA semantics as a fixed point equation (referred to as *LOLA fixed point semantics*).

Therefor let $\varphi = (I = \{s_1, \ldots, s_n\}, S = \{s_{n+1}, \ldots, s_{n+m}\}, O = \{s_{i_1}, \ldots, s_{i_l}\}, E)$ be a LOLA specification. Provided fully known input streams $\Sigma = (\sigma_1, \ldots, \sigma_n) \in \mathbf{S}^{\mathbf{in}}_{\varphi}$, a fixed point equation for φ can be given as function

$$\begin{array}{lcl} [\![\varphi]\!]^{fp}_{\Sigma} & : & \mathbf{S}_{\varphi} \to \mathbf{S}_{\varphi} \\ [\![\varphi]\!]^{fp}_{\Sigma}(\Pi) & = & (\sigma_1, \ldots, \sigma_n, [\![E(s_{n+1})]\!]_{\Pi,\varphi}, \ldots, [\![E(s_{n+m})]\!]_{\Pi,\varphi}) \end{array}$$

By definition 2.38 the solutions of $[\![\varphi]\!]^{fp}_{\Sigma}$ for fixed input streams Σ are exactly the solutions of the LOLA specification. Consequently, if φ is well-defined, also $[\![\varphi]\!]^{fp}_{\Sigma}$ has a unique fixed point $\mu([\![\varphi]\!]^{fp}_{\Sigma}) = \nu([\![\varphi]\!]^{fp}_{\Sigma})$. Thus, for $(\pi_1, \ldots, \pi_{n+m}) = \mu([\![\varphi]\!]^{fp}_{\Sigma})$ the semantics of the LOLA specification is given as

$$\begin{array}{lcl} [\![\varphi]\!] & : & \mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_n} \to \mathcal{S}_{\mathbb{D}_{i_1}} \times \cdots \times \mathcal{S}_{\mathbb{D}_{i_l}} \\ [\![\varphi]\!](\Sigma) & = & (\pi_{i_1}, \ldots, \pi_{i_l}). \end{array}$$

4.2.1. Monitoring semantics for LOLA

The standard LOLA semantics relates a tuple of fully known input streams with the corresponding tuple of intermediate streams. Yet in the process of online monitoring only prefixes of the input streams are available and output streams are iteratively computed by the monitor. In fact, the definition of perfect recurrent monitors in

definition 3.8 determines which monitor outputs for received input prefixes are considered perfect with respect to a semantics (property) for complete traces. However, the definition does not provide a construction for such a monitor nor reveals information about feasibility of perfect monitoring.

Following [Sch24] we define a so-called *monitoring semantics*, on which we will later base our monitor construction. The idea behind this kind of semantics is to provide a mapping from incomplete input streams to incomplete output streams which bear as much information as possible. The definition will also be given as fixed point equation, based on the one above. In the remainder of this chapter we will then use this semantics to build abstractions on it, and make it monitorable this way.

We start with the definition of so-called monitoring (event) streams (i.e. incomplete streams). Here, we can use the idea from [Sch24] which we similarly introduced for dealing with uncertainties in section 2.2.4: We define a monitoring event stream as a set of concrete streams of full length. We will use them to encode all concrete streams which are compatible with the inputs the monitor has received so far also with respect to uncertain inputs.

> **Definition 4.1** (Monitoring event stream; based on [Sch24])**.**
>
> A set of synchronous event streams $S \subseteq \mathcal{S}_{\mathbb{D}}$ of type $\mathbb{D}$ is called a *monitoring (event) stream of type* $\mathbb{D}$.

Imagine for example $\mathbb{T} = \{0, 1, \ldots, 4\}$ and an input stream of type $\mathbb{N}$, where the monitor has received the trace $\langle 0, 4, 2\rangle$ so far. The last two events are not yet available. The corresponding monitoring stream S would be given as

$$S = \{\langle 0, 4, 2, a, b\rangle \mid a, b \in \mathbb{N}\}.$$

In the following we will use the notation $S(t)$ for $t \in \mathbb{T}$ to refer to the set of all possible values of S at instant t. E.g. $S(0) = \{0\}$ and $S(3) = \mathbb{N}$ for the example above.

Based on this notion of incomplete streams we want to define a fixed point semantics that relates input monitoring event streams with output monitoring event streams that reflect as much information about the output streams as possible. Since output and intermediate streams of LOLA specifications are usually not independent of each other, because events of different streams can be related, it is in general not sufficient for perfect monitoring to consider the output or intermediate streams of a specification as individual monitoring event streams. Instead we introduce the notion of a monitoring event stream tuple, which is a set of event stream tuples, encoding all possible combinations of concrete event streams.

Definition 4.2 (Monitoring event stream tuple; based on [HKLS24])**.**

A set of synchronous event streams tuples $T \subseteq \mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_n}$ of types $\mathbb{D}_1, \ldots, \mathbb{D}_n$ is called *monitoring (event) stream tuple of types* $\mathbb{D}_1, \ldots, \mathbb{D}_n$.

With $\mathcal{T}_{\mathbb{D}_1 \times \cdots \times \mathbb{D}_n} = 2^{\mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_n}}$ we denote the set of all monitoring event stream tuples of types $\mathbb{D}_1, \ldots, \mathbb{D}_n$.

For illustration consider the LOLA specification in figure 4.2.

```
1    input i: ℕ
2
3    u := i[now] + 1
4    v := i[now] + 2
5
6    output u
7    output v
```

Figure 4.2.: Example LOLA specification

The monitoring event stream tuple T for the two output streams u, v, given the input monitoring stream tuple $S = \{(\langle 0, 4, 2, a, b\rangle) \mid a, b \in \mathbb{N}\}$, analogous to the one from above, would be:

$$T = \{(\langle 1, 5, 3, c, d\rangle, \langle 2, 6, 4, c+1, d+1\rangle) \mid c, d \in \mathbb{N}^+\}$$

I.e. the set of tuples of two streams which start with $1, 5, 3$ and $2, 6, 4$ followed by two arbitrary positive numbers where the value of the second stream exceeds the one of the first stream by exactly 1. Note that this monitoring stream tuple thus keeps the relation, that the two streams differ by exactly one. In case we would represent both streams as separate monitoring event streams this connection would be lost.

Again, we use the notation $T(t)$ for $t \in \mathbb{T}$ to denote the set of all stream value tuples at instant t. For the example above, $T(0) = \{(1, 2)\}$, $T(3) = \{(c, c+1) \mid c \in \mathbb{N}^+\}$.

We now define a derived version of LOLA's fixed point semantics from above which is operating on monitoring stream tuples and is thus able to handle incomplete and uncertain streams. This adjusted fixed point equation takes a monitoring stream tuple of the input and intermediate streams and also returns such a monitoring stream tuple. The semantics is again invariant in the particular input readings, which are also given as monitoring stream tuple.

Definition 4.3 (LOLA monitoring fixed point equation)**.**

Let $\varphi = (I = \{s_1, \ldots, s_n\}, S = \{s_{n+1}, \ldots, s_{n+m}\}, O, E)$ be a LOLA specification and $\Sigma \in \mathcal{T}_{\mathbf{D}^{\mathbf{in}}_{\varphi}}$ a monitoring event stream tuple corresponding to input stream identifiers $s_1, \ldots, s_n$.

The *monitoring fixed point equation* of φ is given as

$$\begin{aligned}&[\![\varphi]\!]^{mon}_{\Sigma} \quad : \mathcal{T}_{\mathbf{D}_\varphi} \to \mathcal{T}_{\mathbf{D}_\varphi}\\ &[\![\varphi]\!]^{mon}_{\Sigma}(T) = \{[\![\varphi]\!]^{fp}_{(\pi_1,\dots,\pi_n)}(\pi_1,\dots,\pi_{n+m}) \mid (\pi_1,\dots,\pi_{n+m}) \in T, (\pi_1,\dots,\pi_n) \in \Sigma\}\end{aligned}$$

A solution of the monitoring fixed point equation is a monitoring stream tuple over input and intermediate streams. Specifically a set of stream tuples, where the input streams are contained in the input monitoring stream tuple and where each stream tuple results from applying $[\![\varphi]\!]^{fp}_{(\pi_1,\dots,\pi_n)}$ to a tuple in the set.

In contrast to the fixed point equation for a single fully known LOLA stream, the equation above does not have a unique fixed point. Consider e.g. the monitoring stream tuple

$$\Sigma = \{(\langle 1,1,1,1\rangle), (\langle 2,2,2,2\rangle)\}$$

representing a stream over time domain $\mathbb{T} = \{0,1,2,3\}$ which either has the value 1 or 2 at every instant.

Let φ be the LOLA specification from figure 4.2. Using Σ as input monitoring stream tuple, the four fixed points of $[\![\varphi]\!]^{mon}_{\Sigma}$ would be (for streams i, u, v in this order):

$$\begin{gathered}\emptyset,\\ \{(\langle 1,1,1,1\rangle, \langle 2,2,2,2\rangle, \langle 3,3,3,3\rangle)\},\\ \{(\langle 2,2,2,2\rangle, \langle 3,3,3,3\rangle, \langle 4,4,4,4\rangle)\},\\ \{(\langle 1,1,1,1\rangle, \langle 2,2,2,2\rangle, \langle 3,3,3,3\rangle), (\langle 2,2,2,2\rangle, \langle 3,3,3,3\rangle, \langle 4,4,4,4\rangle)\}\end{gathered}$$

Since we want to consider all possible inputs given by Σ, we are interested in the greatest fixed point of $[\![\varphi]\!]^{mon}_{\Sigma}$ according to the subset relation of the monitoring event stream tuples $\cdot \subseteq \cdot$. Note that monitoring event stream tuples together with this order form a complete lattice where the meet and join operation corresponds to the intersection $(\cdot \cap \cdot)$ and union $(\cdot \cup \cdot)$ operation for sets. The minimal element is consequently the empty set and the maximal one the set of all stream tuples, which we denote as $\top^{mon}$. It is easy to see that $[\![\varphi]\!]^{mon}_{\Sigma}$ is monotonic and thus has a least and greatest fixed point (see theorem 2.47). While the least fixed point is always $\emptyset$, the greatest one is equal to the set of all inputs concatenated with the intermediate streams that result from applying the standard LOLA semantics on these streams, i.e. the solution we are interested in:

Lemma 4.4.
Let $\varphi = (I = \{s_1,\dots,s_n\}, S, S, E)$ be a LOLA specification and $\Sigma \in \mathcal{T}_{\mathbf{D}^{\mathbf{in}}_\varphi}$ a monitoring event stream tuple for input stream identifiers $s_1,\dots,s_n$.

The greatest fixed point of $[\![\varphi]\!]^{mon}_{\Sigma}$ is given as

$$\nu([\![\varphi]\!]^{mon}_{\Sigma}) = \{(\sigma_1, \ldots, \sigma_n) \circ [\![\varphi]\!](\sigma_1, \ldots, \sigma_n) \mid (\sigma_1, \ldots, \sigma_n) \in \Sigma\}.$$

Proof. It is easy to see that $\{(\sigma_1, \ldots, \sigma_n) \circ [\![\varphi]\!](\sigma_1, ..., \sigma_n) \mid (\sigma_1, \ldots, \sigma_n) \in \Sigma\}$ is a fixed point of $[\![\varphi]\!]^{mon}_{\Sigma}$, because all input streams $(\sigma_1, \ldots, \sigma_n)$ are from Σ and $[\![\varphi']\!](\sigma_1, ..., \sigma_n)$ is equal to the tuple of intermediate streams from the fixed point of $[\![\varphi]\!]^{fp}_{(\sigma_1,...,\sigma_n)}$ as argued above.

We further have to show that there is no greater fixed point. Therefore we assume the opposite, i.e. $\nu([\![\varphi]\!]^{mon}_{\Sigma}) \neq \{(\sigma_1, \ldots, \sigma_n) \circ [\![\varphi]\!](\sigma_1, \ldots, \sigma_n) \mid (\sigma_1, \ldots, \sigma_n) \in \Sigma\}$. In this case there must be a stream tuple $\Pi \in \nu([\![\varphi]\!]^{mon}_{\Sigma})$ that starts with the input streams $(\sigma_1, \ldots, \sigma_n) \in \Sigma$, yet $\Pi \neq (\sigma_1, \ldots, \sigma_n) \circ [\![\varphi]\!](\sigma_1, \ldots, \sigma_n)$.

Thus, Π must differ in at least one event of some stream from $\Upsilon = (\sigma_1, \ldots, \sigma_n) \circ [\![\varphi]\!](\sigma_1, \ldots, \sigma_n)$. Let $D \subseteq S \times \mathbb{T}$ denote the set of all stream identifiers and positions (i.e. events) where Π is different from Υ. Since Π must be a stream tuple that results from application of $[\![\varphi]\!]^{fp}_{(\sigma_1,...,\sigma_n)}$ (otherwise it could not be contained in $\nu([\![\varphi]\!]^{mon}_{\Sigma})$), but is not the (unique) fixed point of it (as this would imply $\Upsilon = \Pi$), there must be another stream tuple $\Pi' \in \nu([\![\varphi]\!]^{mon}_{\Sigma})$, s.t. $\Pi = [\![\varphi]\!]^{fp}_{(\sigma_1,...,\sigma_n)}(\Pi')$. Observe that for each $d \in D$ this stream tuple Π' must differ from Υ in at least one stream and instant on which the event at d is depending. This is because $\Pi = [\![\varphi]\!]^{fp}_{(\sigma_1,...,\sigma_n)}(\Pi')$ and $\Upsilon = [\![\varphi]\!]^{fp}_{(\sigma_1,...,\sigma_n)}(\Upsilon)$ and if Π' and Υ would not differ for events on which events in D depend, these events in D and likewise Π and Υ would be equal. Since a well-defined LOLA specification does not allow that an event is dependent on itself, the elements in D may not have cyclic references and so Π' must differ from Υ in at least one stream and positions which is not already contained in D.

We can add this stream and position to D and apply the same argument inductively: There must be a further stream tuple $\Pi'' \in \nu([\![\varphi]\!]^{mon}_{\Sigma})$ s.t. $\Pi' = [\![\varphi]\!]^{fp}_{(\sigma_1,...,\sigma_n)}(\Pi'')$ which differs from Υ in even further streams and positions which we can add to D etc.; This however directly leads to a contradiction, as the maximum size of elements in D is limited by the number of streams in the specification and instants in the instant domain. Thus, $\{(\sigma_1, \ldots, \sigma_n) \circ [\![\varphi]\!](\sigma_1, \ldots, \sigma_n) \mid (\sigma_1, \ldots, \sigma_n) \in \Sigma\}$ is the greatest fixed point of $[\![\varphi]\!]^{mon}_{\Sigma}$. □

Using the greatest fixed point of the equation from definition 4.3 we can define the *monitoring semantics* of φ. This semantics is a function from the monitoring stream tuple of the input streams to the monitoring stream tuple of the specification's output streams. It originates from the greatest solution of the monitoring fixed point equation w.r.t. the given inputs.

Definition 4.5 (LOLA monitoring semantics)**.**
Let $\varphi = (I = \{s_1, \ldots, s_n\}, S = \{s_{n+1}, \ldots, s_{n+m}\}, O = \{s'_{o_1}, \ldots, s'_{o_l}\}, E)$ be a LOLA specification. The *monitoring semantics* of φ is the function

$$\begin{array}{lcl} [\![\varphi]\!]^{mon} & : & \mathcal{T}_{\mathbf{D}_\varphi^{\mathbf{in}}} \to \mathcal{T}_{\mathbf{D}_\varphi^{\mathbf{out}}} \\ [\![\varphi]\!]^{mon}(\Sigma) & = & \{(\omega_{o_1}, \ldots, \omega_{o_l}) \mid (\omega_1, \ldots, \omega_{n+m}) \in \nu([\![\varphi]\!]^{mon}_\Sigma)\} \end{array}$$

4.3. Recurrent LOLA monitoring

We will now discuss concepts for building LOLA monitors, based on the specification's monitoring semantics from the previous section. Therefore, we will introduce a generic abstraction framework of the semantics from definition 4.3. In the subsequent section we will then discuss a symbolic approach as possible instantiation of the framework.

In chapter 3, the concepts of recurrent and random access recurrent monitoring have been presented. While random access monitoring was shown to be a very powerful approach, it was already pointed out that it usually faces feasibility issues. This is because it generally requires that information about event values at any instant and their combination is stored by the monitor. This however leads to trace-length-dependence and may make monitoring for realistic trace lengths from hundreds to sometimes even millions of events intractable.

The situation is different for recurrent monitors which yield outputs only for the current instant. Therefore in this section we will discuss a recurrent monitoring approach for LOLA specifications. However, due to the expressiveness of LOLA, several random access monitoring tasks and other advanced monitoring techniques can be reduced to recurrent monitoring, which makes the presented approach quite powerful. Thus, before details of the monitoring algorithm are presented, the following subsection quickly outlines how such reductions can be performed.

4.3.1. Monitoring reductions in LOLA

To carry out advanced monitoring tasks with a recurrent LOLA monitor we add additional streams to the specification. These access the original output stream(s) by offsets and thus provide information about the output values at different locations.

Consider a LOLA specification with a single output stream o (the following constructions can in general also be applied to an arbitrary number of output streams). We can add another stream o' for

- **k-offset recurrent monitoring**, if we define

  o' := o[$-k$|$\perp$]

  where $\perp$ is a default value in the domain of o indicating that there is no k-offset position of o.

- **initial monitoring**, if we define

  o' := ite(ff[-1|true], o[*now*], o'[-1|$\perp$])

  where `ff` is an auxiliary stream which is defined to be `false` at all instants. Note that the expression `ff[-1|true]` is always false except at instant 0 and can thus be used to detect the beginning of the stream. At instant 0, stream o' takes over the value of o and at other positions the previous value of o'. This makes all events of o' equal to o's first event and a perfect recurrent monitor will cast the most precise information about o at instant 0 in each step.

- **monitoring with reset** (c.f. [CTT19]), if we define

  o' := ite(r[*now*], o[*now*], o'[-1|$\perp$])

  where r is an additional boolean input stream indicating a reset of the monitor. Stream o' always carries the value of stream o when the reset event was triggered the last time and $\perp$ up to the first reset.

- **monitoring the distance to the next satisfaction of o** (c.f. [KLSS22]), if we add the definition

  o' := ite(o[*now*], 0, o'[+1|∞] + 1).

  This obviously requires o to be a boolean stream. Stream o' counts the number of steps to the next instant where o is true. A recurrent monitor could provide possible intervals for o', revealing whether o is inevitably satisfied or not, or whether o is known to be false for a certain number of instants. This kind of monitoring might especially be useful to counter inevitable system failures at early stages [KLSS22].

LOLA also makes a combination of several of the upper monitoring techniques possible. E.g. one is able to monitor the values of stream o in a fixed sliding window around the current position by using several k-offset streams in parallel.

4.3.2. Prerequisites for monitor construction

Before we come up with a generic LOLA monitoring approach we first discuss some fundamentals for building such monitors. We start with the translation of input trace prefixes to monitoring stream tuples as required by the theory from the previous section. Subsequently we study the incorporation of uncertainties and assumptions.

Input readings

Note that while the monitoring stream tuples from the previous section are sets of potential streams of full length, the monitors discussed in chapter 3 receive prefixes (sequences of value tuples for all input streams) of the full input, i.e. traces from $(\mathbb{D}_1 \times \cdots \times \mathbb{D}_n)^{\leq t_{max}+1}$. We thus start with the definition of helper function *conv*, translating between the input the monitor receives and the corresponding monitoring stream tuple. For first we do not consider uncertainty in the inputs but rather assume that they are fully known up to the received instant. Therefore, *conv* returns a monitoring event stream tuple, where the beginning of each stream matches the received trace prefix and all possible continuations for the not-yet-received part are contained.

$$\begin{aligned} \mathit{conv} &: (\mathbb{D}_1 \times \cdots \times \mathbb{D}_n)^{\leq t_{max}+1} \to \mathcal{T}_{\mathbb{D}_1 \times \cdots \times \mathbb{D}_n} \\ \mathit{conv}(w) &= \{(\langle \pi_1(w(0)) \ldots \pi_1(w(|w|-1)), u_1^{|w|} \ldots u_1^{t_{max}} \rangle, \ldots, \\ &\qquad \langle \pi_n(w(0)) \ldots \pi_n(w(|w|-1)), u_n^{|w|} \ldots u_n^{t_{max}} \rangle) \mid u_j^i \in \mathbb{D}_j\} \end{aligned}$$

where $\pi_i((v_1, \ldots, v_n)) = v_i$ denotes the i^{th} component of a tuple.

For example consider $\mathbb{T} = \{0, 1, 2, 3\}$ and monitor input

$$w = \langle (1, \texttt{true}), (2, \texttt{false}) \rangle \in (\mathbb{N} \times \mathbb{B})^*$$

which encodes the values of two streams (one of type $\mathbb{N}$ and one of type $\mathbb{B}$) at positions 0,1. The corresponding monitoring event stream tuple is

$$\mathit{conv}(w) = \{(\langle 1, 2, u_1^2, u_1^3 \rangle, \langle \texttt{true}, \texttt{false}, u_2^2, u_2^3 \rangle) \mid u_1^2, u_1^3 \in \mathbb{N}, u_2^2, u_2^3 \in \mathbb{B}\}.$$

Uncertainty

We now enrich the interpretation of input traces with the presence of uncertainties. As suggested in section 3.3.3 we model uncertainty in a way that the monitor receives an input over a special observation domain Γ which encodes an uncertain trace, i.e. a set of possible input traces (see definition 2.28). Such an uncertain trace is already strongly related to a monitoring stream tuple, which is also representing a set of potential stream tuples. However, the difference to the monitoring stream tuples we used before to encode inputs is that now not all events are fully known up to

a certain point in time and unknown after that. In the case of uncertainty, events at arbitrary instants may fully or partially be unknown, as well as the relationship among them.

In the previous subsection we defined the function $conv : (\mathbb{D}_1 \times \cdots \times \mathbb{D}_n)^* \to \mathcal{T}_{\mathbb{D}_1 \times \cdots \times \mathbb{D}_n}$ that converts trace prefixes to corresponding monitoring stream tuples. Assume an uncertain input domain Γ and an uncertainty encoding $\nu : \Gamma^* \to 2^{(\mathbb{D}_1 \times \cdots \times \mathbb{D}_n)^*}$, translating an input prefix over Γ into an uncertain input trace (see definition 2.30). We extend the function *conv* to produce monitoring streams from uncertain input traces in the following way:

$$\begin{aligned} uconv_\nu &: \Gamma^* \to \mathcal{T}_{\mathbb{D}_1 \times \cdots \times \mathbb{D}_n} \\ uconv_\nu(w) &= \bigcup_{v \in \nu(w)} conv(v) \end{aligned}$$

I.e. we union all monitoring event stream tuples which are corresponding to a concrete trace prefix represented by w.

As an example consider a specification with a single input stream of type $\mathbb{R}$. The uncertain domain is $\Gamma = \mathbb{R} \times \mathbb{R} \cup \{?\}$ together with encoding ν where $(a, b) \in \mathbb{R} \times \mathbb{R}$ encodes an uncertain value between $a \in \mathbb{R}$ and $b \in \mathbb{R}$ (both inclusive) and ? a fully uncertain input. For $\mathbb{T} = \{0, \ldots, 4\}$, the monitor input $i = \langle (2, 5), ?, (7, 7) \rangle$ would be translated to the monitoring stream

$$uconv_\nu(i) = \{(\langle u_0, u_1, 7, u_3, u_4 \rangle) \mid 2 \leq u_0 \leq 5, u_i \in \mathbb{R} \text{ for } i \in \{1, 3, 4\}\}$$

We will use the function $uconv_\nu$ instead of *conv* in our monitoring theory to support uncertainty.

Uncertainty supported in recurrent monitoring

The uncertainty support in the recurrent monitoring approach that will be presented in this chapter must be limited to ensure trace-length-independent monitoring. The restrictions are outlined below.

First, as discussed in chapter 3, it is an obstacle to recurrent monitoring if an uncertain input encodes certain inputs of different lengths. This is because in this case there is no unique, current instant for which outputs can be cast. While the definition of a recurrent monitor which deals with this kind of uncertain traces is in principle possible (see definition 3.10) this leads to the case that outputs for different trace positions get mixed, which is oftentimes not desired. Therefore we do not handle this case for LOLA in this chapter. In particular, we restrict to uncertain traces that encode a set of concrete traces of the same length as the uncertain trace.

Additionally for our recurrent monitoring approach we exclude uncertainties which relate events across instants and restrict to *timestamp-immanent uncertainty* (also

referred to as *instant-immanent uncertainty*). For example we do not allow something like "The value of s is `true` at instant 5, if it was also `true` at instant 2 and otherwise `false`". If we would support such kinds of uncertainties, we would ultimately have to store the full received trace in the monitor. If the relations among instants are known to be within a certain bound, however, the monitoring algorithm could in principle be adjusted to support these kind of uncertainties. In this case, the input readings from previous instants would have to be stored until an instant is reached where they cannot have influence on current input values anymore. However, we will not handle this in the following.

In summary, we formally demand for a LOLA specification φ and a sequence of uncertain input prefixes $w^{(0)}, w^{(1)}, \cdots \in \Gamma^*$ the monitor receives with $|w^{(i)}| = i + 1$ and uncertainty encoding ν

$$\forall i, j \in \mathbb{T}.\, j < i \rightarrow \{v(j) \mid v \in uconv_\nu(w^{(i)})\} = \{v(j) \mid v \in uconv_\nu(w^{(i-1)})\} \text{ and}$$

$$\forall i, j \in \mathbb{T}.\, j > i \rightarrow \{v(j) \mid v \in uconv_\nu(w^{(i)})\} = 2^{\mathbf{D}^{\mathbf{in}}_\varphi}.$$

I.e. when receiving an uncertain input trace of length $i + 1$ ($w^{(i)}$), all represented streams at instants before i are the same as for the last received input prefix of length i ($w^{(i-1)}$). Further the streams are fully unknown for instants after i. With these restrictions we effectively limit ourselves to timestamp-immanent uncertainty, where we can only express uncertainty within a specific instant.

Assumptions

For the handling of assumptions in LOLA the concept from [KLS22a, HKLS24] is followed. The LOLA specification is enriched by an intermediate stream Λ of type $\mathbb{B}$, which defines the assumption in terms of a LOLA expression. This stream Λ is assumed to be true at all instants. This means that all input stream tuples which would cause Λ to be false at some trace position are not possible and must not be taken into consideration as potential input streams.

For illustration consider again the example from figure 4.2. Imagine the assumption that the values on the input stream are increasing by at least one per instant. This could be formulated in terms of the following LOLA assumption to be added to the specification:

```
Λ := i[now] ≥ i[-1|-1] + 1
```

The stream Λ is only true if i surpasses its previous value by at least 1. Thereby the default value of the offset expression is chosen in a way that at the first position stream i is at least 0, i.e. any value of the stream's domain $\mathbb{N}$.

Note that for more complex assumptions it is of course also possible to add several intermediate streams to the specification which are then referenced by Λ.

Recall that LOLA is in principle capable of expressing any synchronous property (see theorem 3.12), thus for any set of possible input streams there is a LOLA specification with a stream Λ that is true at all instants if and only if the input streams are contained in this set. Consequently, every assumption can be encoded in a LOLA specification using the proposed approach and adding stream Λ together with utilized helper streams to the original specification.

Yet it is not enough to simply define such a stream Λ in a LOLA specification. During the monitoring process, it is necessary to take into account that the assumption stream cannot be false at any instant.

Therefore in definition 4.6 we slightly adjust the monitoring semantics from definition 4.5 to support assumptions. For a stream tuple $s \in \mathbf{S}_\varphi$ we use the notation $s(t, \Lambda)$ with $t \in \mathbb{T}$ to reference the value of stream Λ at instant t in s.

Definition 4.6 (Monitoring semantics under assumptions; based on [HKLS24])**.**
Let $\varphi = (I = \{s_1, \ldots, s_n\}, S = \{s_{n+1}, \ldots, s_{n+m}\}, O = \{s_{o_1}, \ldots, s_{o_l}\}, E)$ be a LOLA specification with assumption stream $\Lambda \in S$ and $\Sigma \in \mathcal{T}_{\mathbf{D}_\varphi^{\text{in}}}$ an input monitoring event stream tuple.

The *monitoring fixed point equation of φ under assumptions* is given as

$$\begin{array}{lcl} [\![\varphi]\!]_\Sigma^{mon,A} & : & \mathcal{T}_{\mathbf{D}_\varphi} \to \mathcal{T}_{\mathbf{D}_\varphi} \\ [\![\varphi]\!]_\Sigma^{mon,A}(T) & = & \{s = [\![\varphi]\!]^{fp}_{(\pi_1,\ldots,\pi_n)}(\pi_1, \ldots, \pi_{n+m}) \mid (\pi_1, \ldots, \pi_{n+m}) \in T, \\ & & \quad (\pi_1, \ldots, \pi_n) \in \Sigma, \forall t \in \mathbb{T}.\, s(t, \Lambda) = \texttt{true}\} \end{array}$$

The *monitoring semantics of φ under assumptions* is given by the function

$$\begin{array}{lcl} [\![\varphi]\!]^{mon,A} & : & \mathcal{T}_{\mathbf{D}_\varphi^{\text{in}}} \to \mathcal{T}_{\mathbf{D}_\varphi^{\text{out}}} \\ [\![\varphi]\!]^{mon,A}(\Sigma) & = & \{(\omega_{o_1}, \ldots, \omega_{o_l}) \mid (\omega_1, \ldots, \omega_{n+m}) \in \nu([\![\varphi]\!]_\Sigma^{mon,A})\} \end{array}$$

Throughout this chapter, we require the existence of the assumption stream Λ in every LOLA specification. Of course, if there are no assumptions, the stream can (implicitly) be defined to be constantly true. In this case, however, we refrain from explicitly denoting this stream in the examples.

A peculiarity about the proposed way of specifying assumptions is the following: Consider a past-only specification with assumption stream Λ that also uses only negative offsets. Since Λ is known to be true at all instants (including future ones), perfect recurrent monitoring requires reasoning about the future if additionally uncertainty is involved. This is because the assumption in the future could influence the possible values of the uncertain input in the presence.

As example consider the specification in figure 4.3. There are two input streams i, j of type $\mathbb{R}$ and an assumption without future offsets. The assumption enforces that

```
input i : ℝ
input j : ℝ

Λ := (i[now] = i[-1|0] + 1) ∧ ((i[now] ≥ 20) → (j[now] = 10)) ∧
     ((i[now] > 1) → (j[now] = j[-1|0]))
k := j[now] + 1

output k
```

Figure 4.3.: Example LOLA specification with past-only assumption that requires future reasoning if uncertainty is present.

the values of stream i are starting with 1 at instant 0 and are increasing by 1 per instant. Because of the second part of the assumption, this leads to the situation that if $t_{max} \geq 19$, every event of j from instant 19 on has the value 10. By the last part of the assumption however all events of j (except the first one) must be equal to their predecessor and thus all values of j are 10 if $t_{max} \geq 19$. In this case a perfect recurrent monitor for this specification that receives the input ? for j at instant 0 would have to return the value 11 for output stream k, which is defined as $j + 1$. This however requires reasoning about the future.

4.4. An abstraction-based recurrent LOLA monitoring framework

Based on the considerations about monitor inputs, uncertainties and assumptions in the previous section we now discuss a generic, abstraction-based recurrent LOLA monitoring framework. This will serve as the theoretical foundation for the symbolic approach presented afterwards.

4.4.1. Concrete recurrent LOLA monitoring

What we want to construct are sound or perfect recurrent LOLA monitors under uncertainties and assumptions for the induced pointwise property $\mathcal{P}_\varphi$ (definition 3.11) of a LOLA specification φ. A perfect monitor is one that iteratively yields all possible stream value combinations for the current instant that result from the LOLA monitoring semantics under assumptions (definition 4.6) for the input trace received so far. Further, the monitor is sound, if it delivers an over-approximation of the possible stream values at the current instant. This connection between a recurrent monitor as defined in definition 3.10 and the LOLA monitoring semantics is proved in the following lemma.

Lemma 4.7.
Let φ be a LOLA specification with assumption and ν an uncertainty encoding for uncertain input domain Γ.

Monitor M_φ is a sound recurrent monitor under uncertainty and assumption for $\mathcal{P}_\varphi$ with output interpretation $\gamma_{\mathbb{V}} : \mathbb{V} \to 2^{\mathbf{D}_\varphi^{\mathbf{out}}}$ if and only if

$$\forall w \in \Gamma^{\leq t_{max}+1}. \gamma_{\mathbb{V}}(M_\varphi(w)) \supseteq [\![\varphi]\!]^{mon,A}(uconv_\nu(w))(|w|-1)$$

and M_φ is a perfect recurrent monitor under uncertainty and assumption for $\mathcal{P}_\varphi$ if and only if

$$\forall w \in \Gamma^{\leq t_{max}+1}. \gamma_{\mathbb{V}}(M_\varphi(w)) = [\![\varphi]\!]^{mon,A}(uconv_\nu(w))(|w|-1)$$

Proof. We prove the soundness condition. For perfectness the $\supseteq$ in the proof can be replaced by equality. Let $A \subseteq (\mathbf{D}_\varphi^{\mathbf{in}})^*$ be the assumption encoded by Λ.

By definitions 3.10 and 3.11 and our requirement that uncertain inputs only encode certain traces of the same length we have

$$\forall w \in \Gamma^*.$$
$$\gamma_{\mathbb{V}}(M_\varphi(w)) \supseteq \{\mathcal{P}_\varphi(w'v, |w'|-1) \mid w' \in \nu(w), w'v \in A, \mathcal{P}_\varphi(w'v, |w'|-1) \text{ def.}\} \Leftrightarrow$$

$$\forall w \in \Gamma^{\leq t_{max}+1}.$$
$$\gamma_{\mathbb{V}}(M_\varphi(w)) \supseteq \{\mathcal{P}_\varphi(w'v, |w|-1) \mid w' \in \nu(w), w'v \in A, v \in (\mathbf{D}_\varphi^{\mathbf{in}^{t_{max}-|w|+1}})\} =$$

$$\{com([\![\varphi]\!](dec(w'v)))(|w|-1) \mid w' \in \nu(w), w'v \in A, v \in (\mathbf{D}_\varphi^{\mathbf{in}})^{t_{max}-|w|+1}\}.$$

From the fact that $[\![\varphi]\!]^{mon,A}$ filters out all streams from $[\![\varphi]\!]^{mon}$ which contain at least one Λ event which is false and lemma 4.4 which proofs that $[\![\varphi]\!]^{mon}$ and $[\![\varphi]\!](dec(w'v))$ deliver the same set of intermediate stream tuples we can further conclude that

$$\{com([\![\varphi]\!](dec(w'v)))(|w|-1) \mid w' \in \nu(w), w'v \in A, v \in (\mathbf{D}_\varphi^{\mathbf{in}})^{t_{max}-|w|+1}\} =$$

$$[\![\varphi]\!]^{mon,A}(uconv_\nu(w))(|w|-1). \qquad \square$$

The lemma motivates that we can build a recurrent LOLA monitor based on the LOLA monitoring semantics.

As discussed, a recurrent monitor only provides information about the output stream events at the current instant. Thus, for past-only specifications without uncertainty and assumptions, the monitor can compute outputs based only on the stream values from the previous instant and the current input readings (since the specification is flattened). Subsequently it may discard the stream values from the last step, which leads to a trace-length-independent, recurrent monitoring algorithm. Note that, his

is exactly what the traditional LOLA monitoring algorithm, but also similar past-only monitoring approaches, like the past LTL construction from Havelund and Roşu (see section 2.2.2) do.

However, if the specification involves future references, the monitor also has to reason about future events. Yet future events may themselves depend on events further in the future and so on, which at first glance requires "unrolling" of the specification all the way to the end of the trace. Reconsider for example the specification in figure 4.1. There stream s at instant 0 is dependent on stream s at instant 1, which is in turn dependent on stream s at instant 2 and so on. Unfortunately, full unrolling of the specification leads to performance problems for longer trace lengths.

The same holds, if the specification is past-only, but assumptions and uncertain input readings are present. In this case, the future also has to be considered, as the assumption in future states may have implications on the possible values of an uncertain event in the presence as outlined in section 4.3.2.

In the following we tackle the problem of evaluating LOLA specifications with future by introducing an abstraction of the monitoring semantics under assumption $[\![\varphi]\!]^{mon,A}$. The basic idea is to store the possible stream values at a certain instant (almost) independently from the values at other instants. We will then show how this abstract structure can be computed efficiently without unrolling the full trace (sometimes for the sake of over-approximation), and how to build a trace-length-independent recurrent monitor based on it. After the introduction of this generic abstraction-based theory, a symbolic implementation of the approach will be discussed in chapter 5.

For a LOLA specification φ the possible value combinations of all input and intermediate streams at a specific instant (which we call (stream) configurations) can be represented in the domain $2^{\mathbf{D}_\varphi}$.

> **Definition 4.8** (Stream configuration; based on [HKLS24])**.**
>
> Let $\varphi = (I, S, O, E)$ be a LOLA specification.
>
> A *(stream) configuration* of φ is an element $c \in \mathbf{D}_\varphi$. With $c(s)$ for $s \in I \cup S$ we denote the entry of stream s in stream configuration $c \in \mathbf{D}_\varphi$.
>
> Elements $s \in 2^{\mathbf{D}_\varphi}$ are called *(stream) configuration sets* or *(stream) configurations* of φ.

Consider for example a specification with a real-valued input stream and one output stream, where the output stream – also of type $\mathbb{R}$ – is defined as the input stream multiplied by three. If at some instant the input 3 is received the corresponding stream configurations would be given as $\{(3, 9)\}$. If the input value at this instant is unknown, the stream configurations would be $\{(v, 3 \cdot v) \mid v \in \mathbb{R}\}$.

In the semantics that we will define as the basis of our recurrent monitoring approach, we will compute an extended version of the stream configurations at each instant. We will determine the stream configurations *parametric in the configuration of the previous instant*. We call such parametric configurations *configuration transformers* as they can be used to transform one set of stream configurations to the set of stream configurations at the subsequent instant. A formal definition of them is given in definition 4.9.

> **Definition 4.9** (Configuration transformer; based on [HKLS24])**.**
>
> Let φ be a LOLA specification.
>
> A function $\tau \in T^{\varphi} := (\mathbf{D}_{\varphi} \to 2^{\mathbf{D}_{\varphi}})$ is called *(configuration) transformer* for φ.

A configuration transformer is a function assigning all possible stream configurations of the subsequent instant to the stream configuration of the current instant. The usage of configuration transformers instead of plain configuration sets will pay off later, as they allow for a highly efficient computation and storage of the recurrent monitoring semantics.

For illustration of configuration transformers consider the example LOLA specification in figure 4.4.

```
1    input i: ℬ
2
3    u := u[+1 | 0] + ite(i[now], 1, 0)
4    v := u[−1 | 0]
```

Figure 4.4.: Example LOLA specification

In this specification stream u (of type $\mathbb{N}$) sums up the number of instants where i is true until the end of the trace and v (also of type $\mathbb{N}$) takes the last value of u. Consider $\mathbb{T} = \{0, 1, \ldots, 99\}$. The configuration transformer for $t = 20$ without information about the inputs at this instant available would for example be given by the function

$$\begin{aligned}\tau((i^{19}, u^{19}, v^{19})) \quad = \quad & \{(\texttt{true}, u^{20}, u^{19}) \mid 1 \leq u^{20} \leq 80\} \cup \\ & \{(\texttt{false}, u^{20}, u^{19}) \mid 0 \leq u^{20} \leq 79\}\end{aligned}$$

where the tuples indicate the values of i, u and v in this order. The value of u^{20} cannot be greater than 80, since this is the maximum number of trace instants beyond and including 20 where 1 can be added to u. However if i^{20} is true u^{20} must at least be 1, otherwise it can at most be 79. The value of v is in both cases equal to u one instant before.

Configuration transformers build a complete lattice structure $(T^{\varphi}, \preccurlyeq)$ where $\tau_1 \preccurlyeq \tau_2$ holds for $\tau_1, \tau_2 \in T^{\varphi}$ if and only if $\forall v \in \mathbf{D}_{\varphi}. \tau_1(v) \subseteq \tau_2(v)$, i.e. $\preccurlyeq$ is the pointwise $\subseteq$

relation. Meet and join operation are also given as the pointwise application of $\cap$ and $\cup$ on all corresponding function values.

In the following we will define a LOLA semantics over a $(T^\varphi)^{|\mathbb{T}|}$ structure, where each tuple entry contains the configuration transformer for the corresponding instant.Note that this structure $(T^\varphi)^{|\mathbb{T}|}$ is also a complete lattice (see lemma 2.53) together with the order

$$(\tau_0, \ldots, \tau_{t_{max}}) \preceq (\tau'_0, \ldots, \tau'_{t_{max}}) \text{ iff } \forall t \in \mathbb{T}.\, \tau_t \preccurlyeq \tau'_t.$$

For definition of a transformer-based LOLA semantics we first require a definition how configuration transformers for a specific trace instant can be determined. Therefore we start with an adapted LOLA expression semantics. It provides us the value of a LOLA expression, given the configuration for the current, previous and subsequent instant (recall that $c(s)$ for $s \in I \cup S$ denotes the entry of stream s in stream configuration $c \in \mathbf{D}_\varphi$).

Definition 4.10 (Adapted LOLA expression semantics; based on [HKLS24])**.**

Let $\varphi = (I, S, O, E)$ be a LOLA specification. Let $b, a, c \in \mathbf{D}_\varphi$ be configurations over all streams before, after and at the current instant.

The value of a LOLA expression $e \in \mathit{Exp}_{\mathbb{D}}^{I \cup S}$ in the middle of the trace, denoted $[e]^\varphi_{b,c,a} \in \mathbb{D}$, is given as

- $[d]^\varphi_{b,c,a} = d$
- $[s[now]]^\varphi_{b,c,a} = c(s)$
- $[s[-1, d]]^\varphi_{b,c,a} = b(s)$
- $[s[1, d]]^\varphi_{b,c,a} = a(s)$
- $[f(e_1, \ldots, e_k)]^\varphi_{b,c,a} = f([e_1]^\varphi_{b,c,a}, \ldots, [e_k]^\varphi_{b,c,a})$
- $[\mathit{ite}(e_1, e_2, e_3)]^\varphi_{b,c,a} = \begin{cases} [e_2]^\varphi_{b,c,a} & \text{if } [e_1]^\varphi_{b,c,a} = \texttt{true} \\ [e_3]^\varphi_{b,c,a} & \text{else} \end{cases}$

for constant $d \in \mathbb{D}$, stream identifier $s \in I \cup S$ and LOLA sub-expressions $e_1, \ldots, e_k \in \mathit{Exp}^{I \cup S}$.

The value of $e \in \mathit{Exp}_{\mathbb{D}}^{I \cup S}$ at the beginning of the trace, $[e]^{\varphi,\triangleright}_{c,a} \in \mathbb{D}$, is given as

- $[s[-1, d]]^{\varphi,\triangleright}_{c,a} = d$
- analogous to $[e]^\varphi_{b,c,a}$ for other expressions

for stream identifier $s \in I \cup S$ and constant $d \in \mathbb{D}$.

The value of $e \in \mathit{Exp}_{\mathbb{D}}^{I \cup S}$ at the end of the trace, $[e]^{\varphi,\triangleleft}_{b,c} \in \mathbb{D}$, is given as

- $[s[1, d]]^{\varphi,\triangleleft}_{b,c} = d$
- analogous to $[e]^\varphi_{b,c,a}$ for other expressions

for stream identifier $s \in I \cup S$ and constant $d \in \mathbb{D}$.

For specification $\varphi = (I, S, O, E)$ with intermediate streams $s'_1, \ldots, s'_m$ we use the following shorthand syntax to apply the introduced semantics on the defining expressions of all intermediate streams:

$$\begin{array}{lcl} [E]^{\varphi,\triangleleft}_{b,c} & = & ([E(s'_1)]^{\varphi,\triangleleft}_{b,c}, \ldots, [E(s'_m)]^{\varphi,\triangleleft}_{b,c}) \\ [E]^{\varphi}_{b,c,a} & = & ([E(s'_1)]^{\varphi}_{b,c,a}, \ldots, [E(s'_m)]^{\varphi}_{b,c,a}) \\ [E]^{\varphi,\triangleright}_{c,a} & = & ([E(s'_1)]^{\varphi,\triangleright}_{c,a}, \ldots, [E(s'_m)]^{\varphi,\triangleright}_{c,a}) \end{array}$$

Further we drop the φ in the superscript if it is clear from the context.

Based on definition 4.10 we can now define the configuration transformers for each specific instant. Therefore we have to distinguish between the transformer at the trace end, which is constant and the transformers in the middle of the trace and at the beginning which are dependent on the transformer of the subsequent instant to resolve future offsets. To handle assumptions in the definition of the transformers, we also include only those configurations in the set of possible configurations where the entry of the assumption stream Λ is true.

Definition 4.11 (Transformer computation; based on [HKLS24])**.**

Let $\varphi = (I, S, O, E)$ be a LOLA specification with assumption stream $\Lambda \in S$ and an input monitoring stream tuple $\Sigma \in \mathcal{T}_{\mathbf{D}^{\mathbf{in}}_{\varphi}}$.

The transformer of φ and Σ for instant t_{max}, $\tau^{\varphi,t_{max}}_{\Sigma} \in T^{\varphi}$, is given as

$$\begin{array}{lcl} \tau^{\varphi,t_{max}}_{\Sigma} & : & \mathbf{D}_{\varphi} \to 2^{\mathbf{D}_{\varphi}} \\ \tau^{\varphi,t_{max}}_{\Sigma}(b) & = & \{c \mid c = i \circ [E]^{\triangleleft}_{b,c}, i \in \Sigma(t_{max}), c(\Lambda) = \texttt{true}\}. \end{array}$$

The transformer for instant $t \in \mathbb{T}$, $t < t_{max}$, $\tau^{\varphi,t}_{\Sigma}(\tau') \in T^{\varphi}$ in dependence of the subsequent transformer $\tau' \in T^{\varphi} = (\mathbf{D}_{\varphi} \to 2^{\mathbf{D}_{\varphi}})$ is given as

$$\tau^{\varphi,t}_{\Sigma} \quad : \quad (\mathbf{D}_{\varphi} \to 2^{\mathbf{D}_{\varphi}}) \to (\mathbf{D}_{\varphi} \to 2^{\mathbf{D}_{\varphi}})$$

$$\tau^{\varphi,t'}_{\Sigma}(\tau')(b) \quad = \quad \{c \mid c = i \circ [E]_{b,c,a}, i \in \Sigma(t'), a \in \tau'(c), c(\Lambda) = \texttt{true}\}$$

for $t' > 0$ and

$$\tau^{\varphi,0}_{\Sigma}(\tau')(b) \quad = \quad \{c \mid c = i \circ [E]^{\triangleright}_{c,a}, i \in \Sigma(0), a \in \tau'(c), c(\Lambda) = \texttt{true}\}.$$

Thus, the transformers simply result from application of the adapted LOLA expression semantics on the defining expressions of all streams. For predecessor configuration b they yield the set of all configurations c, which are a concatenation of inputs at this instant and the corresponding expression semantics for c itself, b and the configurations of the subsequent instant a. This definition is recursive in the

sense that the computation of a via the subsequent transformer and the expression semantics relies on c. Hence, c is dependent on itself. In the set of solutions we include all vectors c which satisfy the given equality, thus define the maximal set of such vectors. However, since a well-defined LOLA specification cannot define events that depend on themselves, also no entry in c can depend on itself. Consequently, c and a can be computed deterministically, stream by stream alternately. Finally, configurations where the assumption is broken are not considered valid and excluded from the set of possible configurations.

Reconsider again the example specification φ from figure 4.4 for $\mathbb{T} = \{0, 1, 2, 3\}$ and the input trace $\langle \texttt{true}, \texttt{true} \rangle$ received so far, corresponding to the monitoring stream tuple

$$\Sigma = \{(\langle \texttt{true}, \texttt{true}, b_2, b_3 \rangle) \mid b_2, b_3 \in \mathbb{B}\}$$

For this specification the configuration transformer for $t_{max} = 3$ is given as

$$\tau^3((i^2, u^2, v^2)) = \tau_\Sigma^{\varphi,3}((i^2, u^2, v^2)) = \{(\texttt{true}, 1, u^2), (\texttt{false}, 0, u^2)\}$$

where the tuples contain the values of streams i, u and v in that order. It expresses the situation that at instant 3, stream u is 1 if stream i is true and 0 otherwise.

The transformer for instant $t = 2$ can by definition 4.11 be determined as

$$\begin{aligned}\tau^2((i^1, u^1, v^1)) = \tau_\Sigma^{\varphi,2}(\tau^3)((i^1, u^1, v^1)) = &\{c \mid c = (\texttt{true}, a(u) + 1, u^1), a \in \tau^3(c)\} \cup \\ &\{c \mid c = (\texttt{false}, a(u), u^1), a \in \tau^3(c)\}.\end{aligned}$$

For any a in $\tau^3(c)$, the second component $(a(u))$ representing stream u is 0 or 1 independent of the concrete c. Thus we have

$$\tau^2((i^1, u^1, v^1)) = \{(\texttt{false}, 0, u^1), (\texttt{false}, 1, u^1), (\texttt{true}, 1, u^1), (\texttt{true}, 2, u^1)\}.$$

This reflects exactly the possible stream configurations at instant 2, parametric in the configuration of the predecessor instant. The transformer for instant $t = 1$ would likewise be given as

$$\tau^1((i^0, u^0, v^0)) = \tau_\Sigma^{\varphi,1}(\tau^2)((i^0, u^0, v^0)) = \{c \mid c = (\texttt{true}, a(u) + 1, u^0), a \in \tau^2(c)\}.$$

Again for all $a \in \tau^2(c)$, the second component is 0, 1 or 2, independent of c, and thus:

$$\tau^1((i^0, u^0, v^0)) = \tau_\Sigma^{\varphi,1}(\tau^2)((i^0, u^0, v^0)) = \{(\texttt{true}, 1, u^0), (\texttt{true}, 2, u^0), (\texttt{true}, 3, u^0)\}$$

As before, these are exactly the potential stream values at instant 1 in dependence of those at instant 0. The transformer τ^0 can be determined analogously.

Based on definition 4.11 we can now define a LOLA semantics which yields an element from $(T^\varphi)^{|\mathbb{T}|}$, referred to as *transformer structure*, containing the configuration transformer for each instant. The definition is again given by means of a fixed point equation. The transformer at each instant depends solely on the transformer of the subsequent instant in the transformer structure.

Definition 4.12 (Transformer semantics; based on [HKLS24])**.**

Let φ be a LOLA specification and $\Sigma \in \mathcal{T}_{\mathbf{D}_\varphi^{\mathbf{in}}}$ a corresponding input monitoring stream tuple.

The *transformer fixed point equation of* φ is defined as:

$$\begin{array}{lcl} [\![\varphi]\!]^{tra}_{\Sigma} & : & (T^\varphi)^{|\mathbb{T}|} \to (T^\varphi)^{|\mathbb{T}|} \\ [\![\varphi]\!]^{tra}_{\Sigma}(T) & = & (\tau^{\varphi,0}_{\Sigma}(T(1)), \tau^{\varphi,1}_{\Sigma}(T(2)), \ldots, \tau^{\varphi,t_{max}}_{\Sigma}) \end{array}$$

The *transformer semantics of* φ is given as $\mu([\![\varphi]\!]^{tra}_{\Sigma})$.

It is easy to see that the fixed point of this fixed point equation is unique (i.e. $\mu([\![\varphi]\!]^{tra}_{\Sigma}) = \nu([\![\varphi]\!]^{tra}_{\Sigma})$) as it can deterministically be computed from back to front. Note that this is only possible due to the usage of transformers instead of configurations in the structure, as they are decoupled from the predecessor values, one of the causes why this structure was chosen.

Indeed the entries of the LOLA transformer semantics are compliant with the LOLA monitoring semantics $[\![\varphi]\!]^{mon,A}(\Sigma)$ from definition 4.6. The transformers in there match exactly with the configurations of two subsequent instants that appear in a concrete stream from the monitoring semantics.

We can thus consider the transformer semantics as an abstraction of the monitoring semantics in the transformer structure $(T^\varphi)^{|\mathbb{T}|}$. In the transformer structure domain, the information about the relationships between the stream events is abstracted away, and only the relations among events of the same instant and to the instant before are preserved. Thus, the transformer semantics looses information among the interconnection of events at non-consecutive locations, but it is still the perfect result in the transformer structure domain (with the auxiliary condition that we consider only those mappings of the transformers which can actually be triggered with the values from the previous transformer). This relation is visualized in figure 4.5. There

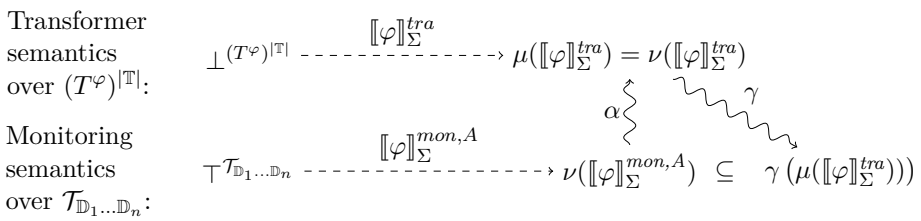


Figure 4.5.: Monitoring semantics abstraction by transformer semantics. The result of the fixed point iteration for the transformer semantics overapproximates the result of the upper fixed point iteration for the monitoring semantics. Yet the abstraction of the monitoring semantics corresponds to the transformer semantics.

$\alpha : \mathcal{T}_{\mathbf{D}_\varphi} \to (T^\varphi)^{|\mathbb{T}|}$, $\gamma : (T^\varphi)^{|\mathbb{T}|} \to \mathcal{T}_{\mathbf{D}_\varphi}$ denote the translation functions between both domains, where α translates a monitoring stream tuple to the transformer structure that represents all transitions that appear in one of the streams. Further γ translates a transformer structure into the set of all streams which are compatible with the transformers. In fact the transformer structure is (though it looses information w.r.t. the monitoring semantics) sufficient for recurrent monitoring, where we only care about the events at the current instant. This is proved in theorem 4.13.

Theorem 4.13.
Let $\varphi = (I, S, O, E)$ be a LOLA specification with assumption stream $\Lambda \in S$ and $\Sigma \in \mathcal{T}_{\mathbf{D}_\varphi^{\mathbf{in}}}$ an input monitoring stream tuple with instant-immanent uncertainty.

For all $t \in \mathbb{T}$ with $t > 0$ and stream configurations $v \in \nu([\![\varphi]\!]_\Sigma^{mon,A})(t-1)$ it holds that

$$\mu([\![\varphi]\!]_\Sigma^{tra})(t)(v) = \{s(t) \mid s \in \nu([\![\varphi]\!]_\Sigma^{mon,A}), s(t-1) = v\}$$

and for any $v \in \mathbf{D}_\varphi$

$$\mu([\![\varphi]\!]_\Sigma^{tra})(0)(v) = \nu([\![\varphi]\!]_\Sigma^{mon,A})(0).$$

Proof. We directly prove by induction over both structures from t_{max} to 1.

We start with $t = t_{max}$ and first show that $\mu([\![\varphi]\!]_\Sigma^{tra})(t_{max})(b) = \nu([\![\varphi]\!]_\Sigma^{tra})(t_{max})(b) \subseteq \{s(t_{max}) \mid s \in \nu([\![\varphi]\!]_\Sigma^{mon,A}), s(t_{max}-1) = b\}$ for $\Sigma \in \mathcal{T}_{\mathbf{D}_\varphi^{\mathbf{in}}}$ and an arbitrary $b \in \nu([\![\varphi]\!]_\Sigma^{mon,A})(t_{max}-1)$.

Therefore assume $c \in \mu([\![\varphi]\!]_\Sigma^{tra})(t_{max})(b)$. By definition 4.11, we have $c = i \circ [E]_{b,c}^{\triangleleft}$ with $i \in \Sigma(t_{max})$. Further the Λ entry in c is true. Note that there must be a stream $s \in \nu([\![\varphi]\!]_\Sigma^{mon,A})$ s.t. $s(t_{max}-1) = b$. Since we demand instant-immanent uncertainty for Σ we can find a stream where in addition to this condition $s(t_{max})$ contains the values from i for the input streams. Due to the equality that c satisfies, we thus have an $s \in \nu([\![\varphi]\!]_\Sigma^{mon,A})$ with $s(t_{max}-1) = b$ and $s(t_{max}) = c$ (because in $\nu([\![\varphi]\!]_\Sigma^{mon,A})$ all equations of φ are satisfied as by definition of $[E]_{b,c}^{\triangleleft}$). Consequently $c \in \{s(t_{max}) \mid s \in \nu([\![\varphi]\!]_\Sigma^{mon,A}), s(t_{max}-1) = b\}$.

Now we show the opposite direction. Assume $c \in \{s(t_{max}) \mid s \in \nu([\![\varphi]\!]_\Sigma^{mon,A}), s(t_{max}-1) = b\}$. Thus, there is a stream $s \in \nu([\![\varphi]\!]_\Sigma^{mon,A})$, s.t. $s(t_{max}) = c$ and $s(t_{max}-1) = b$. Further there is an $i \in \Sigma(t_{max})$ s.t. $s(t_{max}) = c$ contains the entries from i as inputs. Since s is from $\nu([\![\varphi]\!]_\Sigma^{mon,A})$ all equalities of φ are satisfied for s. From the definition of $[E]_{b,c}^{\triangleleft}$ it follows that $c = i \circ [E]_{b,c}^{\triangleleft}$ also holds. Further the Λ entry in $s(t_{max})$ must be true. Consequently $c \in \mu([\![\varphi]\!]_\Sigma^{tra})(t_{max})(b)$.

Provided the proposition holds for $t \in \mathbb{T}$ with $t > 1$, we can use the same idea to show the equality for $t-1$. In difference to above, however, the adapted expression semantics for the middle of the trace, $[E]_{b,c,a}$, which also depends on a successor configuration a has to be used.

For the fist direction we again have to prove that there is an $s \in \nu([\![\varphi]\!]_\Sigma^{mon,A})$ s.t. $s(t-1) = b$ and $s(t) = c = i \circ [E]_{b,c,a}$ with its Λ entry true for some $b \in \nu([\![\varphi]\!]_\Sigma^{mon,A})(t-1)$, $a \in \mu([\![\varphi]\!]_\Sigma^{tra})(t+1)(c)$ and $i \in \Sigma(t)$. We will show, that there is a stream s with $b = s(t-1)$, $c = s(t)$ and $a = s(t+1)$ which is compatible with an input stream from Σ and φ. Since $\nu([\![\varphi]\!]_\Sigma^{mon,A})$ is the greatest fixed point of the monitoring fixed point equation we can then also conclude $s \in \nu([\![\varphi]\!]_\Sigma^{mon,A})$. Consider all intermediate stream events in c and a. Since our LOLA specification is well-formed, every of these events has a finite chain of other input and intermediate stream events it depends on. There may not be a cyclic dependency. Thus for every event in c and a we can set up a term over input events from instants $t-1$ to t_{max} and intermediate events from instant $t-1$ which determines the value of these events. Such a term can be retrieved by unrolling of the specification up to $t-1$ in the one direction and t_{max} in the other. Based on these terms we can then form equations for each event in a and c. We can follow from definitions 4.11 and 4.12 that there must be an input vector from $\Sigma(t')$ for each $t' \in \mathbb{T}$ with $t' > t-1$ s.t. the elements in b, c, a satisfy these equations, otherwise they could not have been included in the corresponding transformers, as the elements there preserve the equations from the LOLA specification. Those input vectors from $t-1, t, t+1$ can further be chosen to fit the inputs in b, c and a. Due to the instant-immanent uncertainty we demanded for Σ and the fact that $b \in \nu([\![\varphi]\!]_\Sigma^{mon,A})(t-1)$ (i.e. all event values in b are by definition compatible with the specification and some prefix of the input streams), we can further conclude that there is also a single $\sigma \in \Sigma$ which contains an input prefix that leads to b and all the subsequent input vectors. This completes the proof of the first direction.

For the second direction we can use exactly the same argumentation as for $t = t_{max}$ above and simply choose the successor state for $[E]_{b,c,a}$ as $a = s(t+1)$. The fact that $a \in \mu([\![\varphi]\!]_\Sigma^{tra})(t+1)(c)$ follows directly from the induction hypothesis Thus we have proved the theorem for all $t \in \mathbb{T}$ with $t > 1$.

The proof for $t = 0$ is analogous, but for $[E]^{\triangleright}_{c,a}$ instead of $[E]_{b,c,a}$, where the transformer parameter b is not used. □

The theorem together with lemma 4.7 justifies the use of the LOLA transformer semantics for a recurrent monitoring algorithm. In the following, the basic idea behind such a monitor for a LOLA specification φ is sketched.

Consider a sequence of (potentially uncertain) monitor inputs $w^{(0)}, w^{(1)}, \ldots, w^{(t_{max})} \in \Gamma^*$ over uncertain input domain Γ and corresponding uncertainty encoding ν. Let

$\Sigma^0 = uconv_\nu(w^{(0)})$, $\Sigma^1 = uconv_\nu(w^{(1)})$, ... $\Sigma^{t_{max}} = uconv_\nu(w^{(t_{max})}) \in \mathcal{T}_{\mathbf{D}_\varphi^{\mathbf{in}}}$ be the correlating monitoring stream tuples.

As mentioned before (see section 4.3.2), we require that the streams represented by Σ^i are fully unknown for instants greater than i, and all stream prefixes up to position i excludingly were already represented by Σ^{i-1}. We have this restriction to guarantee an efficient computation of subsequent transformer semantics. If the monitoring streams could alter at arbitrary positions, this would make a full recomputation of the whole semantics necessary for every new received input.

The monitoring strategy is as follows:

1. First, the monitor computes the *initial transformer semantics* $\mu([\![\varphi]\!]^{tra}_{\top^{\text{mon}}})$ for the empty input trace (encoded by monitoring stream tuple $\top^{\text{mon}}$), i.e. where no input readings are known. This can be done before the actual monitoring starts.

2. Then it reads the input for the current instant t and recomputes transformer $\mu([\![\varphi]\!]^{tra}_{\Sigma^t})(t)$. Note that this transformer only depends on $\mu([\![\varphi]\!]^{tra}_{\Sigma^t})(t+1)$. Since Σ^t may not contain information about instant variables after t and $\mu([\![\varphi]\!]^{tra}_{\Sigma^t})(t+1)$ is due to the chosen transformer structure only dependent on the inputs at instant $t+1$ and greater, we have $\mu([\![\varphi]\!]^{tra}_{\Sigma^t})(t+1) = \mu([\![\varphi]\!]^{tra}_{\top^{\text{mon}}})(t+1)$. *Thus, it can be taken from the initial transformer semantics and only a single transformer has to be recomputed.*

3. Finally the monitor determines the current monitor state $s^t = \mu([\![\varphi]\!]^{tra}_{\Sigma^t})(t)(s^{t-1})$ and repeat from 2 for the subsequent instant unless $t = t_{max}$.

The monitor state s^t, which is a set of stream configurations, can finally be used as monitor output as it reveals possible stream configurations at the current instant. The initial state s^{-1} can be set to any non-empty configuration set, as the transformer for instant 0 is invariant in the concrete predecessor value.

4.4.2. Abstract recurrent LOLA monitoring

As the T^φ elements are usually hard to represent in memory and to compute on, it seems natural to use further abstractions to compute the transformer semantics. The idea is to introduce abstract domains for transformers and configurations and an abstraction of the transformer semantics in these domains. Therefore we introduce the notion of a perfect transformer abstraction, which allows for a lossless translation back to the concrete transformer, and a sound transformer abstraction, which causes an over-approximation of the transformer semantics. The abstracted semantics can then be used in the monitoring algorithm to substitute the computation in the concrete transformer domain. A visualization of the relation between abstract and concrete transformer semantics can be found in figure 4.6.

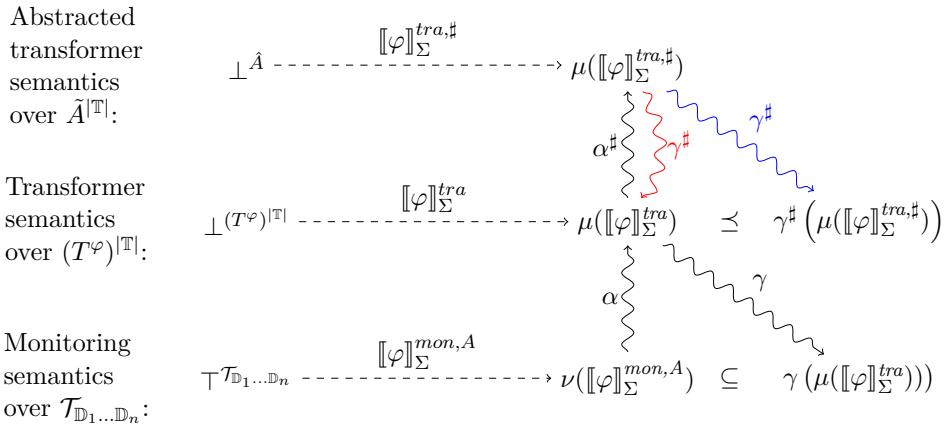


Figure 4.6.: Perfect abstract transformer semantics iteration $[\![\varphi]\!]_\Sigma^{tra,\sharp}$ in case of a sound (blue) and perfect (red) transformer abstraction. While the transformer semantics always over-approximates the monitoring semantic (but preserves the relation between current and previous events), the abstracted transformer semantics may further over-approximate the transformer semantics, but may also yield a perfect representation of it. The latter requires the transformer abstraction to be perfect and the abstracted semantics to be a perfect abstraction.

The definition of a sound and perfect transformer abstraction is given in definition 4.14. To be able to apply the result about abstract interpretation from section 2.4.2 we demand the abstract structure to be a complete lattice and the translations functions to form a Galois connection.

Definition 4.14 (Configuration transformer abstraction; based on [HKLS24])**.**

Let φ be a LOLA specification.

A *(configuration) transformer abstraction* for φ is a complete lattice $(\tilde{A}, \sqsubseteq^{\tilde{A}})$ together with a Galois connection

$$T^\varphi \underset{\alpha^{\tilde{A}}}{\overset{\gamma^{\tilde{A}}}{\leftrightarrows}} \tilde{A}.$$

It is called *perfect (configuration) transformer abstraction* for φ, if and only if

$$\forall \tau \in T^\varphi.\, \tau = \gamma^{\tilde{A}}(\alpha^{\tilde{A}}(\tau)).$$

An analogous notion to abstract transformers is also introduced for configuration sets in definition 4.15.

Definition 4.15 (Configuration set abstraction; based on [HKLS24])**.**

Let φ be a LOLA specification.

A *configuration set abstraction* for φ is a complete lattice $(A, \sqsubseteq^A)$ together with a Galois connection

$$2^{\mathbf{D}_\varphi} \underset{\alpha^A}{\overset{\gamma^A}{\leftrightarrows}} A.$$

It is called *perfect configuration set abstraction* for φ, if and only if

$$\forall c \in 2^{\mathbf{D}_\varphi}.\, c = \gamma^A(\alpha^A(c)).$$

As example consider again the configuration transformer

$$\tau^2((i^1, u^1, v^1)) = \{(\texttt{false}, 0, u^1), (\texttt{false}, 1, u^1), (\texttt{true}, 1, u^1), (\texttt{true}, 2, u^1)\}$$

for the specification from figure 4.4.

In the symbolic abstract domain, which we will discuss in detail in the subsequent chapter, we can encode this transformer as set of symbolic constraints:

$$\{v^2 = u^1, \neg i^2 \rightarrow (0 \leq u^2 \leq 1), i^2 \rightarrow (1 \leq u^2 \leq 2)\}.$$

This domain is indeed a perfect abstraction and preserves all relations between the events of instants 2 and 1.

Another thinkable – but imperfect – abstraction would be to encode the transformer as tuple with one semi-symbolic field per stream. Such semi-symbolic entries are either sets or intervals of possible values or instant variables from a previous stream, e.g.

$$(\{\texttt{true}, \texttt{false}\}, [0, 2], u^1).$$

This abstraction still preserves the relation between v and u at the previous instant, as well as the ranges of the single event values, but looses the interconnection among them.

Finally, it is also a valid but imperfect abstraction to store only an interval or set of possible values per stream:

$$(\{\texttt{true}, \texttt{false}\}, [0, 2], [-\infty, \infty])$$

This way all relations between the stream events and the events at the previous instant are lost.

We will not make use of the latter two abstract domains in this thesis, they were only intended to demonstrate the variety of such abstract transformer domains. The symbolic domain however will be discussed in detail and an implementation of it will be evaluated in the subsequent chapters.

In the abstract setting we will use the domain $\hat{A} = \tilde{A}^{|\mathbb{T}|}$ to represent the abstracted transformers for the whole trace. Like $(T^\varphi)^{|\mathbb{T}|}$, $\hat{A}$ also forms a complete lattice with order $\sqsubseteq^{\hat{A}}$ which results from a pointwise application of $\sqsubseteq^{\tilde{A}}$ on the elements. Furthermore a Galois connection $(T^\varphi)^{|\mathbb{T}|} \underset{\gamma^\sharp}{\overset{\alpha^\sharp}{\leftrightarrows}} \hat{A}$ between both domains, which also consists in pointwise application of the functions $\alpha^{\tilde{A}}, \gamma^{\tilde{A}}$, can be defined (see lemma 2.53).

From principles of abstract interpretation (see theorem 2.55) we can derive that if we have a sound, i.e. over-approximating, abstract semantics $[\![\varphi]\!]_\Sigma^{tra,\sharp}$, which operates on $\hat{A}$, we can compute an over-approximation of the concrete transformer semantics in this domain, that is $\alpha^\sharp(\mu([\![\varphi]\!]_\Sigma^{tra}))$. The analogous result holds for the perfect case. This finding is formulated in corollary 4.16.

Corollary 4.16.

Let φ be a LOLA specification, $\Sigma \in \mathcal{T}_{\mathbf{D}_\varphi^{\mathbf{in}}}$ an input monitoring stream tuple and $[\![\varphi]\!]_\Sigma^{tra,\sharp}$ an abstraction (definition 2.54) of $[\![\varphi]\!]_\Sigma^{tra}$.

It holds that

- $\alpha^\sharp(\mu([\![\varphi]\!]_\Sigma^{tra})) \sqsubseteq^{\hat{A}} \mu([\![\varphi]\!]_\Sigma^{tra,\sharp})$ and
- $\alpha^\sharp(\mu([\![\varphi]\!]_\Sigma^{tra})) = \mu([\![\varphi]\!]_\Sigma^{tra,\sharp})$

  if for all $c \in (T^\varphi)^{|\mathbb{T}|}$: $\alpha^\sharp([\![\varphi]\!]_\Sigma^{tra}(c)) = [\![\varphi]\!]_\Sigma^{tra,\sharp}(\alpha^\sharp(c))$

The question remains how to construct such a sound or perfect abstract transformer semantics. In definition 4.11 we defined the computation of the individual concrete transformers on basis of a LOLA specification and later built up the transformer semantics from them (see definition 4.12). For the abstract semantics it seems advantageous to stick to this kind of construction and build the abstract transformer semantics on abstractions of the single transformer elements. Lemma 4.17 prepares the necessary theoretical foundation for this. It states that if all utilized abstract transformers ($\tau_\Sigma^{t,\sharp}$) are sound approximations or perfect representations of the concrete ones ($\tau_\Sigma^{\varphi,t}$), then the resulting abstract transformer semantics is also a sound or perfect abstraction.

Note that, as in the concrete case before, the abstract transformers $\tau_\Sigma^{t,\sharp}$ which are not at the trace end depend on their successor transformers. Without concrete instantiation with their successor, we call these "transformer templates" of type $\tilde{A} \to \tilde{A}$ or $T^\varphi \to T^\varphi$ *(abstract) dependent transformers*. We use the notations $\tau \precsim \tau^\sharp$ or $\tau \preceq \tau^\sharp$ to denote that an abstract dependent transformer $\tau^\sharp : \tilde{A} \to \tilde{A}$ is a sound or perfect abstraction of a concrete one $\tau : T^\varphi \to T^\varphi$. Therefore we require $\tau^\sharp$ to be a (perfect) abstraction (c.f. definition 2.54 and theorem 2.55), instantiated for any successor transformer:

$$\tau \precsim \tau^\sharp \quad \text{iff} \quad \forall a \in \tilde{A}.\, \tau(\gamma^{\tilde{A}}(a)) \preccurlyeq \gamma^{\tilde{A}}(\tau^\sharp(a)) \text{ and}$$
$$\tau \preceq \tau^\sharp \quad \text{iff} \quad \forall c \in T^\varphi.\, \alpha^{\tilde{A}}(\tau(c)) = \tau^\sharp(\alpha^{\tilde{A}}(c)).$$

Note that by the Galois condition (definition 2.52) the first relation can also be defined as $\forall a \in \tilde{A}.\, \alpha^{\tilde{A}}(\tau(\gamma^{\tilde{A}}(a))) \sqsubseteq^{\tilde{A}} \tau^\sharp(a)$ and thus $\tau^\sharp \preceq \tau$ implies $\tau^\sharp \precsim \tau$ for monotonic $\tau^\sharp$. Based on this notation we can now phrase lemma 4.17.

Lemma 4.17.
Let φ be a LOLA specification and $\Sigma \in \mathcal{T}_{\mathbf{D}_\varphi^{\text{in}}}$ an input monitoring stream tuple.

Let $[\![\varphi]\!]_\Sigma^{tra,\sharp} : \hat{A} \to \hat{A}$ be an abstract transformer semantics s.t.

$$[\![\varphi]\!]_\Sigma^{tra,\sharp}(a) = (\tau_\Sigma^{0,\sharp}(a(1)), \tau_\Sigma^{1,\sharp}(a(2)), \ldots, \tau_\Sigma^{t_{max},\sharp})$$

with $\tau_\Sigma^{t_{max},\sharp} \in \tilde{A}$, $\tau_\Sigma^{t,\sharp} \in \tilde{A} \to \tilde{A}$ for $t \in \mathbb{T}\backslash\{t_{max}\}$ where all dependent transformers $\tau_\Sigma^{t,\sharp} \in \tilde{A} \to \tilde{A}$ are monotonic.

It holds that

- $\alpha^\sharp(\mu([\![\varphi]\!]_\Sigma^{tra})) \sqsubseteq^{\hat{A}} \mu([\![\varphi]\!]_\Sigma^{tra,\sharp})$ if

  $\tau_\Sigma^{\varphi,t_{max}} \preccurlyeq \gamma^\sharp(\tau_\Sigma^{t_{max},\sharp})$ and $\forall t \in \mathbb{T}\backslash\{t_{max}\}.\, \tau_\Sigma^{\varphi,t} \precsim \tau_\Sigma^{t,\sharp}$

- $\alpha^\sharp(\mu([\![\varphi]\!]_\Sigma^{tra})) = \mu([\![\varphi]\!]_\Sigma^{tra,\sharp})$ if

  $\alpha^\sharp(\tau_\Sigma^{\varphi,t_{max}}) = \tau_\Sigma^{t_{max},\sharp}$ and $\forall t \in \mathbb{T}\backslash\{t_{max}\}.\, \tau_\Sigma^{\varphi,t} \preceq \tau_\Sigma^{t,\sharp}$.

Proof. The abstract transformer semantics is – as the concrete transformer semantics – a composition of transformer computations for every instant. Since we have $\tau_\Sigma^{\varphi,t} \precsim \tau_\Sigma^{t,\sharp}$ and $\tau_\Sigma^{\varphi,t_{max}} \preccurlyeq \gamma^\sharp(\tau_\Sigma^{t_{max},\sharp})$ for all components, by definition of $\preceq$ we also have for the complete abstract transformer structure: $[\![\varphi]\!]_\Sigma^{tra}(\gamma^\sharp(T)) \preceq \gamma^\sharp([\![\varphi]\!]_\Sigma^{tra,\sharp}(T))$ for all $T \in \hat{A}$. As all dependent transformers are monotonic, again by definition of $\preceq$, also $[\![\varphi]\!]_\Sigma^{tra,\sharp}$ is monotonic and thus an abstraction of $[\![\varphi]\!]_\Sigma^{tra}$.

For the perfect abstraction we have $\tau_\Sigma^{\varphi,t} \preceq \tau_\Sigma^{t,\sharp}$ and $\alpha^\sharp(\tau_\Sigma^{\varphi,t_{max}}) = \tau_\Sigma^{t_{max},\sharp}$ for all components, which also implies that $[\![\varphi]\!]_\Sigma^{tra,\sharp}$ is an abstraction and by definition of $\sqsubseteq^{\hat{A}}$ also that $\alpha^\sharp([\![\varphi]\!]_\Sigma^{tra}(T)) = [\![\varphi]\!]_\Sigma^{tra,\sharp}(\alpha^\sharp(T))$ for all $T \in (T^\varphi)^{|\mathbb{T}|}$.

The lemma then follows directly from corollary 4.16, which builds on principles of abstract interpretation. □

In the subsequent chapter, a realization of the individual $\tau_\Sigma^{t,\sharp}$ in the mentioned symbolic domain will be presented, which adheres to the conditions of lemma 4.17.

Note that, like the concrete transformer semantics, the abstract semantics can also be determined deterministically from back to front, and thus has a unique fixed point. Therefore in the following we use $\mu([\![\varphi]\!]_\Sigma^{tra,\sharp})$ to denote it.

However, before we can continue with a monitoring algorithm based on the abstract transformer semantics we first discuss how the semantics can efficiently be computed.

Efficient computation of the initial abstract transformer semantics

In order to guarantee an efficient computation of the abstract transformer semantics, there are two more assumptions about the abstract transformers that we have to make for the remainder of this thesis. First, the definition of $\tau_\Sigma^{t,\sharp}$ must exclusively depend on the events of Σ at instant t. It may not depend on inputs at other instants. Second, we require that all dependent transformers $\tau_\Sigma^{t,\sharp}$ for the different $t \in \{1, \ldots, t_{max} - 1\}$ are defined equally. I.e. that they are identical, if the input streams at these instants are. This assumption is reasonable as there is little reason to use different techniques to determine the individual abstract transformers, which are neither at the beginning nor at the end of the trace. We can phrase these two requirements in a single formal condition:

$$\forall \Sigma, \Sigma' \in \mathcal{T}_{\mathbf{D}_\varphi^{\mathbf{in}}}. \forall t, t' \in \mathbb{T} \backslash \{0, t_{max}\}. (\Sigma(t) = \Sigma(t')) \rightarrow (\tau_\Sigma^{t,\sharp} = \tau_\Sigma^{t',\sharp})$$

Based on this, an important observation can be made regarding the efficient computation of the transformer semantics. Note that (abstract) transformers are - in difference to plain stream configurations at some instant - only dependent on the transformer of the subsequent instant, not the one from the previous instant. This is a feature of the special transformer structure we have chosen for our semantics; a transformer is parametric in the value of the events at the previous instant, and thus agnostic about them. Consequently, if in the abstract transformer semantics the entries at instants $t, t' > 1$ are equal and the way the dependent transformers for instants $t - 1$ and $t' - 1$ are defined equally (which we require with the previous assumption), then the abstract transformers for $t - 1$ and $t' - 1$ are also the same. This insight is formulated in lemma 4.18.

Lemma 4.18 (Based on [HKLS24])**.**
Let φ be a LOLA specification, $\Sigma \in \mathcal{T}_{\mathbf{D}_\varphi^{\mathbf{in}}}$ an input monitoring stream tuple and $t, t' \in \mathbb{T}$ with $0 < t < t'$ two instants.

If $\tau_\Sigma^{t-1,\sharp} = \tau_\Sigma^{t'-1,\sharp}$, then

$$\mu([\![\varphi]\!]_\Sigma^{tra,\sharp})(t) = \mu([\![\varphi]\!]_\Sigma^{tra,\sharp})(t') \text{ implies } \mu([\![\varphi]\!]_\Sigma^{tra,\sharp})(t-1) = \mu([\![\varphi]\!]_\Sigma^{tra,\sharp})(t'-1).$$

Proof. $\mu([\![\varphi]\!]^{tra,\sharp}_{\Sigma})(t-1) = \tau^{t-1,\sharp}_{\Sigma}(\mu([\![\varphi]\!]^{tra,\sharp}_{\Sigma})(t)) = \tau^{t'-1,\sharp}_{\Sigma}(\mu([\![\varphi]\!]^{tra,\sharp}_{\Sigma})(t')) = \mu([\![\varphi]\!]^{tra,\sharp}_{\Sigma})(t'-1)$. □

This lemma has significant impact on the computation of the abstract initial transformer semantics $[\![\varphi]\!]^{tra,\sharp}_{\top mon}$, i.e. the one for completely unknown input streams. With the assumption from above, we have that, if the input streams are fully unknown (and thus equal for all instants), then $\tau^{t,\sharp}_{\top mon} = \tau^{t',\sharp}_{\top mon}$ for all $t, t' \in \mathbb{T}\backslash\{0, t_{max}\}$. This means, if we compute the abstract initial transformer semantics from back to front as lemma 4.17 suggests and we recognize that one element in the semantic repeats, *we can directly get the whole semantics (except for the first instant) without further computations.*

This is because of the following: If the transformers for $t, t' \in \mathbb{T}$ are equal, then by lemma 4.18 also those for $t-1$ and $t'-1$ and - by the same argument - those for $t-2$ and $t'-2$ are equal and so on, as long as they are greater than instant 0. I.e., when computing the initial abstract transformer semantics from backward, the whole structure (except the first entry) is determined as soon as one transformer element repeats. Algorithmically the procedure for this accelerated computation of $[\![\varphi]\!]^{tra,\sharp}_{\top mon}$ is represented in figure 4.7.

```
1 S(t_max) ← τ^{t_max,♯}_{⊤mon};
2 for t ← t_max − 1, ..., 1 do
3 |   S(t) ← τ^{t,♯}_{⊤mon}(S(t + 1));
4 |   if exists t' ∈ {t + 1, ..., t_max} s.t. S(t) = S(t') then
5 |   |   for k ← 1, ..., t − 1 do
6 |   |   |   S(t − k) ← S(t' − k);
7 |   |   break;
8 S(0) ← τ^{0,♯}_{⊤mon}(S(1));
```

Figure 4.7.: Algorithm for fast computation of the initial transformer semantics $S = [\![\varphi]\!]^{tra,\sharp}_{\top mon}$ for LOLA specification φ under the premise that $\forall t, t' \in \mathbb{T} \setminus \{0, t_{max}\}.\ (\Sigma(t) = \Sigma(t')) \rightarrow (\tau^{t,\sharp}_{\Sigma} = \tau^{t',\sharp}_{\Sigma})$.

The algorithm determines the initial transformer semantics $S = [\![\varphi]\!]^{tra,\sharp}_{\top mon}$ from back to front by application of the transformer constructions from lemma 4.17. As soon as a repeating element is found (i.e. condition in line 4 is met) all other element in S (except for the one at instant 0) are directly copied from the already existing part of the structure and not explicitly computed: As argued above, that abstract transformers at t and t' repeat, implies that the elements at $t-k$ and $t'-k$ are also repeating unless $t-k=0$ and so the element from $t'-k$ is copied to $t-k$ in a loop until the beginning of the structure is reached. The first element however requires a

dedicated computation as it depends on default values for offsets and thus is defined differently.

In a practical implementation one can of course even avoid the explicit copying, as described in figure 4.7. Therefore a smart data structure for the storage of the initial semantics can be chosen which uses pointers to link the reused entries to their corresponding transformer object instead of copying it. Consequently, if a repeating element is found during computation, this enables efficient runtime and also an efficient storage of the initial semantics.

Enforcing a repeating abstract transformer with widening techniques

If the chosen abstract transformer domain $\tilde{A}$ is finite and smaller than the chosen trace length t_{max}, a repeating entry is eventually inevitable (a selection of domains with practical relevance that have this property are listed in [FMPW23]). Abstract domains which do not have this guarantee can still lead to a repeating entry for particular specifications but don't have to. If this is not the case, we can take an adaptation of the previous algorithm. Instead of the assignment $S(t) \leftarrow \tau^{t,\sharp}_{\top mon}(S(t+1))$ in line 3 one can use $S(t) \leftarrow S(t+1) \nabla \tau^{t,\sharp}_{\top mon}(S(t+1))$ where ∇ is a *widening operator* (see section 2.4.2) as known from the field of abstract interpretation [CC77, Cou21]. Such a widening operator $\nabla : \tilde{A} \times \tilde{A} \to \tilde{A}$ is characterized by two properties. First, $a \sqcup a' \sqsubseteq^{\tilde{A}} a \nabla a'$ has to hold for any $a, a' \in \tilde{A}$. I.e. the result of the widening must over-approximate its arguments. Second, for every sequence $s_0, s_1, \ldots$ with $s_i = s_{i-1} \nabla a_i$ for $i \in \mathbb{N}^+$ and arbitrary $a_i \in \tilde{A}$, there must be an $n \in \mathbb{N}$ s.t. $s_n = s_{n+1}$. This means, that a subsequent application of widening must eventually result in a repeating entry.

Widening is a very common technique for fixed point approximation in the field of static analysis, especially abstract interpretation. In our setting the widening operator forces the appearance of a repeating element in the initial fixed point computation, yet the corresponding abstract transformers at the widened positions are over-approximations of the actual transformers. Thus they make the computed initial semantics a sound over-approximation (see lemma 4.17). This will later also make the outputs of the monitor sound over-approximations, yet will guarantee an efficient termination of the initial fixed point computation for domains where this is not possible otherwise. There is a wide range of widening operators available in literature, which are usually introduced together with the abstract domains they are used in. For the symbolic domain which is presented in the next chapter we will also briefly touch upon a possible widening strategy.

For the computation of the initial semantics in practice it makes sense to use the following strategy: One starts with an exact computation of the abstract transformers and hopes for a repeating entry. If no repeating entry is found after several steps, a widening strategy can be applied. In particular it might even make sense to use

a cascade of differently coarse widening operators or stages which are increasingly over-approximating the concrete element and thus make a repetition more likely.

With the insight about efficient computation of the initial transformer semantics, we have now prepared all the requirements for the abstract recurrent LOLA monitoring algorithm presented in the next subsection.

4.4.3. Abstract recurrent LOLA monitoring algorithm

Consider a LOLA specification φ, a sequence of inputs $w^{(0)}, w^{(1)}, \ldots, w^{(t_{max})} \in \Gamma^*$ over uncertain input domain Γ and corresponding uncertainty encoding ν, which are passed to the monitor. For $t \in \mathbb{T}$ let $\Sigma^t = uconv_\nu(w^{(t)})$ be the input monitoring stream tuple corresponding to this instant. Recall that we assumed for $w^{(t)}$ to encode only words of length $t+1$ and that at instant t the monitor may also only receive information about the events at instant t (see section 4.3.2).

The recurrent abstract monitoring algorithm for φ (following the principle already sketched in the previous section) is depicted in figure 4.8. The monitoring procedure consists of three steps:

1. First the monitor computes the initial transformer semantics $\mu([\![\varphi]\!]^{tra,\sharp}_{\top^{mon}})$, without any information about the input streams. This can be done in an efficient manner following the algorithm from figure 4.7 possibly extended by application of a widening operation. Note that this first monitoring step can be performed before the actual monitoring starts and the result (the initial transformer semantics) be reused for further monitor executions.

2. After the initial transformer semantics is computed, the monitor iteratively receives the input readings for current instant t and the belonging transformer $\tau = \mu([\![\varphi]\!]^{tra,\sharp}_{\Sigma^t})(t)$ is recomputed. Due to the assumption from the previous section about equal transformer computation in the middle of the trace and the fact that Σ^t may not contain information about instants beyond t we have $\mu([\![\varphi]\!]^{tra,\sharp}_{\Sigma^t})(t+1) = \mu([\![\varphi]\!]^{tra,\sharp}_{\top^{mon}})(t+1)$. Thus the recomputation of $\mu([\![\varphi]\!]^{tra,\sharp}_{\Sigma^t})(t) = \tau^{t,\sharp}_{\Sigma^t}(\mu([\![\varphi]\!]^{tra,\sharp}_{\top^{mon}})(t+1))$ only requires a single application of $\tau^{t,\sharp}_{\Sigma^t}$. The subsequent transformer $\mu([\![\varphi]\!]^{tra,\sharp}_{\top^{mon}})(t+1)$ can be taken from the pre-computed initial semantics.

3. Finally the monitor determines the abstract stream configuration set s for the current instant (called *monitor state*). Therefore it applies the computed transformer τ on the previous monitor state (which is initially $\top^A$) and receives the state of the new instant.

   Given an abstract configuration set $a \in A$ and an abstract transformer $\tau \in \tilde{A}$, τ can be applied to a with help of function *app*. We receive the result – in

general – by applying the concretization of τ on all configurations represented by a and translating it back to the abstract configuration set domain.

$$\begin{aligned} app &: \tilde{A} \times A \to A \\ app(\tau, a) &= \alpha^A(\{r \mid r \in \gamma^{\tilde{A}}(\tau)(c), c \in \gamma^A(a)\}) \end{aligned}$$

The resulting monitor state is then used as monitor output as it can be concretized to the set of potential stream configurations at this instant. After output, the monitor proceeds with step 2 if the end of the input trace has not yet been reached.

---

1 Compute $\mu([\![\varphi]\!]^{tra,\sharp}_{\top^{mon}})$ (or over-approximation);
2 $s \leftarrow \top^A$;
3 **foreach** $t \in \mathbb{T}$ **do**
4 Read input $w^{(t)}$;
5 $\tau \leftarrow \tau^{t,\sharp}_{uconv_\nu(w^{(t)})}(\mu([\![\varphi]\!]^{tra,\sharp}_{\top^{mon}})(t+1))$;
6 $s \leftarrow app(\tau, s)$;
7 Output s;

Figure 4.8.: Abstract recurrent monitoring algorithm for LOLA specification φ. The received input trace at instant t is $w^{(t)}$ over a possible uncertain input domain and corresponding encoding ν. The abstract configuration set domain is denoted by A.

Altogether the monitor yields the output sequence

$$\begin{aligned} s_0 &= app(\tau^{0,\sharp}_{uconv_\nu(w^{(0)})}(\mu([\![\varphi]\!]^{tra,\sharp}_{\top^{mon}})(1)), \top^A), \\ s_1 &= app(\tau^{1,\sharp}_{uconv_\nu(w^{(1)})}(\mu([\![\varphi]\!]^{tra,\sharp}_{\top^{mon}})(2)), s_0), \\ s_2 &= app(\tau^{2,\sharp}_{uconv_\nu(w^{(2)})}(\mu([\![\varphi]\!]^{tra,\sharp}_{\top^{mon}})(3)), s_1), \\ &\vdots \end{aligned}$$

Note that the first element of the initial transformer semantics ($\mu([\![\varphi]\!]^{tra,\sharp}_{\top^{mon}})(0)$) is actually never used in the monitoring algorithm, thus we can directly skip its computation in the algorithm from figure 4.7.

If the specification does not contain future-offsets and either no assumptions or all input readings are certain (and thus the assumption has no effect), then the computation of the initial fixed point can completely be skipped. In this case the current monitor state relies only on the past. Thus, the current abstract transformer is independent of the subsequent one and can be computed without it. However, if uncertainty and assumptions are present, the assumptions from the future may discard some input values for the current instant (see section 4.3.2), and thus the subsequent transformer which encodes this relation must be taken into account.

An example of the proposed monitoring algorithm follows at the end of the next chapter for the symbolic abstract domain, which is presented there.

The LOLA monitor resulting from the algorithm in figure 4.8 is a sound recurrent monitor under uncertainties and assumptions, if the chosen abstract domain and abstract semantics are sound. Further it is a perfect recurrent monitor if both are perfect. This is a direct result from theorem 4.13, lemma 4.17 and the fact that for a perfect transformer abstraction $\gamma^\sharp \circ \alpha^\sharp = \mathbf{id}$ holds. The mentioned theorem and lemma proof that the semantics we compute in the algorithm from figure 4.8 are sound over-approximations of the transformer semantics from definition 4.12. This semantics is in turn a perfect approximation of the monitoring semantics (see theorem 4.13) which yields a perfect recurrent monitor (see lemma 4.7). Further, if the computed transformers are perfect abstractions then this also holds for the corresponding abstract transformer semantics. If additionally the abstract transformer and configuration set domains are perfect, a lossless translation back to concrete transformers and configurations is possible. We can summarize this result in corollary 4.19.

Corollary 4.19 (Based on [HKLS24])**.**

The algorithm from figure 4.8 delivers a sound recurrent monitor under uncertainties and assumptions for LOLA specification φ.

The monitor is perfect if $[\![\varphi]\!]_\Sigma^{tra,\sharp}$ is a perfect abstraction of $[\![\varphi]\!]_\Sigma^{tra}$ and the utilized transformer and configuration abstractions are perfect.

Note again at this point that if widening is used in the algorithm for determining the initial transformer semantics, then the resulting structure is a (sound) over-approximation of the actual abstract transformer semantics. In this case the monitor would be sound but lose perfection.

In addition to soundness and perfectness, we also have that the monitor is trace-length-independent (see definition 2.13), if the computation of the transformer and its application to the current monitor state takes constant time and if the representation of state and transformer in the memory are constant (which is related to the constant runtime). Note that in terms of trance-length-independence we considered the maximal instant t_{max} to be implicit part of the input (see section 2.3.1). Without having available this information at monitor synthesis, a computation of the initial transformer semantics before monitoring starts is only possible if a repeating element is found after a fixed number of steps and no full unrolling of the specification is necessary. In this case the computation of the initial transformer semantics is independent of the concrete t_{max}. Thus, for trace-length-independence we also demand that the initial transformer semantics can efficiently be computed, i.e. a repeating transformer is inevitable to appear.

4.5. Summary

In this chapter we have devoted ourselves to the study of recurrent LOLA monitoring under uncertainties and assumptions. Therefore we have started with the definition of an extended LOLA fixed point semantics (the monitoring semantics) which is capable of describing the output of a LOLA specification for partially given traces.

We have further argued that a simple (0-offset) recurrent LOLA monitor is powerful enough to encompass several advanced monitoring approaches. E.g. traditional, initial monitoring, as well as random access monitoring in a sliding window around the current instant and many more can all be solved with such a monitor. Consequently, we focused in this chapter on the construction of a recurrent LOLA monitor without offset or random access. Additionally in section 4.3.2 we have discussed restrictions for the uncertain inputs that we require for an efficient LOLA monitoring algorithm and a strategy was described how assumptions can be embedded in LOLA specifications.

The main contribution of the chapter is a general algorithmic monitoring approach based on abstractions of the concrete monitoring semantics. Therefore in section 4.4 we have introduced a first abstraction of the monitoring semantics which only preserves the relations between events of subsequent instants (transformer semantics). Later a general theory for a further abstraction of this semantics and conditions for soundness and perfectness, based on principles of abstract interpretation, were given. Subsequently we could show how an efficient, under certain conditions also trace-length-independent monitoring algorithm can be built, provided that an abstraction of stream configurations and transformers is available which is sufficient precise for the intended purpose.

An advantage of the generality of the introduced theory is that it can serve as a universal framework for constructing trace-length-independent monitors for arbitrary LOLA sub-fragments (and other formalisms that can be translated to them) and for analyzing their monitoring properties. In the subsequent chapter we will revisit the theory for a symbolic abstraction of the transformer semantics.

5

Symbolic LOLA monitoring

In this chapter, an instantiation of the abstract recurrent LOLA monitoring framework based on symbolic computation is presented and discussed in terms of soundness, perfectness, and efficiency. The resulting recurrent LOLA monitoring algorithm will be shown to have a constant resource bound and still maintain perfectness in some cases. We will also identify some LOLA fragments, for which the recurrent monitoring is perfect and trace-length-independent and for other fragments a sound over-approximation will be suggested.

An implementation and evaluation of the symbolic recurrent monitoring algorithm from this chapter will then be presented in the subsequent chapter.

The contents of this chapter are based on [HKLS24, KLS22a].

We will start with a discussion on how individual events of a LOLA specification and their combinations can be symbolically encoded in general. Subsequently we will derive symbolic representations of stream configurations and transformers and also a symbolic transformer semantics.

5.1. Symbolic constraints

To encode possible event values and combinations symbolically we use constraint sets of boolean expressions over instant variables (i.e. stream identifiers with attached timestamp to represent a stream event, see definition 2.41). Recall that we use $\mathbb{V}^\varphi$ and $\mathfrak{A}^\varphi$ to denote the instant variables and induced algebra of a LOLA specification φ (see definition 2.37). With $\mathbb{E}^{\mathbb{D}}_{\mathfrak{A},\mathbb{V}}$ we denote expressions of type $\mathbb{D}$ over algebra $\mathfrak{A}$ and variables $\mathbb{V}$. Additionally we use the abbreviation $\mathbb{E}^{\mathbb{D}}_{\mathfrak{A},\varphi} := \mathbb{E}^{\mathbb{D}}_{\mathfrak{A},\mathbb{V}^\varphi}$. We sometimes refrain from explicitly noting the utilized algebra if it is clear from the context. In the case of boolean expressions, we also skip the $\mathbb{B}$ in the superscript and just write $\mathbb{E}_{\mathbb{V}}$

or $\mathbb{E}_\varphi$ respectively for these sets of expressions. For further notation on constraints and constraint sets, see section 2.1.5.

The domain of all boolean constraint sets over algebra $\mathfrak{A}$ and variables $\mathbb{V}$ is called symbolic domain.

> **Definition 5.1** (Symbolic domain)**.**
> The *symbolic domain* over algebra $\mathfrak{A}$ and variables $\mathbb{V}$ is given as $\mathbb{S}^{\mathfrak{A}}_{\mathbb{V}} = 2^{\mathbb{E}^{\mathbb{B}}_{\mathfrak{A},\mathbb{V}}}$.

With the order relation $\cdot \models \cdot$ on constraint sets (see section 2.1.5) the symbolic domain $(\mathbb{S}^{\mathfrak{A}}_{\mathbb{V}}, \models)$ is a complete lattice together with meet operation $\wedge$ and join operation $\vee$ applied to the term representation (i.e. the conjunction of all constraints). Greatest element of this domain is $\emptyset$ not restricting the involved instant variables in any way. Least element is $\{\texttt{false}\}$ which cannot be satisfied by any valuation of the instant variables.

For a LOLA specification φ we use $\mathbb{S}^{\mathfrak{A}}_{\varphi}$ for $\mathbb{S}^{\mathfrak{A}}_{\mathbb{V}_\varphi}$. Again we drop the algebra $\mathfrak{A}$ if it is clear from the context.

5.1.1. Encoding of streams and events

We can encode certain event values easily by constraints like $s^t = v$, which determine the value of stream s of type $\mathbb{D}$ at instant t to have the value $v \in \mathbb{D}$. This way we can represent single certain events but also their combinations, e.g. fully known streams and monitoring stream tuples of stream prefixes without uncertainty. Therefore we just add constraints like the one above for every certain event to a constraint set. Note that to construct such constraints, the induced algebra of the corresponding LOLA specification $\mathfrak{A}^\varphi$ is always sufficient. It contains by definition all necessary sorts from the specification and corresponding value constants, as well as the equality relation for these sorts.

As an example take a LOLA specification φ with input streams s of type $\mathbb{R}$ and t of type $\mathbb{B}$. Let further be $\mathbb{T} = \{0, 1, 2, 3\}$. Suppose the streams are only known for instant 0, where s has an event with value 11 and t with value `true`, and instant 1, where the values are 23 and `false`, respectively. The corresponding monitoring stream tuple T is thus given as

$$T = \{(\langle 11, 23, r_2, r_3\rangle, \langle \texttt{true}, \texttt{false}, b_2, b_3\rangle) \mid \forall i \in \{2, 3\}.\, r_i \in \mathbb{R}, b_i \in \mathbb{B}\}.$$

Algebra $\mathfrak{A}^\varphi$ would contain sorts $\mathbb{R}$ and $\mathbb{B}$ together with their constants and the operator $=$ for $\mathbb{R}$ and $\mathbb{B}$. For representation of T we can use the constraint set

$$\mathcal{C} = \{s^0 = 11, t^0 = \texttt{true}, s^1 = 23, t^1 = \texttt{false}\} \in \mathbb{S}^{\mathfrak{A}^\varphi}_{\varphi}.$$

Since we also want to deal with uncertain inputs in our monitoring approach we have to consider them in the symbolic encoding. First, note that in the case of uncertainties it is strongly influenced by the expressiveness of the utilized algebra $\mathfrak{A}$ which event relations can actually be described in $\mathbb{S}_{\varphi}^{\mathfrak{A}}$. This is different to the symbolic encoding of certain events as discussed above which can always be described in $\mathfrak{A}^{\varphi}$. Algebra $\mathfrak{A}$ thus has to be carefully chosen, s.t. we are able to represent the input monitoring streams, the configurations and transformers we want to deal with in the subsequent monitoring algorithm.

As an example consider the uncertain input $i = \langle (2,5), ?, (7,7) \rangle$ for input stream identifier s of type $\mathbb{R}$, where the tuples denote inclusive intervals of potential values and ? total uncertainty. For $\mathbb{T} = \{0,1,2,3\}$ this input would be represented by monitoring stream tuple

$$T = \{(\langle u_0, u_1, 7, u_3 \rangle) \mid 2 \leq u_0 \leq 5, u_i \in \mathbb{R} \text{ for } i \in \{1,3\}\}.$$

A possible constraint set encoding of T would be

$$C = \{2 \leq s^0, s^0 \leq 5, s^2 = 7\}.$$

However, this symbolic encoding would require the operation $\leq$ on reals to be part of the utilized algebra. Yet, this operator is not necessarily contained in the induced algebra of the belonging specification (e.g. if it is from the linear algebra LOLA fragment).

In the following we therefore demand that an algebra $\mathfrak{A}^{unc}$ is given, which is able to encode all uncertainties that may appear on input streams and which is a super set of the specification's induced algebra. We will see later that this algebra is also sufficient to encode all symbolic configuration sets and transformers that arise in the symbolic monitoring algorithm for this input.

In general, all uncertain input events and their combinations can be expressed over an algebra with appropriate predicates of form $P : \mathbb{D} \to \mathbb{B}$ indicating possible and non-possible values for a particular event and predicates of form $P : \mathbb{D}_1 \times \cdots \times \mathbb{D}_n \to \mathbb{B}$ indicating possible value combinations among different events. These predicates can be logically combined to encode complete monitoring event stream tuples or sections of them.

Prominent examples of partial uncertainty can be encoded as follows:

- **Numeric intervals.** If for numeric stream s the value at instant t is known to be within the interval $[a, b]$, it can be expressed by the constraint $(a \leq s^t) \wedge (s^t \leq b)$, which we also denote as $a \leq s^t \leq b$ in the following. For excluded interval bounds $<$ can be used instead of $\leq$. For upper bound ∞ or lower bound $-\infty$ no constraint has to be included.

- **Finite set of potential values.** If for the event of stream s at instant t the value is known to be within a finite set of values $V = \{v_1, \ldots, v_n\}$ it can be expressed as $\bigvee_{v \in V} (s^t = v)$.

- **Finite set of non-possible values.** If for the event of stream s at instant t the value is known not to be within a finite set $V = \{v_1, \ldots, v_n\}$ it can be expressed as $\bigwedge_{v \in V} (s^t \neq v)$.

- **Linear relations between values.** Linear relations between numeric events on streams $s_1, \ldots, s_n$ at instant t can be expressed as $0 = \sum_{i \in 1,\ldots,n} c_i \cdot s_i^t + o$ for constants $c_1, \ldots, c_n, o \in \mathbb{R}$. Similarly an encoding of non-linear (e.g. quadratic) relations is thinkable. Note that such constraints could also be used to express relations among events from different instants, but this is not supported by the monitoring approach from this thesis (see section 4.3.2).

For complete uncertainty about an event, simply no constraint involving the instant variable is included in the constraint set.

In the following we assume that for all uncertain monitoring inputs a translation strategy to symbolic constraints is available.

5.1.2. Symbolic configuration abstraction

We can now use the theory about symbolic encoding of events and their combinations to build a symbolic abstraction of the configuration sets according to definition 4.15.

Since a configuration is only expressing the relations between events of a specific instant $t \in \mathbb{T}$, we will consequently only use instant variables for instant t in the corresponding constraint set. With $\mathbb{V}^{\varphi,t} := \{s^t \mid s \in I \cup S\}$ for LOLA specification $\varphi = (I, S, O, E)$ we denote all instant variables for instant $t \in \mathbb{T}$. We use $\mathbb{S}^{ctx,\mathfrak{A}}_{\varphi,t} := \mathbb{S}^{\mathfrak{A}}_{\mathbb{V}^{\varphi,t}}$ for the corresponding domain of constraint sets over these instant variables and drop the algebra if clear from the context, especially if it is chosen as $\mathfrak{A}^{unc}$.

It was argued that the symbolic domain over any set of variables forms a complete lattice with order relation $\cdot \models \cdot$. For LOLA specification $\varphi = (I, S, O, E)$ with input stream identifiers $I = \{s_1, \ldots, s_n\}$ and intermediate stream identifiers $S = \{s_{n+1}, \ldots, s_{n+m}\}$, we can additionally define translation functions between stream configurations and their symbolic representations:

$$\begin{aligned}\gamma^{ctx,t} &: \mathbb{S}^{ctx}_{\varphi,t} \to 2^{\mathbf{D}_\varphi} \\ \gamma^{ctx,t}(A) &= \{(v_1,\ldots,v_{n+m}) \mid \bigwedge_{1\leq i\leq n+m} (s_i^t = v_i) \models A\}\end{aligned}$$

and

$$\begin{aligned}\alpha^{ctx,t} &: 2^{\mathbf{D}_\varphi} \to \mathbb{S}^{ctx}_{\varphi,t} \\ \alpha^{ctx,t}(C) &= [C].\end{aligned}$$

Thereby $[C]$ describes a canonical (arbitrarily chosen) symbolic encoding of configuration set C, s.t. $C = \gamma^{ctx,t}([C])$. As mentioned before, the existence of $[C]$ requires the utilized algebra to be powerful enough to encode the configuration set C in a symbolic way, which we assume. In case of fully certain monitor inputs, the induced algebra of the specification $\mathfrak{A}^{\varphi}$ is sufficient. The presence of uncertainty may however require an extended algebra $\mathfrak{A}^{unc}$ to encode the resulting relations among events.

The functions $\alpha^{ctx,t}, \gamma^{ctx,t}$ together with $\mathbb{S}^{ctx}_{\varphi,t}$ form a perfect configuration abstraction as lemma 5.2 shows.

> **Lemma 5.2** (Based on [HKLS24])**.**
>
> Let φ be a LOLA specification and $t \in \mathbb{T}$ an instant.
>
> The complete lattice $(\mathbb{S}^{ctx}_{\varphi,t}, \models)$ and $2^{\mathbf{D}_\varphi} \underset{\alpha^{ctx,t}}{\overset{\gamma^{ctx,t}}{\leftrightarrows}} \mathbb{S}^{ctx}_{\varphi,t}$ are a perfect configuration abstraction.

Proof. Let $C \in 2^{\mathbf{D}_\varphi}$ be a set of stream configurations for specification φ. By definition we have

$$\gamma^{ctx,t}(\alpha^{ctx,t}(C)) = \gamma^{ctx,t}([C]) = C.$$

Further from the definition of $\cdot \models \cdot$ (see section 2.1.5) we can conclude that $\alpha^{ctx,t}$, $\gamma^{ctx,t}$ form a Galois connection:

Let $A \in \mathbb{S}^{ctx}_{\varphi,t}$, $C \in 2^{\mathbf{D}_\varphi}$ be arbitrary elements.

$$\alpha^{ctx,t}(C) \models A \Leftrightarrow [C] \models A \Leftrightarrow [\![[C]]\!] \subseteq [\![A]\!] \Leftrightarrow \gamma^{ctx,t}([C]) \subseteq \gamma^{ctx,t}(A) \Leftrightarrow C \subseteq \gamma^{ctx,t}(A)$$

Consequently $(\mathbb{S}^{ctx}_{\varphi,t}, \models)$ and the belonging translation functions are a perfect configuration abstraction. □

Note that the abstract domain we have introduced is parametric in the concrete instant t of the represented configuration set. We use this for convenience s.t. we can deal with absolute variable identifiers later on. Yet, in section 4.3 we assumed a unique configuration abstraction for every instant instead. This is not a problem, however, as the constraint sets could also be translated into an instant-independent format by simply renaming all instant variables to their plain stream identifiers. Since the constraint sets only contain variables from one instant at a time, this would not cause any conflicts.

5.1.3. Symbolic transformer abstraction

We can also utilize the introduced symbolic encoding for transformers. The basic idea is to use symbolic constraints that involve instant variables of streams at the current and previous instant. Therefore we define $\overline{\mathbb{V}}^{\varphi,t} := \{s^t, s^{t-1} \mid s \in I \cup S\}$ for some $t \in \mathbb{T}$ as the set of instant variables at instant t and the one before (if $t = 0$ we technically use the pseudo-instant -1; however, these variables will not actually be used in the subsequent symbolic transformer semantics). With $\mathbb{S}^{tra,\mathfrak{A}}_{\varphi,t} = \mathbb{S}^{\mathfrak{A}}_{\overline{\mathbb{V}}^{\varphi,t}}$ we denote the corresponding domain of constraint sets and discard the algebra as usual if it is clear from the context.

Note that in general we can understand a transformer $\tau : (\mathbf{D}_\varphi) \to 2^{\mathbf{D}_\varphi}$ as set of tuples

$$\tau \mathrel{\widehat{=}} \{(\overline{v}_1, \ldots, \overline{v}_n, v_1, \ldots, v_n) \mid (v_1, \ldots, v_n) \in \tau((\overline{v}_1, \ldots, \overline{v}_n))\}.$$

So we can basically reuse the functions $\alpha^{ctx,t}$ and $\gamma^{ctx,t}$ from above, extended to instant variables for t and $t-1$. Let therefore $\varphi = (I, S, O, E)$ be a LOLA specification with input stream identifiers $I = \{s_1, \ldots, s_n\}$ and intermediate stream identifiers $S = \{s_{n+1}, \ldots, s_{n+m}\}$. The translation functions are given as

$$\begin{array}{lcl} \gamma^{tra,t} & : & \mathbb{S}^{tra}_{\varphi,t} \to 2^{\mathbf{D}_\varphi} \\ \gamma^{tra,t}(C) & = & \{(\overline{v}_1, \ldots, \overline{v}_{n+m}, v_1, \ldots, v_{n+m}) \mid \bigwedge_{1 \le i \le n+m} ((s_i^t = v_i) \wedge (s_i^{t-1} = \overline{v}_i)) \models C\} \end{array}$$

and

$$\begin{array}{lcl} \alpha^{tra,t} & : & 2^{\mathbf{D}_\varphi} \to \mathbb{S}^{tra}_{\varphi,t} \\ \alpha^{tra,t}(\tau) & = & [\tau]. \end{array}$$

Again $[\tau]$ is an arbitrarily chosen, canonical symbolic encoding of transformer τ, s.t. $\tau = \gamma^{tra,t}([\tau])$.

Analogous to configurations, symbolic transformers are a perfect transformer abstraction.

> **Lemma 5.3** (Based on [HKLS24])**.**
> Let φ be a LOLA specification and $t \in \mathbb{T}$ an instant.
>
> The complete lattice $(\mathbb{S}^{tra}_{\varphi,t}, \models)$ and $T^\varphi \underset{\alpha^{tra,t}}{\overset{\gamma^{tra,t}}{\leftrightarrows}} \mathbb{S}^{tra}_{\varphi,t}$ are a perfect transformer abstraction.

Proof. Based on the isomorphism between transformers and sets of tuples the proof is analogous to the one from lemma 5.2. □

As for configurations before, we use domain and abstraction function that are parametric in the instant of the abstracted transformer. Again, from a theoretical standpoint, this makes no difference, since the instant variables could be renamed to

instant-independent identifiers, because transformers always contain only variables for the current and previous instants. However, this makes also an adaption of the equivalence checking and copying of abstract transformers during the initial fixed point computation (figure 4.7, line 4,6) necessary. Two symbolic transformers have to be considered equal if they describe the same relations between instant variables with respect to the shifted timestamp. Likewise a repeating symbolic transformer in the initial semantics must be adjusted to the actual instant before it can be used in the monitoring algorithm.

5.2. Constraint rewriting

We have now seen how input readings, stream configurations and transformers can be encoded as symbolic constraint sets.

For the symbolic computation of the transformers, it will be necessary to determine a symbolic transformer from its successor. In this context it will be required to remove instant variables from the constraint sets which originate from subsequent transformers but are no longer relevant for the transformer of the current instant. Otherwise we would accumulate more and more instant variables and basically unroll the whole specification. In this section we study the technique to remove variables from constraint sets, which we call *constraint rewriting*. For a perfect monitor it will be important that the relations between the remaining instant variables are the same as before the rewriting process. For a sound monitor, the resulting constraints must at least over-approximate the original ones. We will discuss the topic of symbolic constraint rewriting upfront in this section and then introduce the whole symbolic monitoring approach, which is fundamentally based on it.

As a very simple example to illustrate the idea of constraint rewriting, consider the following constraint set over boolean instant variables s^0, t^0, i^1, i^2, i^3.

$$\mathcal{C} = \{s^0 = (i^1 \wedge i^2 \wedge i^3), t^0 = (i^1 \vee i^2 \vee i^3)\}$$

Imagine we are not interested in the concrete relation to and among the values of i^1, i^2 and i^3. A more concise representation of this constraint set preserving only the variables s^0, t^0 would be

$$\mathcal{C}' = \{s^0 \rightarrow t^0\}.$$

The possible value combinations of s^0, t^0 satisfying $\mathcal{C}$ are (`false`, `false`), (`false`, `true`), (`true`, `true`). Exactly these combinations satisfy the rewritten constraint set $\mathcal{C}'$. However, the connection between s^0, t^0 and i^1, i^2, i^3 is lost during rewriting.

We will now introduce a general theory for perfect and sound rewriting of constraint sets with a constant bound on their size. Therefore we start with a size measure of symbolic constraint sets over an arbitrary algebra $\mathfrak{A}$. We distinguish between a weak measure which is assuming variables, constants and function symbols in the formulas

to be of constant size and a strict one which assumes them to have logarithmic size in terms of the number of available variables, constants and functions. This is the amount of memory a computer can be assumed to require to store the corresponding symbol. Both versions of the measure will later play a role w.r.t. the trace-length-independence of the recurrent LOLA monitor.

Definition 5.4 (Constraint set measure; based on [KLS22a])**.**

The *weak size measure* of an expression over algebra $\mathfrak{A}$ with constants $\mathcal{C}$, function symbols $\mathcal{F}$ and variables $\mathbb{V}$, $|\cdot|_w : \mathbb{E}_{\mathfrak{A},\mathbb{V}} \to \mathbb{N}$ is defined as

$$|v|_w = 1, |c|_w = 1, |f(e_1, \ldots, e_n)|_w = |e_1|_w + \cdots + |e_n|_w + 1$$

for variable $v \in \mathbb{V}$, constant $c \in \mathcal{C}$ and function symbol $f \in \mathcal{F}$.

The *strict size measure* of an expression over algebra $\mathfrak{A}$ with constants $\mathcal{C}$, function symbols $\mathcal{F}$ and variables $\mathbb{V}$, $|\cdot|_s : \mathbb{E}_{\mathfrak{A},\mathbb{V}} \to \mathbb{R} \cup \{\infty\}$ is defined as

$$|v|_s = \log(|\mathbb{V}|), |c|_s = \log(|\mathcal{C}|), |f(e_1, \ldots, e_n)|_s = |e_1|_s + \cdots + |e_n|_s + \log(|\mathcal{F}|)$$

for variable $v \in \mathbb{V}$, constant $c \in \mathcal{C}$ and function symbol $f \in \mathcal{F}$.

Let $S \subseteq \mathbb{E}_{\mathfrak{A},\mathbb{V}}$ be a constraint set. The weak and strict size measure of S is given as

$$|S|_w = \sum_{\varphi \in S} |\varphi|_w \quad \text{and} \quad |S|_s = \sum_{\varphi \in S} |\varphi|_s.$$

If an algebra has an infinite number of function or constant symbols we have $\log |\mathcal{F}| = \infty$ and $\log |\mathcal{C}| = \infty$, respectively. Thus, for algebras with infinitely many constant or function symbols, the strict size measure of a constraint is ∞ as soon as one constant or function application is involved.

A perfect rewriting strategy transforms one constraint set into another one, such that both constraint sets are satisfied by exactly the same valuations of the preserved variables. These variables are called *relevant variables* in the following. We model a rewriting strategy as a function $\mathcal{R}^R : 2^{\mathbb{E}_{\mathfrak{A},\mathbb{V}}} \to 2^{\mathbb{E}_{\mathfrak{A},\mathbb{V}}}$ where $R \subseteq \mathbb{V}$ is the set of relevant variables. If the rewritten constraint set is over-approximating the valuations of the original constraint set, we call the rewriting strategy sound.

Definition 5.5 (Rewriting strategy; based on [KLS22a])**.**

Let $\mathfrak{A}$ be an algebra and $\mathbb{V}$ a set of variables. Let further $R \subseteq \mathbb{V}$ be the subset of relevant variables.

A *rewriting strategy* $\mathcal{R}^R : 2^{\mathbb{E}_{\mathfrak{A},\mathbb{V}}} \to 2^{\mathbb{E}_{\mathfrak{A},\mathbb{V}}}$ is called

- *sound* if $\forall C \subseteq \mathbb{E}_{\mathfrak{A},\mathbb{V}}.\, C \models_R \mathcal{R}^R(C)$.
- *perfect* if $\forall C \subseteq \mathbb{E}_{\mathfrak{A},\mathbb{V}}.\, C \equiv_R \mathcal{R}^R(C)$.

> The rewriting strategy $\mathcal{R}^R$ is further called *constant* w.r.t. the weak or strict constraint set measure respectively, if
>
> $$\exists c \in \mathbb{R}. \forall \mathcal{C} \subseteq \mathbb{E}_{\mathfrak{A},\mathbb{V}}. |\mathcal{R}^R(\mathcal{C})|_w < c \quad \text{or} \quad \exists c \in \mathbb{R}. \forall \mathcal{C} \subseteq \mathbb{E}_{\mathfrak{A},\mathbb{V}}. |\mathcal{R}^R(\mathcal{C})|_s < c.$$

Depending on the algebra $\mathfrak{A}$, there may or may not be a perfect, strict or weak constant rewriting strategy. Note that a strict-constant rewriting strategy can only depict to a finite amount of elements.

For a constraint set $\mathcal{C}$ over variables $\mathbb{V} = \{r_1, \ldots, r_k, n_1, \ldots, n_l\}$ let $\gamma = \bigwedge_{c \in \mathcal{C}} c$ be the term representation, i.e. the conjunction of all constraints in $\mathcal{C}$. Note that perfect rewriting of $\mathcal{C}$ or γ w.r.t. relevant variables $R = \{r_1, \ldots, r_k\}$ can be solved by finding a (quantifier-free) expression γ' s.t. $\gamma' \equiv (\exists n_1, \ldots, n_l. \gamma)$ This task is handled in particular by so-called quantifier elimination strategies, which have been extensively studied in literature (e.g. [Pre29, Hod93, CJ12]). However, these strategies are usually not constant (which may cause the following monitoring algorithm not to be trace-length-independent). Further quantifier elimination is not available for all algebras (e.g. Presburger arithmetic without division and congruency [Nip10]).

In this section, we will study three prominent LOLA fragments and their corresponding algebras in terms of rewritability. Simple constant rewriting (or quantifier elimination) strategies for the fragments will be presented to illustrate the underlying concept and to reason about the properties of these fragments w.r.t. definition 5.5.

The recurrent monitoring algorithm which will be presented afterwards relies on a strict-constant rewriting strategy. Under specific circumstances also a weak-constant rewriting strategy will be sufficient. We will show that the monitor is perfect if the corresponding rewriting strategy is perfect, and sound if it is sound. Thus, the existence of a rewriting strategy for a family of specifications becomes the key to its perfect, recurrent monitorability under uncertainties and assumptions. In general, the more restricted the form of the specifications is, the better a rewriting strategy can be found.

In particular we will examine rewriting for the boolean, the linear algebra and the linear arithmetic fragment of LOLA as defined in definition 2.43. For the boolean fragment we will find a perfect, strict-constant rewriting strategy, for the linear algebra fragment a perfect, weak-constant one, and for linear arithmetic a perfect, non-constant and a sound, weak-constant one. Further we will show that better rewriting strategies do not exist for these fragments.

However, note that common quantifier elimination strategies are usually more sophisticated and use multiple optimizations compared to the simple rewriting strategies presented in this thesis. Thus, they should be preferred in practical applications if they are available and meet the required properties such as constantness (as e.g. for

the boolean fragment). It also makes sense to first use non-constant, perfect quantifier elimination techniques in the monitoring process and switch to other rewriting strategies if the computation of the initial fixed point does not terminate or the monitor runs out of resources. Later in this chapter, we will discuss this aspect in more detail.

5.2.1. The boolean fragment

We start with the boolean LOLA fragment, where all streams (inputs and outputs) are of type $\mathbb{B}$, involved constants are `true` and `false` and functions are the usual boolean operators $\vee, \wedge, \neg \ldots$.

As described above, we require a strategy $\mathcal{R}^R_{\mathbb{B}}$ that receives a boolean constraint set and delivers one of constant size, which is equivalent w.r.t. the variables in R.

The rewriting that will be presented is based on the following fundamental insight: The number of value combinations expressible by the boolean variables in R is finite and thus describable with a constraint set of constant size w.r.t. $|R|$.

As a consequence we can iterate all value combinations of variables in R (i.e. $2^{|R|}$ many) and check if they are satisfiable by the constraint set. In the end, we set up the new constraint set from a single disjunction of all possible variable assignments. The procedure is formalized in the algorithm in figure 5.1.

---

1 $\mathcal{C}' \leftarrow \emptyset$;
2 **foreach** $v \in \mathbb{B}^{|R|}$ **do**
3 **if** $\bigwedge_{r_i \in R} (r_i = v(i)) \models_R \mathcal{C}$ **then**
4 $\gamma \leftarrow \gamma \vee \left(\bigwedge_{r_i \in R} (r_i = v(i)) \right)$;
5 $\mathcal{C}' \leftarrow \{\gamma\}$

Figure 5.1.: Algorithm of rewriting strategy $\mathcal{R}^R_{\mathbb{B}}$ for constraint set $\mathcal{C}$ with $R = \{r_0, \ldots, r_n\}$. Resulting constraint set: $\mathcal{C}'$.

As example consider the constraint set

$$\mathcal{C} = \{u^2 = (i^2 \vee v^1 \vee u^3), v^2 = (i^2 \vee v^1)\}.$$

Assume we apply the rewriting for $R = \{v^1, v^2, u^2\}$. The following combinations of values for R do satisfy $\mathcal{C}$:

$$v^1 = \texttt{false}, v^2 = \texttt{false}, u^2 = \texttt{false} \quad v^1 = \texttt{false}, v^2 = \texttt{false}, u^2 = \texttt{true}$$
$$v^1 = \texttt{false}, v^2 = \texttt{true}, u^2 = \texttt{true} \quad v^1 = \texttt{true}, v^2 = \texttt{true}, u^2 = \texttt{true}$$

Thus, the rewriting (with $= \texttt{true}$ erased and $= \texttt{false}$ replaced by negation) results in the constraint set

$$\{(\neg v^1 \wedge \neg v^2 \wedge \neg u^2) \vee (\neg v^1 \wedge \neg v^2 \wedge u^2) \vee (\neg v^1 \wedge v^2 \wedge u^2) \vee (v^1 \wedge v^2 \wedge u^2)\}$$

which is equal to

$$\mathcal{C}' = \{(v^1 \to v^2) \wedge (v^2 \to u^2)\}$$

This constraint set reflects exactly the relation between v^2, u^3, v^3 that was expressed by the original constraint set $\mathcal{C}$.

In lemma 5.6 we conclude that the boolean rewriting strategy is strict-constant and perfect.

> **Lemma 5.6** (Based on [KLS22a])**.**
>
> The rewriting strategy for boolean algebra as presented in figure 5.1 is strict-constant and perfect.

Proof. The rewriting produces a DNF over the variables in R. Thus, the rewriting is a strict-constant one. Likewise the resulting constraint set models exactly those valuations of variables in R which were modeled by the original constraint set. □

Several other finite domains (e.g. enumerations, modulo fields etc.) can be embedded in boolean algebra. Hence, they can also be rewritten in a perfect, strict-constant manner.

5.2.2. The linear algebra fragment

In the linear algebra LOLA fragment the constraint set may only contain constraints of the form

$$\sum_{v \in \mathbb{V}} c_v \cdot v + o_1 = \sum_{v \in \mathbb{V}} c'_v \cdot v + o_2$$

where $\mathbb{V}$ is the set of variables, $c_v, c'_v \in \mathbb{R}$ for $v \in \mathbb{V}$ and $o_1, o_2 \in \mathbb{R}$ are constants.

We assume all these equations to be transformed into a normal form where every variable multiplication is on the right hand side and constants are on the left (which can be easily done by equivalence transformations). Consequently the equations look as follows.

$$o = \sum_{v \in \mathbb{V}} c_v \cdot v$$

We call the c_v coefficients and o the offset. We say an equation contains a variable if it has a non-zero coefficient.

Note that by definition 2.43 we do actually not allow assumptions in the linear algebra fragment, as they are encoded in a boolean stream Λ which is not supported

there. Yet we can permit such a stream if it only consists of a conjunction of linear algebra equations. In this case we can consider these equations directly as part of the constraint set, instead of the two constraints $\Lambda^t = \dots$ and Λ^t as usual.

In linear algebra we can interpret a constraint set as a linear equation system over $\mathbb{V} = \{s_1, \dots, s_n\}$, which can also be denoted in matrix notation:

$$\begin{pmatrix} o_1 \\ \vdots \\ o_m \end{pmatrix} = \begin{pmatrix} c_{1,1} & \dots & c_{1,n} \\ & \ddots & \\ c_{m,1} & \dots & c_{m,n} \end{pmatrix} * \begin{pmatrix} s_1 \\ \vdots \\ s_n \end{pmatrix}$$

Concerning the order of the variables we require the relevant variables ($R = \{r_1,\dots,r_l\} \subseteq \mathbb{V}$) to be preceded by the non-relevant ones ($\mathbb{V} \backslash R = \{n_1, \dots, n_k\}$):

$$\begin{pmatrix} o_1 \\ \vdots \\ o_m \end{pmatrix} = \left(\begin{array}{ccc|ccc} c'_{1,1} & \dots & c'_{1,k} & c'_{1,k+1} & \dots & c'_{1,k+l} \\ & \ddots & & & \ddots & \\ c'_{m,1} & \dots & c'_{m,k} & c'_{m,k+1} & \dots & c'_{m,k+l} \end{array}\right) * \left(\begin{array}{c} n_1 \\ \vdots \\ n_k \\ \hline r_1 \\ \vdots \\ r_l \end{array}\right)$$

We assume throughout our algorithm that the constraint set is indeed satisfiable. The opposite could only happen if the assumption encoded in stream Λ is not satisfiable for the given inputs. However, as already mentioned in the preliminaries chapter, we do not consider this case in this thesis.

We now aim to eliminate every non-relevant variable from the system of equations using the *Gaussian elimination method* [Str80]. Therefore we transform the matrix into *row echelon form* [Str80], i.e. the first non-zero coefficient of each row is strictly right of the one from the previous row, or the row contains only zero coefficients. To transform the matrix into this shape we proceed as follows: We choose one equation from the system s.t. no other equation has a non-zero coefficient further on the left. We then eliminate this left-most variable from all other equations. Therefore we multiply and subtract the chosen equation from all others, s.t. the resulting equations do not contain the variable to be removed anymore. The chosen equation remains the only one with this variable but no variable further on the left and is added to the resulting equation system in row echelon from. The procedure is then repeated for the remaining equations until no more equations are left. After this transformation the matrix multiplication is in the following form:

$$\begin{pmatrix} o'_1 \\ \vdots \\ o'_m \end{pmatrix} = \begin{pmatrix} c''_{1,1} & c''_{1,2} & \dots & c''_{1,i} & \dots & c''_{1,n} \\ 0 & c''_{2,2} & \dots & c''_{2,i} & \dots & c''_{2,n} \\ \vdots & & \ddots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & c''_{m,i} & \dots & c''_{m,n} \end{pmatrix} * \left(\begin{array}{c} n_1 \\ \vdots \\ n_k \\ \hline r_1 \\ \vdots \\ r_l \end{array}\right)$$

Depending on whether the equation system is over- or under-determined, the matrix might contain rows with all coefficients zero, or the last row might still contain non-zero coefficients.

We can then remove all equations from the system, where a non-relevant variable is contained (i.e. has a non-zero coefficient). This is because these equations do not contain any direct information about the relation among relevant variables, as also non-relevant ones are involved. Additionally there are no other equations to eliminate the non-relevant variables without introducing a new non-relevant variable, because of the row echelon form the matrix has. Equations where all coefficients are zero can of course also be neglected. Let row o be the first row where all non-relevant variables are zero. Altogether we receive the following equation system of r equations consisting only of relevant variables:

$$\begin{pmatrix} o''_1 \\ \vdots \\ o''_r \end{pmatrix} = \begin{pmatrix} c''_{o+1,k+1} & c''_{o+1,k+2} & \cdots & c''_{o+1,k+l} \\ 0 & c''_{o+2,k+2} & \cdots & c''_{o+2,k+l} \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & c''_{o+r,k+l} \end{pmatrix} * \begin{pmatrix} r_1 \\ \vdots \\ r_l \end{pmatrix}$$

From these equations the rewritten constraint set can be built. Note that r is limited by the number of relevant variables l. This is because the row vectors in the final matrix are linearly independent (because of the row echelon form). Since the matrix has l columns its rank and thus the number of linearly independent row vectors is limited by l.

For illustration consider the following constraint set with assumption Λ^4:

$$\{u^4 = 2i^2 + 4i^3, v^4 = i^0 + i^1 - i^2, w^4 = 2i^3 + i^0, \Lambda^4 = ((i^1 = 7) \wedge (-i^0 = 2i^3)), \Lambda^4\}$$

Suppose we want to rewrite this constraint set for relevant variables $R = \{u^4, v^4, w^4\}$ After resolving the assumptions and transforming the equations we receive the following equation system:

$$\begin{pmatrix} 0 \\ 0 \\ 0 \\ 7 \\ 0 \end{pmatrix} = \left(\begin{array}{cccc|ccc} 0 & 0 & 2 & 4 & -1 & 0 & 0 \\ 1 & 1 & -1 & 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & 2 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 2 & 0 & 0 & 0 \end{array}\right) * \left(\begin{array}{c} i^0 \\ i^1 \\ i^2 \\ i^3 \\ \hline u^4 \\ v^4 \\ w^4 \end{array}\right)$$

In row echelon form we have

$$\begin{pmatrix} 0 \\ 0 \\ 7 \\ -14 \\ 0 \end{pmatrix} = \left(\begin{array}{cccc|ccc} 1 & 1 & -1 & 0 & 0 & -1 & 0 \\ 0 & -1 & 1 & 2 & 0 & 1 & -1 \\ 0 & 0 & 1 & 2 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & -1 & -2 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right) * \left(\begin{array}{c} i^0 \\ i^1 \\ i^2 \\ i^3 \\ \hline u^4 \\ v^4 \\ w^4 \end{array}\right).$$

Thus we preserve the last two equations and get

$$\begin{pmatrix} -14 \\ 0 \end{pmatrix} = \begin{pmatrix} -1 & -2 & 2 \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} u^4 \\ v^4 \\ w^4 \end{pmatrix}.$$

or in constraint representation

$$\{-u^4 - 2v^4 + 2w^4 = -14, w^4 = 0\} \equiv \{u^4 = -2v^4 + 14, w^4 = 0\}$$

which expresses exactly the relation between u^4, v^4, w^4 in the original constraint set, i.e. that w^4 is 0 and u^4 is the double of $-v^4$ plus 14.

Algorithmically the rewriting procedure for linear algebra is described in figure 5.2. The version there is slightly adjusted and performs the transformation to echelon normal form and elimination of equations in parallel. Therefore the algorithm proceeds as follows: After transforming the equation system into the proposed normal form it iterates over all variables in the suggested order. One after another it takes a row r where this coefficient of the variable is not zero and subtracts this row (adjusted by a factor) from all other rows, including r itself, to eliminate this variable from the equation system. Note that by this procedure row r is replaced by $0 = 0$. The rows where all non-relevant variables are 0 are added to the resulting constraint set C'.

As in the case of boolean rewriting, the linear algebra rewriting strategy is also perfect, but only weak-constant.

> **Lemma 5.7** (Based on [KLS22a])**.**
>
> The rewriting strategy for linear algebra as presented in figure 5.2 is weak-constant and perfect.

Proof. The algorithm only utilizes equivalence rewritings, which do not introduce errors. As argued above, equations with non-relevant variables in the echelon normal form do not allow any implications on relations among relevant variables and can thus be removed.

$\mathcal{C}' \leftarrow \emptyset$;
Transform $\mathcal{C}$ to $O = (M_N|M_R) * (n_1, \ldots, n_k, r_1, \ldots, r_l)^T$;
$M \leftarrow (M_N|M_R|O)$;

foreach $i \in \{1, \ldots, k+l\}$ **do**
  if *exists row* r *in* M *with first non-zero coefficient in column* i **then**
    if $i > k$ **then**
      $C' \leftarrow C' \cup \{r\}$;
    foreach *row* r' *in* M **do**
      Replace r' by $r' - \frac{r'(i)}{r(i)} \cdot r$ in $(M_N|M_R|O)$;

Figure 5.2.: Algorithm of rewriting strategy $\mathcal{R}^R_{\mathcal{LA}}$ with $R = \{r_1, \ldots, r_l\}$ for constraint set $\mathcal{C}$ over $\mathbb{V} = R \cup \{n_1, \ldots, n_k\}$. Resulting constraint set: $\mathcal{C}'$.

The resulting constraint set contains at most l equations with at most l variables and is thus of constant size for a fixed number of relevant variables, according to the weak measure. □

It is easy to see that a perfect, strict-constant rewriting strategy cannot exist for the linear algebra fragment. Consider e.g. the family of constraint sets $\{s = 1\}, \{s = 2\}, \ldots$. Perfect rewriting for relevant variables $R = \{s\}$ would require each of these sets to be rewritten to a different constraint set. This causes infinitely many constraint sets to result from rewriting, which prevents a strict constant size.

Finally, note that the approach which was presented in this section is not only applicable to real linear algebra, but in general to any vector space algebra.

5.2.3. The linear arithmetic fragment

The linear real arithmetic LOLA fragment combines both previously examined fragments, additionally allowing the operators $<, =$ applied to reals inside boolean constraints (see definition 2.43). Linear real arithmetic is of great practical relevance as lots of properties with continuous and discrete components can be expressed. Unfortunately there is no perfect, weak-constant (thus also no strict-constant) rewriting strategy for this fragment, as proved in the subsequent lemma.

Lemma 5.8.
There is no perfect, strict-constant nor weak-constant rewriting strategy for linear real arithmetic.

Proof. Let $\mathcal{C}$ be a constraint set over variables $\mathbb{V} = \{r\}$ and γ its term representation, resulting from conjunction of all constraints in $\mathcal{C}$. We say for variable r, that $p \in \mathbb{R}$ is a separating point if there is a $\delta \in \mathbb{R}^+$, s.t. $\forall \epsilon \in \mathbb{R}. (0 < \epsilon < \delta) \Rightarrow (r = p) \models \mathcal{C} \Leftrightarrow (r = p + \epsilon) \not\models \mathcal{C}$ In other words: the satisfaction of $\mathcal{C}$ flips directly at value p for r.

Note that we can easily construct a constraint set with any number of separating points for r: $\mathcal{C}_n = \{(r = 1) \vee (r = 2) \vee \cdots \vee (r = n)\}$. While the expression $(r = k)$, for $k \in \mathbb{N}$, $1 \leq k \leq n$ models $\mathcal{C}_n$, $(r = k + \epsilon)$ for $0 < \epsilon < 1$ does not and hence $\mathcal{C}_n$ has n separating points for r.

A perfect rewriting strategy for relevant variables $R = \{r\}$ would by definition have to preserve all separating points. On the other hand the number of separating points a constraint set $\mathcal{C}$ (with term representation γ) has, is limited by its size. Assume γ contains m linear sub-expressions of form $o + cr = o' + c'r$ or $o + cr < o' + c'r$ for constants $o, o', c, c' \in \mathbb{R}$. Every conjunction which contains every of these linear sub-expressions or its negation can have at most m separating point for r. Consequently, a constraint with m linear (in)-equalities as sub-expressions cannot have more than $2^m \times m$ separating points for a variable r as it can be written in distributive normal form over its linear sub-expressions.

Hence, there is no weak-constant (and thus also no strict-constant) rewriting strategy for linear real arithmetic. □

As a consequence this section will present a perfect, but non-constant rewriting strategy for the linear real arithmetic LOLA fragment. Further, a sound and weak-constant rewriting strategy is discussed.

The idea behind perfect rewriting of linear real arithmetic formulas is to substitute all (in)-equalities in the constraint set by fresh boolean variable identifiers and, as a first step, to perform boolean rewriting. The boolean rewriting results in a single constraint in distributive normal form. In this constraint the boolean variables are replaced by the original linear real arithmetic expressions again and variable elimination is then performed on each conjunction of (in)-equalities.

Core of the procedure is thus the elimination of real variables from conjunctions of (in)-equalities. The approach that was used in the previous section for elimination of variables in sets of equations can in general not be applied to sets of inequalities. This is because in the case of inequalities it is not sufficient anymore to use a single equation to remove a variable from all other constraints. One would rather have to use all pairs of equations to generate the new constraints without the variable to be eliminated.

A strategy to eliminate variables in sets of inequalities, which basically relies on the idea of combining all suitable pairs of inequalities, is *Fourier–Motzkin elimination* [KS16]. In the first step, the approach rewrites all the equations, s.t. they have

the variable to be eliminated isolated on one side. Then it combines all suitable (according to their inequality sign) pairs of sides, which do not contain the variable to be eliminated, to build new inequalities. Specifically, for a set of inequalities where variable x is isolated on the left side,

$$\{x < \sum_i c_i^1 x_i, \ldots, x < \sum_i c_i^n x_i, x > \sum_i d_i^1 x_i, \ldots, x > \sum_i d_i^m x_i\}$$

each right hand side of the $<$-equations is combined with each right hand side of the $>$-equations. This leads to the following equivalent set of $n \cdot m$ inequalities without variable x:

$$\{\sum_i d_i^1 x_i < \sum_i c_i^1 x_i, \sum_i d_i^2 x_i < \sum_i c_i^1 x_i, \ldots, \sum_i d_i^m x_i < \sum_i c_i^n x_i\}$$

Thus, unlike Gaussian elimination, the number of resulting constraints after removing a single variable is quadratic in the number of original constraints, making the rewriting non-constant. The method is also applicable to non-strict inequalities ($\leq/\geq$). In this case the combination of two non-strict inequalities also leads to a non-strict one, while combinations of at least one strict inequality result in a strict one. Equations can be handled by splitting them into two non-strict inequalities. The overall algorithm for elimination of a variable from a set of linear inequalities can be found in figure 5.3.

The algorithm for the entire linear real arithmetic rewriting strategy based on Fourier-Motzkin elimination is presented in figure 5.4. It is an extension of the boolean rewriting strategy from figure 5.1. At first all inequality sub-expressions in $\mathcal{C}$ are replaced by fresh variables, then all combinations of relevant boolean variables and the replacement variables are iterated. If a specific valuation is a model of the rewritten constraint set, the replacement variables are substituted again by the original inequalities, and the procedure from figure 5.3 is applied to eliminate all non-relevant variables from them. Out of the resulting constraints and the valuations of the relevant boolean variables a product term is built and added to the DNF γ, which becomes the only constraint of the resulting constraint set C'.

The rewriting strategy is perfect, as shown in lemma 5.9.

Lemma 5.9.
The rewriting strategy for linear real arithmetic from figure 5.4 is perfect.

Proof. The boolean rewriting preserves equivalence for all relevant variables, i.e. the relevant boolean variables and the boolean variables substituting inequalities. For each product term in the resulting DNF the conjunction of inequalities is rewritten to be free of non-relevant variables by Fourier-Motzkin elimination which solely applies equivalence transformations. □

```
1  L ← ∅;
2  G ← ∅;
3  I' ← ∅;
4  foreach inequality i ∈ I do
5      if i does not contain x then
6          I' ← I' ∪ {i};
7          continue;
8      Transform i s.t. x is isolated without coefficient on the left hand side;
9      if sign in i is < or ≤ then
10         L ← L ∪ {i};
11     else
12         G ← G ∪ {i};
13 foreach x ⊗ t ∈ L do
14     foreach x ⊙ t' ∈ G do
15         if {⊗, ⊙} = {≤, ≥} then
16             ⊕ ← ≤
17         else
18             ⊕ ← <
19         I' ← I' ∪ {t' ⊕ t};
```

Figure 5.3.: Fourier-Motzkin algorithm [KS16] for elimination of variable x from the set of inequalities $\mathcal{I}$. The symbols $\otimes, \odot, \oplus \in \{<, >, \leq, \geq\}$ are placeholders for inequality relations. Resulting set of inequalities $\mathcal{I}'$.

As example consider the following constraint set on which we perform rewriting for $R = \{s^1, e^1\}$

$$\mathcal{C} = \{e^2 < 20, s^1 \rightarrow (e^1 - e^2 < 10), \neg s^1 \rightarrow (e^1 - e^2 < 5)\}$$

We replace $e^2 < 20$ by a_1, $e^1 - e^2 < 10$ by a_2 and $e^1 - e^2 < 5$ by a_3, and receive the constraint set

$$\{a_1, s^1 \rightarrow a_2, \neg s^1 \rightarrow a_3\}.$$

The satisfying valuations of this constraint set are

$$(s^1 \wedge a_1 \wedge a_2 \wedge a_3), (\neg s^1 \wedge a_1 \wedge a_2 \wedge a_3), (\neg s^1 \wedge a_1 \wedge \neg a_2 \wedge a_3) \text{ and } (s^1 \wedge a_1 \wedge a_2 \wedge \neg a_3).$$

One of the contained assignments to a_1, a_2, a_3 is $a_1 \wedge a_2 \wedge a_3$, corresponding to

$$\{e^2 < 20, e^1 - e^2 < 10, e^1 - e^2 < 5\}.$$

To eliminate e^2 we isolate it on the left hand side of each inequality:

$$\{e^2 < 20, e^2 > e^1 - 10, e^2 > e^1 - 5\}$$

1 Replace all equations in $\mathcal{C}$ by conjunction of two non-strict inequalities;
2 Replace inequalities $i_1, \ldots, i_n$ in $\mathcal{C}$ by fresh boolean variables $a_i \in A = \{a_0, \ldots, a_n\}$;
3 $\mathcal{C}' \leftarrow \emptyset$;
4 **foreach** $v \in \mathbb{B}^{|R_\mathbb{B}|}$ **do**
5 **foreach** $v' \in \mathbb{B}^{|A|}$ **do**
6 **if** $\bigwedge_{r_i \in R_\mathbb{B}} (r_i = v(i)) \wedge \bigwedge_{a_i \in A} (a_i = v'(i)) \models_{R_\mathbb{B} \cup A} \mathcal{C}$ **then**
7 $S \leftarrow \emptyset$;
8 **foreach** $a_i \in A$ **do**
9 **if** $v'(i)$ **then**
10 $S \leftarrow S \cup \{i_i\}$;
11 **else**
12 $S \leftarrow S \cup \{\overline{i_i}\}$;
13 **foreach** $e \in \mathbb{V}_\mathbb{R} \backslash R_\mathbb{R}$ **do**
14 Eliminate e in S;
15 $\gamma \leftarrow \gamma \vee \left(\left(\bigwedge_{r_i \in R_\mathbb{B}} (r_i = v(i)) \right) \wedge \left(\bigwedge_{s \in S} s \right) \right)$;
16 $\mathcal{C}' \leftarrow \{\gamma\}$;

Figure 5.4.: Algorithm of rewriting strategy $\mathcal{R}^R_{\mathcal{LRA}}$ for constraint set $\mathcal{C}$ over variables $\mathbb{V}$. Set $\mathbb{V}$ splits into $\mathbb{V}_\mathbb{B}$ which contains all boolean variables and $\mathbb{V}_\mathbb{R}$ which contains all real variables. An analogous notation is used for the sets of relevant variables R and its subsets $R_\mathbb{B} = \{r_0, \ldots, r_m\}$ and $R_\mathbb{R}$. With $\overline{i_i}$ the negated version of inequality i_i is denoted (i.e. $<$ exchanged by $\geq$ etc.). Resulting constraint set: $\mathcal{C}'$.

Applying the Fourier-Motzkin transformation ultimately leads to the constraints $\{e^1 - 10 < 20, e^1 - 5 < 20\} \equiv \{e^1 < 25\}$. I.e. $a_1 \wedge a_2 \wedge a_3$ can be replaced by $e^1 < 25$. Applying the same strategy for the remaining a_i conjunctions we get

$$\{(s^1 \wedge (e^1 < 25)) \vee (\neg s^1 \wedge (e^1 < 25)) \vee (\neg s^1 \wedge (e^1 < 25) \wedge (e^1 - 5 < e^1 - 10)) \vee (s^1 \wedge (e^1 < 30) \wedge (e^1 - 10 < e^1 - 5))\}$$

which can be simplified (note that the third product term is not satisfiable) to

$$\{(s^1 \wedge (e^1 < 30)) \vee (e^1 < 25)\}.$$

As mentioned above, the problem with this strategy is that the size of the resulting constraint set is not bounded by a constant, but depends on the size of the original constraint set, specifically the number of inequalities there. There are several approaches to tackle this problem (which also plays a significant role in the field of

static analysis) and provide a weak-constant rewriting algorithm. One possibility is a restriction of the set of considered inequalities A to a constant number (either in general or separately for each product term in the resulting DNF). This leads to a sound but less precise over-approximation of the original constraint set. It is especially useful when there is an intelligent strategy for the application scenario that determines which inequalities to select. A selection strategy for a similar problem is e.g. discussed in [SQ18].

In this thesis we will consider another rewriting strategy (similar to the one from [KLS22a]) for the linear real arithmetic fragment in detail, which is based on a separate rewriting of the real and boolean parts of the specification. This separation leads to a total loss of information about all interconnections between variables from the two different types. To mitigate this somewhat, minimum and maximum bounds for all relevant real variables (if they exist) are added to the constraint set instead.

The specific steps of the rewriting strategy are:

1. Separate the constraint set into linear equations and other constraints.
2. Use the linear algebra rewriting algorithm (figure 5.2) for all constraints w.r.t. the relevant real variables.
3. Compute bounds for all relevant real variables and add them to the new constraint set.
4. Use the boolean rewriting algorithm (figure 5.1) for all constraints w.r.t. the relevant boolean variables.

However, the applied bound query in this algorithm does not have to be perfect but can be done by any sound numerical approximation approach. Obviously bound inference always causes precision loss for linear real arithmetic constraints, even if the bounds are inferred perfectly. Consider e.g. the simple constraint set $\mathcal{C} = \{r = 1 \vee r = 3\}$ and $R = \{r\}$. For r, the perfect bounds $1 \leq r$ and $r \leq 3$ can be inferred, yet $r = 2$ does not conform to $\mathcal{C}$.

The algorithm for the whole imperfect rewriting strategy for linear real arithmetic is presented in figure 5.5.

As example consider again the constraint set $\mathcal{C}$ from before and $R = \{s^1, e^1\}$:

$$\mathcal{C} = \{e^2 < 20, s^1 \rightarrow (e^1 - e^2 < 10), \neg s^1 \rightarrow (e^1 - e^2 < 5)\}$$

Since there are no linear equations in the constraint set, $\mathcal{C}_{\mathcal{LA}} = \emptyset$ follows. The linear algebra rewriting strategy consequently delivers an empty constraint set. Yet one is able to infer the bound $e^1 < 30$ on e^1, thus $\mathcal{C}'_{\mathcal{LA}} = \{e^1 < 30\}$. The boolean rewriting for $R_{\mathbb{B}} = \{s^1\}$ results in $\mathcal{C}'_{\mathbb{B}} = \{s^1 \vee \neg s^1\} \equiv \{\texttt{true}\} \equiv \{\}$. Thus the rewritten constraint set is given as

$$\mathcal{C}' = \{e^1 < 30\}.$$

Extract linear algebra constraints from $\mathcal{C}$ into $\mathcal{C}_{\mathcal{LA}}$;
$\mathcal{C}'_{\mathcal{LA}} \leftarrow \mathcal{R}^{R_{\mathbb{R}}}_{\mathcal{LA}}(\mathcal{C}_{\mathcal{LA}})$;
foreach $r \in R_{\mathbb{R}}$ **do**
    if *there is bound* $m \in \mathbb{R}$ *s.t. for all* $m' \leq m$ *:* $(r = m') \not\models_{\{r\}} \mathcal{C}$ **then**
        $\mathcal{C}'_{\mathcal{LA}} \leftarrow \{m < r\}$
    if *there is bound* $m \in \mathbb{R}$ *s.t for all* $m' \geq m$ *:* $(r = m') \not\models_{\{r\}} \mathcal{C}$ **then**
        $\mathcal{C}'_{\mathcal{LA}} \leftarrow \{m > r\}$
$\mathcal{C}'_{\mathbb{B}} \leftarrow \mathcal{R}^{R_{\mathbb{B}}}_{\mathbb{B}}(\mathcal{C})$;
$\mathcal{C}' \leftarrow \mathcal{C}'_{\mathcal{LA}} \cup \mathcal{C}'_{\mathbb{B}}$;

Figure 5.5.: Algorithm of imperfect rewriting strategy $\widetilde{\mathcal{R}}^{R}_{\mathcal{LRA}}$ with $R = R_{\mathbb{B}} \cup R_{\mathbb{R}}$ for constraint set $\mathcal{C}$. $R_{\mathbb{B}}$ contains all boolean variables in R, $R_{\mathbb{R}}$ all real variables. Resulting constraint set: $\mathcal{C}'$.

While the bound on e^1 can thus be preserved, the relation between the boolean variable s^1 and the real variable e^1 gets lost.

However the presented rewriting strategy is still sound and weak-constant. This is proofed in lemma 5.10.

Lemma 5.10 (Based on [KLS22a])**.**

The rewriting strategy for linear real arithmetic as presented in figure 5.5 is sound and weak-constant.

Proof. Assuming the opposite, there would have to be a valuation of variables in R, s.t. $\{r_1 = v_1, \ldots, r_n = v_n\} \models_R \mathcal{C}$ but $\{r_1 = v_1, \ldots, r_n = v_n\} \not\models_R \mathcal{C}'$. This is a contradiction. Let $r_1, \ldots, r_k$ be the relevant boolean variables, then we have by perfectness of the boolean rewriting strategy (lemma 5.6) $\{r_1 = v_1, \ldots, r_k = v_k\} \models_{R_{\mathbb{B}}} \mathcal{C}'_{\mathbb{B}}$.

For real variables $r_{k+1}, \ldots, r_n$ we have $\{r_{k+1} = v_{k+1}, \ldots, r_n = v_n\} \models_{R_{\mathbb{R}}} \mathcal{C}$ and thus also $\{r_{k+1} = v_{k+1}, \ldots, r_n = v_n\} \models_{R_{\mathbb{R}}} \mathcal{C}_{\mathcal{LA}}$, as $\mathcal{C}_{\mathcal{LA}}$ contains a subset of the constraints from $\mathcal{C}$. By perfectness of the real rewriting strategy (lemma 5.7) $\{r_{k+1} = v_{k+1}, \ldots, r_n = v_n\} \models_{R_{\mathbb{R}}} \mathcal{C}'_{\mathcal{LA}}$ holds before insertion of additional constraints. If we add a constraint $m < r_i$ (or $m > r_i$ respectively), then $v_i > m$ ($v_i < m$) holds by the if condition and thus $\{r_{k+1} = v_{k+1}, \ldots, r_n = v_n\} \models_{R_{\mathbb{R}}} \mathcal{C}'_{\mathcal{LA}}$ still holds for the final constraint set $\mathcal{C}'_{\mathcal{LA}}$.

As there is no constraint in $\mathcal{C}'_{\mathbb{B}}$ or $C'_{\mathcal{LA}}$ involving boolean and real variables at the same time we also have $\{r_1 = v_1, \ldots, r_n = v_n\} \models_R \mathcal{C}'$.

The resulting constraint set contains the results from the boolean and linear algebra rewriting and additionally at most two bound constraints for each relevant real variable. The rewriting is thus weak-constant. □

5.3. Symbolic monitoring

Based on the symbolic abstraction of stream configurations and configuration transformers, as well as the rewriting strategies, we will now develop a symbolic abstraction of the transformer semantics from definition 4.12. This will eventually yield us a sound or perfect recurrent LOLA monitor (see corollary 4.19) – dependent on whether a sound or perfect rewriting strategy was chosen.

5.3.1. Symbolic transformer semantics

Following lemma 4.17, it is sufficient for the abstract transformer semantics to give computations for all abstract transformers and compose them. We define a symbolic LOLA expression semantics, and based on this, the symbolic transformer computation in definition 5.11.

Definition 5.11 (Symbolic transformer computation; based on [HKLS24])**.**

Let $\varphi = (I, S, O, E)$ be a LOLA specification with assumption stream $\Lambda \in S$.

The symbolic semantics of a LOLA expression $e \in \mathrm{Exp}_{\mathbb{D}}^{I \cup S}$ at instant $t \in \mathbb{T}$, $[\![e]\!]^{\mathrm{sym}}(t) \in \mathbb{S}_{\varphi}^{\mathfrak{A}^{\varphi}}$, is given as

- $[\![c]\!]^{\mathrm{sym}}(t) = c$
- $[\![s[o, c]]\!]^{\mathrm{sym}}(t) = c$ if $(t + o) \notin \mathbb{T}$
- $[\![s[o, c]]\!]^{\mathrm{sym}}(t) = s^{t+o}$ if $(t + o) \in \mathbb{T}$
- $[\![f(e_1, \ldots, e_n)]\!]^{\mathrm{sym}}(t) = \mathbf{f}([\![e_1]\!]^{\mathrm{sym}}(t), \ldots, [\![e_n]\!]^{\mathrm{sym}}(t))$
- $[\![ite(e_1, e_2, e_3)]\!]^{\mathrm{sym}}(t) = \mathbf{ite}([\![e_1]\!]^{\mathrm{sym}}(t), [\![e_2]\!]^{\mathrm{sym}}(t), [\![e_3]\!]^{\mathrm{sym}}(t))$

for constant $c \in \mathbb{D}$, offset $o \in \{-1, 0, 1\}$ and expressions $e_1, \ldots, e_n \in \mathrm{Exp}^{I \cup S}$. With **f** and **ite** the symbolic representations of functions f and ite is denoted.

Let $\Sigma \in \mathbb{S}_{\varphi}^{\mathfrak{A}^{unc}}$ be a symbolic encoding of the (potentially) uncertain input steams and $\mathcal{R}$ a rewriting strategy for $\mathfrak{A}^{unc}$. The *symbolic transformer for* t_{max} is defined as

$$\tau_{\Sigma}^{\mathrm{sym}, t_{max}} = \mathcal{R}^{\overline{\mathbb{V}}^{\varphi, t_{max}}}(\{s^{t_{max}} = [\![E(s)]\!]^{\mathrm{sym}}(t_{max}) \mid s \in S\} \cup \Sigma \cup \{\Lambda^{t_{max}}\})$$

The *symbolic transformer for* $t \in \mathbb{T}\backslash\{t_{max}\}$ w.r.t. its subsequent symbolic transformer τ' is defined as

$$\tau_{\Sigma}^{\mathrm{sym},t}(\tau') = \mathcal{R}^{\overline{\mathbb{V}}^{\varphi,t}}(\{s^t = [\![E(s)]\!]^{\mathrm{sym}}(t) \mid s \in S\} \cup \Sigma \cup \tau' \cup \{\Lambda^t\}).$$

The symbolic semantics of a LOLA expression is a symbolic representation of the expression, where all offsets operators (also those with zero offset), which do not evaluate to their default value, are replaced by the corresponding instant variables.

The symbolic transformers contain a defining equation for all intermediate stream events of the corresponding instant and the symbolic encoding of the input readings. Additionally the transformer constraint sets include the constraints from the subsequent symbolic transformer (in case $t \neq t_{max}$) and the assumption instant variable for the current instant. However, the instant variables of other instants (which originate from +1 offsets, the subsequent transformer or input readings) are removed from the constraint set by application of a rewriting strategy, which only preserves instant variables of the current and previous instant. An example of symbolic transformers which are computed by the given rules can be found at the end of this subsection after definition 5.13 and for a more complex setting at the end of this chapter in section 5.3.4.

In fact the presented transformers are sound or perfect abstractions of the concrete ones, according to lemma 4.17, depending on whether the utilized rewriting strategy is sound or perfect. This is shown in theorem 5.12.

Theorem 5.12.
Let $\varphi = (I, S, O, E)$ be a LOLA specification and $\Sigma \in \mathbb{S}_{\varphi}^{\mathfrak{A}^{unc}}$ a symbolic encoding of the (potentially) uncertain input steams.

If a sound rewriting strategy $\mathcal{R}$ is used in the transformer computation, then

$$\tau_{\Sigma}^{\varphi,t_{max}} \preccurlyeq \gamma^{tra,t_{max}}(\tau_{\Sigma}^{\mathrm{sym},t_{max}}) \text{ and } \forall t \in \mathbb{T}\backslash\{t_{max}\}.\, \tau_{\Sigma}^{\varphi,t} \precsim \tau_{\Sigma}^{\mathrm{sym},t}$$

If a perfect rewriting strategy $\mathcal{R}$ is used in the transformer computation, then

$$\alpha^{tra,t_{max}}(\tau_{\Sigma}^{\varphi,t_{max}}) \equiv \tau_{\Sigma}^{\mathrm{sym},t_{max}} \text{ and } \forall t \in \mathbb{T}\backslash\{t_{max}\}.\, \tau_{\Sigma}^{\varphi,t} \eqsim \tau_{\Sigma}^{\mathrm{sym},t}.$$

Proof. Assume $I = \{s_1, \ldots, s_n\}$, $S = \{s_{n+1}, \ldots, s_{n+m}\}$. Let $t \in \mathbb{T}\backslash\{0, t_{max}\}$ be an instant.

In the first part of the proof we will show that the symbolic semantics of all defining expressions of a LOLA specification coincide with the adapted LOLA expression semantics from definition 4.10. In the second part we will then argue that the

symbolic transformers which are built on top of the symbolic expression semantics are related to the concrete transformers as stated in the theorem.

Let $C \in \mathbb{S}_{\mathbb{V}}$ be a constraint set over $\mathbb{V} = \{s^{t-1}, s^t, s^{t+1} \mid s \in I \cup S\}$, i.e. the set of instant variables for $t-1$, t and $t+1$. Let C further not contain variables s^t for $s \in S$, i.e. C may not restrict the intermediate streams from the current instant.

Let D be the set of models (see section 2.1.5) of C restricted to $s_1^{t-1}, \ldots, s_{n+m}^{t-1}, s_1^t, \ldots, s_n^t, s_1^{t+1}, \ldots, s_{n+m}^{t+1}$

$$D = [\![C]\!]_{\{s_1^{t-1},\ldots,s_{n+m}^{t-1},s_1^t,\ldots,s_n^t,s_1^{t+1},\ldots,s_{n+m}^{t+1}\}}$$

Thus, set D contains all valuations of input and intermediate streams at previous and next instant and input streams at the current instant which are encoded by C. As usual, for $d \in D$ we denote with $d(s_i^t)$ the entry of s_i^t in d.

We will now show that $\{c \mid c = i \circ [E]^{\varphi}_{b,c,a}, b \circ i \circ a \in D\}$ is exactly the set of models restricted to $s_{n+1}^t, \ldots, s_{n+m}^t$ of $C \cup \{s^t = [\![E(s)]\!]^{\text{sym}}(t) \mid s \in S\}$.

Note, that due to the acyclicity of φ, the streams in S can be put into an order $s_{j_1}, s_{j_2}, \ldots, s_{j_m}$ (with $j_i \in \{n+1, \ldots, n+m\}$) s.t. for instant t the value of s_{j_k} is not dependent on the value of any s_{j_l} with $l > k$.

Observe that, $s_{j_1}^t$ does only depend on the values of the previous and next instant plus inputs from the current instant. Further the definition of $[\![E(s_{j_1})]\!]^{\text{sym}}$ and $[E]^{\varphi}_{b,c,a}$ are substantially equal such that

$$[\![C \cup \{s_{j_1}^t = [\![E(s_{j_1})]\!]^{\text{sym}}(t)]\!]_{\{s_{j_1}^t\}} = \{c(s_{j_1}) \mid c = i \circ [E]^{\varphi}_{b,c,a}, b \circ i \circ a \in D\}.$$

I.e. the possible values for $s_{j_1}^t$ w.r.t. the symbolic constraints are exactly the values for s_{j_1} in all vectors c that satisfy $c = i \circ [E]^{\varphi}_{b,c,a}$.

By induction principle we can apply the same argumentation for all $s \in S$ in the given order and receive

$$[\![C \cup \{s^t = [\![E(s)]\!]^{\text{sym}}(t) \mid s \in S\}]\!]_{\{s^t \mid s \in S\}} = \{c \mid c = i \circ [E]^{\varphi}_{b,c,a}, b \circ i \circ a \in D\}.$$

For $t = 0$ and $t = t_{max}$ we can do analogous reasoning with the expression semantics $[E]^{\varphi,\triangleright}_{c,a}$, $[E]^{\varphi,\triangleleft}_{c,b}$ instead of $[E]^{\varphi}_{b,c,a}$.

In the remainder of this proof we will use these finding in the following way: We will argue that Σ together with the assumption instant variable is a set of constraints (C) which restricts the possible value combinations for $t-1$, t and $t+1$. I.e. it reflects a set D of possible value combinations, on which the concrete transformer computation is also based. By the equalities above we can then conclude that the symbolic transformer is equal to the concrete one for all s^t combinations. Then we additionally restrict D and the possible combinations of current and subsequent values in the symbolic transformer, which is done by incorporation of the subsequent

transformer. Since the values from the previous instant are not restricted in the symbolic semantics and in the concrete transformers, the mentioned equality does finally also hold for all combinations of s^t and s^{t-1}.

We start with $t = t_{max}$:

$$\tau_\Sigma^{\varphi,t_{max}}(b) = \{c \mid c = i \circ [E]_{b,c}^{\varphi,\triangleleft}, i \in \Sigma(t_{max}), c(\Lambda) = \texttt{true}\}$$

As argued, it follows from the equality above and the definition of $\alpha^{tra,t}$ that

$$\alpha^{tra,t_{max}}(\tau_\Sigma^{\varphi,t_{max}}) \equiv_{\{s^t, s^{t-1} \mid s \in S\}} \{s^{t_{max}} = [\![E(s)]\!]^{\text{sym}}(t_{max}) \mid s \in S\} \cup \Sigma \cup \{\Lambda^{t_{max}}\}$$

Now we handle the case for $t \in \mathbb{T}\backslash\{t_{max}\}$ For a symbolic transformer $\tau \in \mathbb{S}_{\varphi,t+1}^{tra}$ observe that we have $a \in \gamma^{tra,t+1}(\tau)(c)$ if and only if

$$\{s_1^t = c(s_1), \ldots, s_{n+m}^t = c(s_{n+m}), s_1^{t+1} = a(s_1), \ldots, s_{n+m}^{t+1} = a(s_{n+m})\} \models \tau.$$

Thus adding τ to the set of symbolic constraints rules out all pairs of a, c which are not compatible with τ. This implies by the same argumentation as for the transformer at the trace end that

$$\alpha^{tra,t}(\tau_\Sigma^{\varphi,t}(\gamma^{tra,t+1}(\tau))) \equiv_{\{s^t, s^{t-1} \mid s \in S\}} \{s^t = [\![E(s)]\!]^{\text{sym}}(t) \mid s \in S\} \cup \Sigma \cup \tau \cup \{\Lambda^t\}.$$

Finally the application of a sound rewriting strategy $\mathcal{R}^{\overline{\mathbb{V}}^{\varphi,t}}$ removes non-relevant variables from the constraint set and preserves the soundness. This leads to

$$\alpha^{tra,t_{max}}(\tau_\Sigma^{\varphi,t_{max}}) \models \tau_\Sigma^{\text{sym},t_{max}} \text{ and } \alpha^{tra,t}(\tau_\Sigma^{\varphi,t}(\gamma^{tra,t+1}(\tau))) \models \tau_\Sigma^{\text{sym},t}(\tau).$$

If perfect rewriting strategies are used the $\models$ relation of the subsequent and current transformer becomes an $\equiv$ relation.

By the Galois condition from definition 2.52 and the fact that $\gamma^{tra,t} \circ \alpha^{tra,t} = \mathbf{id}$ the theorem can finally be concluded from these propositions. □

By lemma 4.17 we get the symbolic transformer semantics for LOLA specifications.

Definition 5.13 (Symbolic transformer semantics; based on [HKLS24])**.**

Let $\varphi = (I, S, O, E)$ be a LOLA specification and $\Sigma \in \mathbb{S}_\varphi^{\mathfrak{A}^{unc}}$ be a symbolic encoding of the (potentially) uncertain input steams.

The *symbolic transformer semantics* of φ is given as

$$\begin{array}{lcl} [\![\varphi]\!]_\Sigma^{str} & : & \mathbb{S}_{\varphi,0}^{tra} \times \cdots \times \mathbb{S}_{\varphi,t_{max}}^{tra} \\ [\![\varphi]\!]_\Sigma^{str} & = & \mu(T \mapsto (\tau_\Sigma^{\text{sym},0}(T(1)), \ldots, \tau_\Sigma^{\text{sym},t_{max}-1}(T(t_{max})), \tau_\Sigma^{\text{sym},t_{max}})) \end{array}$$

Note that in this context the definition of the $\tau_{\Sigma}^{\text{sym},t}$ is monotonic in terms of their subsequent transformer. Thus by lemma 4.17 the symbolic transformer semantics above is a sound or perfect abstraction of the concrete one, depending on the chosen rewriting strategy.

Note also that the definition of the symbolic transformer semantics includes the symbolic encoding of the full input monitoring stream tuples Σ. Due to the assumption that uncertain input readings do not relate events from different instants (section 4.3.2), this is actually not necessary. This is because only the input stream events of the current, previous, and next instant can be referenced by the constraints in the symbolic transformer. Since in our monitoring approach the inputs of the next instant have not yet been received, and those of the previous instant are passed to the transformer as argument, it is sufficient to add only the symbolic encoding of the inputs at the current instant to the symbolic transformer instead of the full Σ.

Example

For a first small example (a more sophisticated one follows in section 5.3.4) consider the LOLA specification φ for the LTL formula $\mathcal{F}(\mathcal{P}q)$ given in figure 5.6.

```
1  input q: 𝔹
2
3  FPq := q[-1|false] ∨ FPq[+1|false]
4  output FPq
```

Figure 5.6.: Example LOLA specification equivalent to LTL property $\mathcal{F}(\mathcal{P}q)$.

Let $\mathbb{T} = \{0, \ldots, 100\}$ be the instant domain. The initial symbolic transformer semantics $[\![\varphi]\!]_{\emptyset}^{str}$ for φ without inputs available (i.e. $\Sigma = \emptyset$) and a perfect rewriting strategy $\mathcal{R}$ can be determined from back to front as follows (note that we skip the Λ constraint as the specification has no assumption).

The transformer $[\![\varphi]\!]_{\emptyset}^{str}(100) = \tau_{\emptyset}^{sym,100}$ is given as

$$\tau_{\emptyset}^{sym,100} = \mathcal{R}^{\overline{\mathbb{V}}^{\varphi,100}}(\{FPq^{100} = q^{99} \vee \texttt{false}\}) = \{FPq^{100} = q^{99}\}$$

Based on this, the transformer for $t = 99$, $\tau^{99} := [\![\varphi]\!]_{\emptyset}^{str}(99) = \tau_{\emptyset}^{sym,99}(\tau_{\emptyset}^{sym,100})$ can be determined as

$$\tau^{99} = \mathcal{R}^{\overline{\mathbb{V}}^{\varphi,99}}(\{FPq^{99} = q^{98} \vee FPq^{100}, FPq^{100} = q^{99}\}) = \{FPq^{99} = q^{98} \vee q^{99}\}$$

and $\tau^{98} := [\![\varphi]\!]_{\emptyset}^{str}(98) = \tau_{\emptyset}^{sym,98}(\tau^{99})$ as

$$\tau^{98} = \mathcal{R}^{\overline{\mathbb{V}}^{\varphi,98}}(\{FPq^{98} = q^{97} \vee FPq^{99}, FPq^{99} = q^{98} \vee q^{99}\}) = \{(q^{97} \vee q^{98}) \rightarrow FPq^{98}\}.$$

We can then apply the same procedure for $t = 97$ and get:

$$\tau^{97} = \mathcal{R}^{\overline{\mathbb{V}}^{\varphi,97}}(\{FPq^{97} = q^{96} \vee FPq^{98}, (q^{97} \vee q^{98}) \rightarrow FPq^{98}\}) = \{(q^{96} \vee q^{97}) \rightarrow FPq^{97}\}$$

Note that the transformers for instants 97 and 98 are equal modulo renaming. By lemma 4.18 we thus have that for all $k \in \{1, \ldots, 97\}$

$$[\![\varphi]\!]^{str}_{\emptyset}(k) = \{(q^{k-1} \vee q^{k}) \rightarrow FPq^{k}\}.$$

The first symbolic transformer for $t = 0$ is further given as

$$[\![\varphi]\!]^{str}_{\emptyset}(0) = \mathcal{R}^{\overline{\mathbb{V}}^{\varphi,0}}(\{FPq^{0} = \texttt{false} \vee FPq^{1}, (q^{0} \vee q^{1}) \rightarrow FPq^{1}\}) = \{q^{0} \rightarrow FPq^{0}\}$$

This example illustrates the concept of symbolic transformers and the corresponding symbolic transformer semantics. Note that the symbolic transformers basically reflect the gist of the LOLA specification with respect to the previous and current instants: `FPq` is true at the current instant, if q is true at the current or previous instant (if existent) and otherwise unknown. In the monitoring algorithm from figure 4.8 (which will in detail be discussed for the symbolic setting in the subsequent sections) we can use this information for perfect recurrent monitoring.

Note, that even in this simple example, where the amount of reasoning about the future is rather limited, the symbolic approach surpasses the capabilities of the traditional LOLA monitor from figure 2.9. Imagine the monitor receives the input that q is true at instant 0. The traditional LOLA monitoring algorithm would not be able to conclude that stream `FPq` is true at this instant. It would just add the equation $Fpq^0 = (\texttt{false} \vee Fpq^1)$ to its internally maintained equations and wait for the next instant to resolve Fpq^1. The symbolic transformer semantics on the other hand directly contains the information $q^0 \rightarrow Fpq^0$ in the symbolic transformer for the first instant $t = 0$.

5.3.2. Symbolic transformer application

The presented symbolic semantics builds the core of the symbolic recurrent monitoring. However, before we discuss this algorithm in detail, there is one more aspect that needs to be addressed. That is, how in the symbolic domain a transformer can be applied to a given configuration set.

In fact, it is sufficient to simply unify the symbolic constraint sets representing transformers and configurations, and then apply a rewriting strategy that removes superfluous instant variables not belonging to the current instant. All stream configurations of the current instant that model the unified constraint set are obviously the results of the transformer application. This is simply because they adhere to the constraints of the transformer and the configuration set it is applied to. Formally for a LOLA specification φ and $t \in \mathbb{T}$ we have

$$\begin{array}{lcl} app^{sym} & : & \mathbb{S}^{tra,\mathfrak{A}}_{\varphi,t} \times \mathbb{S}^{ctx,\mathfrak{A}}_{\varphi,t-1} \rightarrow \mathbb{S}^{ctx,\mathfrak{A}}_{\varphi,t} \\ app^{sym}(\tau, A) & = & \mathcal{R}^{\mathbb{V}^{\varphi,t}}(\tau \cup A) \end{array}$$

where $\mathcal{R}$ is a perfect rewriting strategy for corresponding algebra $\mathfrak{A}$. Clearly, for a sound rewriting strategy the resulting constraint set is an over-approximation.

For example consider a specification with a single stream s of type $\mathbb{R}$ and the transformer for instant $t \in \mathbb{T}$, $\tau(s^{t-1}) = \{(s^{t-1} + 1)\}$, which is symbolically represented by $\{s^t = s^{t-1} + 1\}$. If we apply the transformer to the uncertain configuration $\{(5), (6), (7)\}$, expressing that s is either 5, 6 or 7, which is symbolically encoded as $\{(s^{t-1} = 5) \vee (s^{t-1} = 6) \vee (s^{t-1} = 7)\}$, we get for a perfect rewriting strategy $\mathcal{R}$

$$\mathcal{R}^{\{s^t\}}(\{s^t = s^{t-1} + 1, s^{t-1} = 5 \vee s^{t-1} = 6 \vee s^{t-1} = 7\}) = \{s^t = 6 \vee s^t = 7 \vee s^t = 8\}$$

This matches exactly the configurations $\{(6), (7), (8)\}$ which result from the application in the concrete.

5.3.3. Symbolic monitoring algorithm

We can use the symbolic semantics $[\![\varphi]\!]^{str}_{\Sigma}$ and application function app^{sym} in the algorithm from figure 4.8 to receive a symbolic recurrent LOLA monitor. The adaption of the algorithm can be found in figure 5.7.

```
1 Compute [[φ]]^str_∅;
2 s ← ∅;
3 foreach t ∈ T do
4 |   Read symbolic input encoding for t, Σ^t;
5 |   τ ← τ^{sym,t}_{Σ^t}([[φ]]^str_∅(t + 1));
6 |   s ← app^sym(τ, s);
7 |   Output s;
```

Figure 5.7.: Symbolic instantiation of the monitoring algorithm from figure 4.8.

By corollary 4.19 the algorithm yields a sound or perfect recurrent LOLA monitor (depending on the rewriting strategy), which we conclude in corollary 5.14.

Corollary 5.14.

The algorithm from figure 5.7 for symbolic transformer semantics $[\![\varphi]\!]^{str}_{\Sigma}$ yields

- a sound recurrent LOLA monitor, if a sound rewriting strategy is available for the corresponding algebra and
- a perfect recurrent LOLA monitor, if a perfect rewriting strategy is available for the corresponding algebra.

The symbolic outputs of the monitor can be interpreted by use of an SMT solver which may reveal information about possible events and their combinations.

Concerning trace-length-independence of the monitor (definition 2.13) it has already been outlined that trace-length-independent monitoring can be achieved if the transformers and states (line 5,6 in the algorithm) can be computed in constant time and (related to this) the transformers and monitor states can be stored in constant memory. Additionally we demanded that the initial semantics is efficiently computable due to appearance of a repeating transformer. This is possible if the utilized rewriting strategy is strict-constant, as in this case a repeating element in the computation of the initial symbolic semantics is guaranteed and it has finite size. Yet, it is not the case for a weak-constant rewriting strategy which may still yield an infinite number of different symbolic transformers and thus no repeating transformer may be found. However, if the specification does not contain future offsets and the monitoring does either not involve assumptions or uncertainty (which makes the assumptions useless) the computation of the initial semantics can totally be skipped (see section 4.4.3). In this case a weak-constant rewriting strategy also guarantees a pseudo-constant size of the transformer and configuration set abstractions used in the monitoring algorithm. Still the involved constants (e.g. numbers) in these constraint sets can grow arbitrarily large, yet in a practical setting we assume storage and computation of these constants to be constant, which is reasonable as even huge numbers can be encoded with relatively small memory requirements. Thus, without future reasoning, a weak-constant rewriting strategy is sufficient for trace-length-independence.

Further, if a weak-constant rewriting strategy is utilized for a specification which depends on future estimation and a repeating element cannot be found during computation of the initial fixed point, it is also possible to use a widening strategy which enforces such a repeating element. This however comes at the cost of an over-approximation and implies a sound but imperfect monitor. For the linear algebra and linear arithmetic LOLA fragment a widening strategy is e.g. the one proposed in [CH78]. The idea consists in removing those constraints from the set which do not stabilize or adjusting their coefficients s.t. they are over-approximative. This way the constraint set either repeats or looses at least one constraint per iteration finally leading to an empty constraint set.

Altogether we can summarize the insights from above in the following corollary.

Corollary 5.15.

The symbolic LOLA monitoring algorithm from figure 5.7 yields

- a trace-length-independent recurrent LOLA monitor, if a strict-constant rewriting strategy or a weak-constant rewriting strategy together with widening is available for the corresponding algebra and
- a trace-length-independent recurrent LOLA monitor for specifications without future, if a weak-constant rewriting strategy is available for the corresponding algebra and no assumptions or uncertainties are present.

The results from this corollary are also summarized in the following table, which visualizes for which specifications and rewriting strategies trace-length-independent recurrent monitoring is possible (✓) and for which it is not (✗).

Specification	without future				with future
		+Ass.	+Unc.	+Ass. +Unc.	
Strict-const. rewriting	✓	✓	✓	✓	✓
Weak-const. rew. & widening	✓	✓	✓	✓	✓
Weak-const. rewriting	✓	✓	✓	✗	✗

Corollary 5.15 together with the results from section 5.2 finally implies the following findings on specific LOLA fragments and logics:

- The symbolic monitoring approach is trace-length-independent and perfect for the boolean LOLA fragment $\text{Lola}_{\mathbb{B}}$ with uncertainty and assumptions and thus also for LTL specifications translated to LOLA.
- The symbolic monitoring approach is trace-length-independent and perfect for the linear algebra LOLA fragment $\text{Lola}_{\mathcal{LA}}$ restricted to past-only specifications with either uncertainty or assumptions.
- The symbolic monitoring approach is trace-length-independent and sound for the linear algebra and linear arithmetic LOLA fragments $\text{Lola}_{\mathcal{LA}}$ and $\text{Lola}_{\mathcal{LRA}}$ with uncertainty and assumptions and thus also for M(I)TL and STL specifications translated to LOLA[1].

In the next section the introduced symbolic, recurrent monitoring approach is presented again in its entirety by means of an example.

5.3.4. Overall example

As final example for the symbolic LOLA monitoring algorithm we consider the specification $\varphi = (I, S, O, E)$ depicted in figure 5.8, which shows similarities to the one from the introduction chapter. As in chapter 1, the specification checks whether an input stream `vel` (for velocity), which indicates the speed of a vehicle, e.g. a robot, exceeds a certain value (3). Stream `err` is true if `vel` is greater or equal 3 at the current or any subsequent instant. Stream `diff` determines the difference between the current and last speed. The specification also contains an assumption, encoded in stream Λ, which rules that the value of `diff` between two steps (i.e. the second derivation of stream `vel`) does not exceed 1 in absolute value.

[1] Note that depending on the chosen translation from section 3.4.2, the specifications may contain `ite` operators which are actually not allowed in linear real arithmetic but can be handled there by introducing helper constraints (see [KLS22a]).

```
1    input vel: ℝ^≥0
2
3    def diff = vel[now] - vel[-1|0]
4    def err = err[+1|false] ∨ (vel[now] ≥ 3)
5    output err
6
7    def Λ = |diff[now] - diff[-1|0]| ≤ 1
```

Figure 5.8.: Example LOLA specification with future references.

In difference to the specification from chapter 1 the speed limit was adjusted from 5 to 3 to make the fixed point convergence faster for sake of the full presentation of the example. Furthermore in this example stream `err` is true if an error occurs at the current or a future trace position. In the example from the introduction it was true if an error occurred at the current or a past position. This adjustment was made to illustrate the handling of future offsets by the algorithm, but the kind of anticipatory reasoning in both examples is comparable. As time domain we assume $\mathbb{T} = \{0, \ldots, 99\}$, representing a time domain of non-trivial size. For better readability we will also abbreviate the involved streams in the constraint sets by their initial letters (i.e. $I = \{v\}$, $S = \{d, e, \Lambda\}$).

The example is in the linear arithmetic LOLA fragment (though we do not directly allow the absolute value operator there, the corresponding constraints in this example could be expressed by a conjunction of two inequalities). For the monitoring we will apply the perfect rewriting strategy from figure 5.4 (which will lead to a repeating symbolic transformer in this concrete example). To optimize readability, the particular rewritings are not performed step by step, but simplified versions of the resulting constraint sets are given and their correctness is argued.

Computation of the initial symbolic transformer semantics

Following the algorithm from figure 5.7 we start with the computation of $[\![\varphi]\!]^{str}_{\emptyset}$. In this example we abbreviate the transformers of the initial semantics as $\tau^0, \ldots, \tau^{99}$. For $\tau^{99} = \tau^{sym,99}_{\emptyset}$ we instantiate the specification symbolically for the last instant and receive

$$\begin{aligned}
\tau^{99} &\equiv \mathcal{R}^{\overline{\mathbb{V}}^{\varphi,99}}_{\mathcal{LRA}}(\{s^{99} = [\![E(s)]\!]^{\mathrm{sym}}(99) \mid s \in S\} \cup \emptyset \cup \{\Lambda^{99}\}) \\
&\equiv \mathcal{R}^{\overline{\mathbb{V}}^{\varphi,99}}_{\mathcal{LRA}}(\{d^{99} = v^{99} - v^{98}, e^{99} = (v^{99} \geq 3), \Lambda^{99} = (|d^{99} - d^{98}| \leq 1), \Lambda^{99}\}) \\
&\equiv \{d^{99} = v^{99} - v^{98}, e^{99} = (v^{99} \geq 3), \Lambda^{99} = (|d^{99} - d^{98}| \leq 1), \Lambda^{99}\}.
\end{aligned}$$

Note that at this first step no non-relevant variables are contained in the constraint set and the application of the rewriting $\mathcal{R}^{\overline{\mathbb{V}}^{\varphi,99}}_{\mathcal{LRA}}$ has no effect. We can proceed with

computation of $\tau^{98} = \tau_{\emptyset}^{sym,98}(\tau^{99})$. After unrolling the specification symbolically we get the symbolic transformer

$$\begin{aligned}\tau^{98} \;\equiv\; & \mathcal{R}_{\mathcal{LRA}}^{\overline{\mathbb{V}}^{\varphi},98}(\{s^{98} = [\![E(s)]\!]^{\mathrm{sym}}(98) \mid s \in S\} \cup \emptyset \cup \{\Lambda^{98}\} \cup \tau^{99}) \\ \equiv\; & \mathcal{R}_{\mathcal{LRA}}^{\overline{\mathbb{V}}^{\varphi},98}(\{d^{99} = v^{99} - v^{98}, e^{99} = (v^{99} \geq 3), \Lambda^{99} = (|d^{99} - d^{98}| \leq 1), \Lambda^{99}, \\ & \qquad d^{98} = v^{98} - v^{97}, e^{98} = e^{99} \vee (v^{98} \geq 3), \Lambda^{98} = (|d^{98} - d^{97}| \leq 1), \Lambda^{98}\}).\end{aligned}$$

Now we can apply the rewriting strategy and eliminate all instant variables for instant 99 from these constraints. This leads to a symbolic transformer equivalent to the constraint set

$$\begin{aligned}\tau^{98} \;\equiv\; & \{d^{98} = v^{98} - v^{97}, ((v^{98} + d^{98} \geq 4) \vee (v^{98} \geq 3)) \rightarrow e^{98}, \\ & \Lambda^{98} = (|d^{98} - d^{97}| \leq 1), \Lambda^{98}\}.\end{aligned}$$

The sub-constraint $(v^{98} + d^{98} \geq 4)$ results from the rewriting of $(|d^{99} - d^{98}| \leq 1)$ which (together with other constraints) is equal to $(|v^{99} - v^{98} - d^{98}| \leq 1)$. It follows from this constraint that $(v^{98} + d^{98} \geq 4)$ implies $v^{99} \geq 3$ and consequently e^{99} and e^{98}. Thus, the rewriting delivers us the constraint $((v^{98}+d^{98} \geq 4)\vee(v^{98} \geq 3)) \rightarrow e^{98}$. As mentioned, the rewriting strategy from figure 5.4 would produce an equivalent constraint set as the one above in distributive normal form, but for the sake of the example we use the simplified version here.

The initial fixed point computation then proceeds in a similar fashion and yields

$$\begin{aligned}\tau^{97} \;\equiv\; & \{d^{97} = v^{97} - v^{96}, ((v^{97} + 2d^{97} \geq 6) \vee (v^{97} + d^{97} \geq 4) \vee (v^{97} \geq 3)) \rightarrow e^{97}, \\ & \Lambda^{97} = (|d^{97} - d^{96}| \leq 1), \Lambda^{97}\}\end{aligned}$$

and

$$\begin{aligned}\tau^{96} \;\equiv\; & \{d^{96} = v^{96} - v^{95}, \\ & ((v^{96} + 3d^{96} \geq 9) \vee (v^{96} + 2d^{96} \geq 6) \vee (v^{96} + d^{96} \geq 4) \vee (v^{96} \geq 3)) \rightarrow e^{96}, \\ & \Lambda^{96} = (|d^{96} - d^{95}| \leq 1), \Lambda^{96}\}.\end{aligned}$$

After these four computation steps a repeating transformer entry is found, because

$$\tau^{96} \;\equiv\; \tau^{97}[97\backslash 96]$$

where $[m\backslash n]$ represents a renaming of all instant variables in the symbolic transformer from instant m to n and $m-1$ to $n-1$. This is because the constraint $(v^{96} + 3d^{96} \geq 9)$ is already implied by $(v^{96} + 2d^{96} \geq 6)$ for all non-negative v^{96} (note that `vel` is from $\mathbb{R}^{\geq 0}$). Thus we have

$$\tau^{k} \equiv \tau^{97}[97\backslash k]$$

for all $k \in \{1, 2, \ldots, 96\}$, and so $[\![\varphi]\!]_{\emptyset}^{str}$ is determined except for the first transformer entry, which is based on default values for -1 offsets and thus different. As mentioned,

the first transformer in the initial semantics is actually never used in the monitoring algorithm and its determination can therefore be skipped.

Symbolic Monitoring

Now the actual online monitoring of the specification in lines 3 to 7 of figure 5.7 starts. Assume the monitor receives the uncertain input trace

$$\langle 0, 1, [2.5, 3], \dots \rangle$$

letter by letter, where $[2.5, 3]$ denotes the uncertain interval between 2.5 and 3. We go through the first three monitoring steps.

The first input reading can symbolically be encoded as $\Sigma^0 = \{v^0 = 0\}$. In line 5 the monitoring algorithm proceeds with the recomputation of the symbolic transformer of the current instant, $\tau = \tau^{sym,0}_{\{v^0=0\}}([\![\varphi]\!]^{str}_{\emptyset}(1)) = \tau^{sym,0}_{\{v^0=0\}}(\tau^1)$.

$$\begin{aligned}
\tau \;&\equiv\; \mathcal{R}^{\overline{\mathbb{V}}^{\varphi,0}}_{\mathcal{LRA}}(\{s^0 = [\![E(s)]\!]^{\mathrm{sym}}(0) \mid s \in S\} \cup \{v^0 = 0\} \cup \{\Lambda^0\} \cup \tau^1) \\
&\equiv\; \mathcal{R}^{\overline{\mathbb{V}}^{\varphi,0}}_{\mathcal{LRA}}(\{d^0 = v^0, e^0 = e^1 \vee (v^0 \geq 3), \Lambda^0 = (|d^0| \leq 1), v^0 = 0, \Lambda^0, d^1 = v^1 - v^0, \\
&\qquad ((v^1 + 2d^1 \geq 6) \vee (v^1 + d^1 \geq 4) \vee (v^1 \geq 3)) \rightarrow e^1, \Lambda^1 = (|d^1 - d^0| \leq 1), \Lambda^1\}) \\
&\equiv\; \{v^0 = 0, d^0 = 0, \Lambda^0\}
\end{aligned}$$

In line 6, the monitor then applies the transformer to the current state s (which is initially $\emptyset$) which leads to the monitoring state

$$s \;=\; app^{sym}(\tau, s) \;\equiv\; \mathcal{R}^{\overline{\mathbb{V}}^{\varphi,0}}_{\mathcal{LRA}}(\tau) \;\equiv\; \{v^0 = 0, d^0 = 0, \Lambda^0\}.$$

This state is used as monitor output. As mentioned, it may further be interpreted with help of an SMT solver, e.g. to receive the set of possible values for stream `err`. In our case this would yield $e^0 \in \{\texttt{true}, \texttt{false}\}$, as both values are conformant with the computed constraint set, which contains no constraints about e^0.

The monitor then proceeds in line 4 with receiving the second input value of stream `vel` which is 1, i.e. $\Sigma^1 = \{v^1 = 1\}$ (recall that the transformer computation only requires the input readings of the current events). The transformer $\tau = \tau^{sym,1}_{\{v^1=1\}}(\tau^2)$ can be determined as

$$
\begin{aligned}
\tau \;&\equiv\; \mathcal{R}^{\overline{\mathbb{V}}^{\varphi,1}}_{\mathcal{LRA}}(\{s^1 = [\![E(s)]\!]^{\mathrm{sym}}(1) \mid s \in S\} \cup \{v^1 = 1\} \cup \{\Lambda^1\} \cup \tau^2) \\
&\equiv\; \mathcal{R}^{\overline{\mathbb{V}}^{\varphi,1}}_{\mathcal{LRA}}(\{d^1 = v^1 - v^0, e^1 = e^2 \vee (v^1 \geq 3), \Lambda^1 = (|d^1 - d^0| \leq 1), v^1 = 1, \Lambda^1, \\
&\quad\; d^2 = v^2 - v^1, ((v^2 + 2d^2 \geq 6) \vee (v^2 + d^2 \geq 4) \vee (v^2 \geq 3)) \rightarrow e^2, \\
&\quad\; \Lambda^2 = (|d^2 - d^1| \leq 1), \Lambda^2\}) \\
&\equiv\; \{v^1 = 1, d^1 = 1 - v^0, ((v^1 + 2d^1 \geq 6) \vee (v^1 + d^1 \geq 4) \vee (v^1 \geq 3)) \rightarrow e^1, \\
&\quad\; |d^1 - d^0| \leq 1, \Lambda^1\} \\
&\equiv\; \{v^1 = 1, d^1 = 1 - v^0, ((3 - 2v^0 \geq 6) \vee (2 - v^0 \geq 4)) \rightarrow e^1, |d^1 - d^0| \leq 1, \Lambda^1\}
\end{aligned}
$$

If we apply this transformer by adding the constraints $\{v^0 = 0, d^0 = 0, \Lambda^0\}$ we get the new monitor state

$$
s \;=\; app^{sym}(\tau, s) \;\equiv\; \mathcal{R}^{\mathbb{V}^{\varphi,1}}_{\mathcal{LRA}}(\tau \cup s) \;\equiv\; \{v^1 = 1, d^1 = 1, \Lambda^1\}.
$$

which again yields $e^1 \in \{\texttt{true}, \texttt{false}\}$ as output for stream `err`.

If we finally receive the interval $[2.5, 3]$ as input for stream `vel` at instant 2, symbolically expressed as $\Sigma^2 = \{2.5 \leq v^2 \leq 3\}$, we compute the symbolic transformer $\tau = \tau^{sym,2}_{\{2.5 \leq v^2 \leq 3\}}(\tau^3)$ as

$$
\begin{aligned}
\tau \;&\equiv\; \mathcal{R}^{\overline{\mathbb{V}}^{\varphi,2}}_{\mathcal{LRA}}(\{s^2 = [\![E(s)]\!]^{\mathrm{sym}}(2) \mid s \in S\} \cup \{2.5 \leq v^2 \leq 3\} \cup \{\Lambda^2\} \cup \tau^3) \\
&\equiv\; \mathcal{R}^{\overline{\mathbb{V}}^{\varphi,2}}_{\mathcal{LRA}}(\{d^2 = v^2 - v^1, e^2 = e^3 \vee (v^2 \geq 3), \Lambda^2 = (|d^2 - d^1| \leq 1), 2.5 \leq v^2 \leq 3 \\
&\quad\; \Lambda^2, d^3 = v^3 - v^2, ((v^3 + 2d^3 \geq 6) \vee (v^3 + d^3 \geq 4) \vee (v^3 \geq 3)) \rightarrow e^3, \\
&\quad\; \Lambda^3 = (|d^3 - d^2| \leq 1), \Lambda^3\}) \\
&\equiv\; \{2.5 \leq v^2 \leq 3, d^2 = v^2 - v^1, |d^2 - d^1| \leq 1, \Lambda^2, \\
&\quad\; ((3v^2 - 2v^1 \geq 6) \vee (2v^2 - v^1 \geq 4) \vee (v^2 = 3)) \rightarrow e^2\}
\end{aligned}
$$

which together with the previous monitoring state $\{v^1 = 1, d^1 = 1, \Lambda^1\}$ leads to

$$
s \;=\; app^{sym}(\tau, s) \;\equiv\; \mathcal{R}^{\mathbb{V}^{\varphi,2}}_{\mathcal{LRA}}(T \cup s) \;\equiv\; \{2.5 \leq v^2 \leq 3, d^2 = v^2 - 1, e^2, \Lambda^2\}.
$$

This output is anticipatory as $e^2 = \texttt{true}$ can be concluded even though v^2 is not necessarily greater or equal 3. The cause for this is that in the transformer constraints, $2v^2 - v^1 \geq 4$ implies e^2, which together with $v^1 = 1$ is equal to $(2v^2 - 1 \geq 4)$. The constraint $(2v^2 - 1 \geq 4) \equiv (v^2 \geq 2.5)$ however is implied by $2.5 \leq v^2 \leq 3$.

Intuitively the reason that we can conclude this output is the following. At the current instant stream `diff` lies between 1.5 and 2. According to the assumption, stream `diff` at the next instant (d^3) must be in the interval between 0.5 and 3 and v^3 must consequently be at least 3. This is because $d^3 = v^3 - v^2$, i.e. $v^3 = v^2 + d^3$

and with $2.5 \leq v^2 \leq 3$ and $0.5 \leq d^3 \leq 2.5$ we have $v^3 \geq 3$. So the actual error condition `vel` ≥ 3 is satisfied at instant 3.

Altogether this example shows how the symbolic recurrent monitoring approach can be used to perfectly monitor LOLA specifications without the unrolling of the full specification. The following subsection contains some final remarks on the presented symbolic monitoring technique.

5.3.5. Remarks

The symbolic recurrent LOLA monitoring algorithm presented in this chapter shows parallels to the traditional anticipatory LTL_3 monitoring approach, described in section 2.2.2 and may be considered as a generalization of it. The analogy is the following.

The LTL_3 monitor operates on states, which encode sub-formulas of the original LTL formula that still have to be satisfied. As such, these states can be seen as concise encodings of the trace which has been received so far with respect to the information which is relevant for the specification. The outputs of the monitor rely on an emptiness per state check (done during monitor synthesis) which determines if acceptance from the current state, i.e. sub-formulas, in the negated and non-negated Büchi automaton is still possible at all by considering all possible future input extensions.

The recurrent monitoring approach presented in this chapter also develops constraint sets (the monitor states) of finite size, capturing all information from the received inputs with respect to the specification. These monitoring states encode all stream value combinations that are possible under the previously received inputs. Furthermore constraint sets about the future are determined (before monitoring) to give information about possible values of future references by considering all possible future inputs. Unlike the LTL_3 monitor however, the presented recurrent monitoring algorithm is focused on monitoring of pointwise properties, like the past LTL monitor from section 2.2.2. It can thus be considered as a combination of both techniques.

Altogether, this chapter provides a theory about symbolic synchronous monitoring of finite fixed-size traces under instant-immanent uncertainty, assumptions and with consideration of the future. This is because, as discussed in chapter 3, LOLA can serve as a general formalism for pointwise properties over a discrete time domain. Corollary 5.14 shows that if for the algebra which encodes the specification (including assumptions) and uncertain input readings, there is a perfect, strict-constant rewriting strategy then also a perfect recurrent monitor exists and how it can be constructed. The same holds for sound monitoring which requires a sound, strict-constant rewriting strategy or a weak-constant rewriting strategy together with a widening technique. If the specification does only contain past references and either

no assumptions or no uncertainty is present, also a weak-constant rewriting strategy is sufficient. As a consequence the problem of building a recurrent monitor gets reduced to finding an appropriate rewriting strategy.

As for several algebras there is no strict-constant, perfect rewriting (most prominently linear arithmetic), these sound rewriting strategies also play an important role when it comes to building such monitors in practice. We defined sound rewriting strategies generally as those which over-approximate the actual configurations of the involved variables. The quality of these strategies however, i.e. how many additional value-combinations are introduced by rewriting, also plays a significant role. Therefore it may make sense to investigate appropriate quality measures on sound rewriting strategies and likewise widening operators, which we leave for future work here. However, note that such a measure would also be useful to build a cascade of more and more coarse rewriting and widening strategies which are applied after each other until a stabilizing approximation of the initial semantics is found.

Also some perfect, strict-constant rewriting strategies, e.g. the one for the boolean fragment in section 5.2 might suffer from efficiency issues in practice. This is because their runtime and resulting constraint set is in the worst case exponential in the number of streams in the specification. For practical application it might thus in general make sense to use sound (i.e. over-approximating) strategies with run time benefits, in case the computation of the initial semantics does not terminate in reasonable time.

Further, it seems natural to ask whether the presented approach can be extended to infinite instant domains and to stream runtime verification languages other than LOLA. This question is also left for future work here. Yet, the definition of a fixed point based monitoring semantics as done in section 4.2 as basis of the algorithm can be considered a first step towards support of other SRV formalisms (c.f. [Sch24]), including those with richer instant domains. Also, the recurrent monitoring approach discussed in this chapter does not necessitate a full unrolling of the specification along the trace, as the computation of the initial semantics terminates as soon as equivalent constraint sets for two transformers are recognized. This property principally makes the theory suitable for an extension to formalisms over instant domains with an unbounded number of elements. One major problem however remains for infinite time domains (i.e. those without a maximal timestamp): The computation of the initial semantics as presented in this thesis starts at the end of the trace with the default values of future offsets. In case of infinite instant domains, however, there is no end of the trace. Therefore, an adapted strategy for computing the initial semantics is required. How this can be done is highly dependent on the definition of the concrete infinite semantics.

5.4. Summary

In this and the previous chapter, recurrent monitoring of LOLA specifications under uncertainties and assumptions has been studied. While the previous chapter introduced a generic theory based on abstract interpretation, this chapter dealt with a concrete, symbolic monitoring approach based on this theory. As outlined, the presented symbolic monitoring algorithm can also serve as a general framework for sound and perfect synchronous monitoring over discrete time domains, since it essentially reduces the monitoring problem to finding perfect or sound rewriting strategies for the induced algebra of the LOLA specification.

In the following section, we discuss a prototypical implementation of the symbolic recurrent monitoring approach for linear arithmetic specifications. The implementation is evaluated in three practical application scenarios, and its practicality and weaknesses are examined.

6

Application and evaluation

In this chapter, we will discuss a proof-of-concept implementation of the symbolic LOLA monitoring approach that was theoretically discussed in the previous chapter. The goal is to provide some first insights into the capabilities of symbolic LOLA monitoring, to show where the potential shortcomings lie, and for which kinds of applications the approach may be suitable.

The tool which is presented and evaluated in this chapter has been developed at the Institute for Software Engineering and Programming Languages of University of Lübeck alongside the publications [KLS22a, HKLS24].

The chapter begins with a brief overview of the implementations used. This covers two variants of the symbolic algorithm from this thesis and another simple LOLA backend for comparison. In the second part the results of the tool's usage in three different case studies are presented and discussed.

6.1. Implementation

The tool we use for the evaluation of the previously presented monitoring approach is written in Scala[1] and provides three backends for the evaluation of LOLA specifications. As mentioned, two of these backends are symbolic engines following the approach from the previous chapter. The third backend follows a different monitoring strategy and will be used for comparison and grading of the symbolic approach. Specifically the backends are:

- **Interval backend:** *Past-only interval backend with uncertainties.* This implementation basically follows the standard LOLA monitoring algorithm from

[1] `https://www.scala-lang.org/`

figure 2.9 restricted to past-only specifications, but also supports the presence of uncertainties (yet no assumptions). Thereby uncertainties can be fully unknown events or numeric intervals. For the computations, the backend utilizes simple, over-approximating set and interval semantics comparable to [LSS+19]. That is, instead of concrete stream events it computes sets (for boolean streams) and intervals (for real-valued streams) which contain and possibly over-approximate all potential event values. This implementation is used for qualitative and quantitative comparison with the symbolic approach which is in difference to this backend able to encode relations among various stream events and to support assumptions.

- **Past-only symbolic backend:** *Sound, past-only symbolic backend with uncertainties and assumptions.* This symbolic implementation from [KLS22a] follows in principle the approach discussed in this thesis but is not anticipatory. This includes that it only considers an unrolling of the assumption (if available) up to the current instant but not for the future. For the linear arithmetic LOLA fragment it uses a sound but imperfect weak-constant rewriting strategy which is an adaption of the one from figure 5.5. Due to these restrictions the approach can use an optimized internal representation of the symbolic constraints. Instead of a single constraint set, it organizes the maintained symbolic constraints in a map data structure where each stream identifier is mapped to a corresponding symbolic expression describing its current value. These expressions are then used for computation of the subsequent symbolic map entries. Additional constraints that might arise during rewriting are appended to the symbolic representation of the assumption stream. The outlined technique turns out to be way faster than the plain symbolic approach. However, the sound but imperfect rewriting strategy also makes it less precise. Furthermore the approach, as it is implemented in this backend, cannot easily be extended to support future reasoning, as the union of the current monitoring state and the computed symbolic transformer introduces cyclic dependencies among instant variables. Also the (perfect) rewriting strategies as proposed in the previous chapter cannot directly be applied to this approach, as they do not yield a single symbolic constraint for every stream. Nevertheless, this backend provides insight into the capabilities of an optimized version of the symbolic monitoring approach for past-only reasoning. The incorporation of these optimizations in a perfect backend with future reasoning is left for future work. As engine for symbolic reasoning the backend uses Z3 [dMB08] with its Java bindings[2].

- **Full symbolic backend:** *Perfect past and future symbolic backend with uncertainties and assumptions.* This symbolic backend is the direct implementation of the full symbolic approach from section 5.3 with uncertainties, assumptions and future reasoning. As symbolic engine this approach also utilizes Z3.

[2] `https://github.com/Z3Prover/Z3`

For the computation of the symbolic transformers it uses quantifier elimination as rewriting strategy to achieve perfectness. The backend also contains a widening technique in case the perfect rewriting does not lead to a repeating transformer or the monitor state gets too large. However, in the case studies from this evaluation where this backend was used, the application of widening was not necessary. In general this implementation follows quite strictly the approach outlined in the previous chapter and contains only a couple of obvious optimizations. Thus, the evaluation of this backend reflects directly the practicality of the basic symbolic monitoring approach. With further complex engineering and application-specific optimizations, the performance of the tool can probably be increased significantly.

All three backends share a common ANTLR[3]-based frontend which parses LOLA specifications into an abstract syntax tree (AST), as well as a common input/output handling and logging facility. This architecture enables an easy and precise comparison of the different monitoring approaches. The LOLA specification format is a direct realization of definition 2.36. Supported data types are booleans and floating point numbers (as substitution for reals) with the common operations (`&&`, `||`, `!`, `+`, `*` ...). There is no restriction to a special LOLA fragment (like linear arithmetic) per se, the two symbolic backends however are restricted to linear arithmetic or subsets and refuse execution if the input specification does not belong to the supported fragment.

Besides the standard LOLA operators the tool also supports a macro syntax for the quick definition of a number of similar streams. These macros are unfolded during the monitor synthesis. Likewise the specification is flattened – as required by the symbolic monitoring algorithm – such that only -1 and +1 offsets are contained in the final specification. For each output stream the tool casts (independent of the chosen backend) the outputs `true`, `false` and ? in case of boolean streams and numeric values or intervals in case of real-valued streams at each instant.

6.2. Evaluation

With regard to the evaluation of the symbolic monitoring approach from the previous chapter, we concentrate on the following two aspects of interest.

- How does the symbolic algorithm compare to other existing monitoring approaches under uncertainty and assumptions? This includes a qualitative comparison, i.e. under which circumstances which approach provides more accurate results, and a quantitative comparison in terms of runtime and memory required for monitor execution, but also for monitor synthesis.

[3] `https://www.antlr.org/`

- What is the qualitative and quantitative difference when uncertainties and assumptions are present or not?

For an exhaustive evaluation of the symbolic approach, it would be necessary to check all these aspects in different combinations for different LOLA fragments and for different synthetic and realistic scenarios. Even then, however, the problem arises that the implementation of all backends presented here is of prototypical nature and leaves potential for optimization in some places. Therefore, the implementation cannot demonstrate the maximum efficiency of the symbolic approach per se. Consequently, the following is not intended to be a complete evaluation of the symbolic monitoring approach or tool, but rather a presentation of three case studies that provide first insights into the general practicality and potential pitfalls of symbolic monitoring.

6.2.1. Case studies and evaluation results

The three presented case studies restrict to the linear arithmetic fragment, as a lot of properties with practical relevance can be formulated in there. In terms of quantitative analysis we only measure the runtime of the monitor per instant and the amount of time required for synthesis (i.e. computation of the initial semantics). Memory consumption is hard to measure for programs running on the Java virtual machine, mainly because garbage collection is subject to random influences. For all case studies, however, it can be confirmed that the memory consumption did not continuously grow during the monitoring process and is thus trace-length-independent.

For the comparison with other tools we restrict to the mentioned interval backend. This is mainly because no other implementations which perform anticipatory LOLA monitoring under uncertainties and assumptions for the linear arithmetic LOLA fragment exist. The comparison to similar tools, which would require a transformation of the specification to another formalism, is little meaningful. Especially because these translations are often not even possible, when other formalisms are less expressive than linear arithmetic LOLA. Besides, interval computation can be considered as standard imperfect monitoring approach for handling uncertainty in the linear arithmetic fragment. However, since the interval approach is limited to past specifications, we can only conduct the comparison for the first case study, which does not involve future reasoning.

All experiments presented in the following are taken from the publications [KLS22a, HKLS24], but extended in some points (which is noted in the text). For the measurements in this thesis, they were rerun on a Linux machine (kernel version 6.5.3) with an Intel i7-8550U CPU running at 1.80 GHz base frequency and with 8 GB of RAM. All measured values can be found in appendix B.

Case Study 1. ECG signal analysis: Past-only monitoring

The first case study (also presented in [KLS22a]) is the analysis of an electrocardiogram (ECG) signal of the human heartbeat by means of a LOLA monitor. The specification is a simplified version of the one from [GS21a]. The input data (also taken and adapted from [GS21a]) consists of a timed sequence of ECG measurements. The LOLA specification is used to detect heartbeats in the trace, i.e. locations where the value of the smoothed input signal surpasses a certain threshold and is the greatest of the 100 surrounding data values. Therefore it has a boolean output stream which is true if the peak of a heartbeat is present at the current instant. To determine the values of this stream, the specification first smooths the input stream by applying a sliding average filter of size 15. Further the technique from section 4.4 is applied to mimic k-offset monitoring with $k = 50$, s.t. the whole window around a certain position is available when the monitoring output for a particular instant is cast. This way no future reasoning is necessary for the monitoring task. In particular the value of the last 100 (smoothed) input values is maintained in 100 streams. In additional streams it is checked if the 50^{th} last value is greater than all others. This can effectively be specified by usage of the aforementioned macro syntax. The sliding average mentioned above is implemented in such a way that a stream `convoluted` aggregates the 15 last values by adding the current input value and subtracting the 15^{th} last input again, i.e.

```
convolved = convolved[-1|0] + squared[now] - squared[-15|0]
```

where input stream `squared` denotes the (squared) input ECG signal. For some of the following experiments, also an assumption was included in the specification, stipulating that two heartbeats must be a certain number of steps apart.

For the qualitative analysis of the approach the capabilities of the past-only symbolic backend, which is specialized for this kind of specification, were compared to those of the interval backend. As input, a preprocessed ECG signal (the value was squared which could not be done by the LOLA specification since it wouldn't have been in the linear arithmetic fragment then) with 14 heartbeats was used where uncertainty was manually injected. For the first group of experiments noise was added to the trace. A certain percentage of input values was replaced by an interval of a given percent range around the actual value. For further experiments five gaps (bursts) of size 6 to 19 events at random positions have been inserted into the trace where the input values were changed to completely unknown.

First, the specification has been evaluated with both approaches for a trace where 20% of the values were exchanged by ranges of ±20% around the actual values. The symbolic approach was still able to detect all heartbeats with certainty. The interval monitor however delivered significantly worse results. Around the first two heartbeats it gave the output ?, indicating that there may be a peak. Later it did yield more and more ? outputs even far off from heartbeats until after about a third

of the trace it did permanently cast ? for any trace position. The resulting traces with certainly (green) and maybe (yellow) detected heartbeats are visualized in the upper part of figure 6.1.

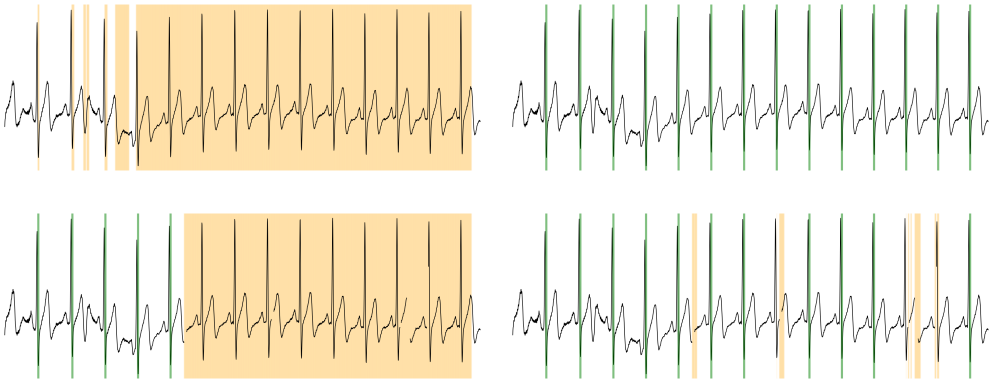


Figure 6.1.: Comparison of the interval (left) and past-only symbolic backend (right) for a trace with blurred input values (above) and bursts of full uncertainty (below). Yellow: Maybe peaks (? output). Green: Certainly detected peaks.

The main reason for the different qualities of the approaches lies in the stream `convoluted` (see above). If an event on stream `squared` is uncertain in the symbolic approach it is represented by an instant variable in the symbolic expression for `convoluted` and subtracted again 15 instants later. This causes the variable to be removed again from the symbolic expression, and certainty about the stream value may be regained. In the interval approach however, the value of stream `convolved` becomes uncertain (i.e. a non-singular interval) once stream `squared` is uncertain. When 15 instants later the uncertain value of `squared` is subtracted again from `convolved`, the value of the stream stays uncertain. This is because the information that the subtracted uncertain interval is exactly the same as the one added 15 steps before is lost in the interval calculation.

For the five uncertainty bursts the result is comparable (see lower part of figure 6.1). While the symbolic approach yields some ? verdicts around the gaps, it recovers and detects the heartbeats between the gaps again with certainty. The interval approach looses complete track after the first gap as stream `convolved` and several derived streams get fully unknown from then on.

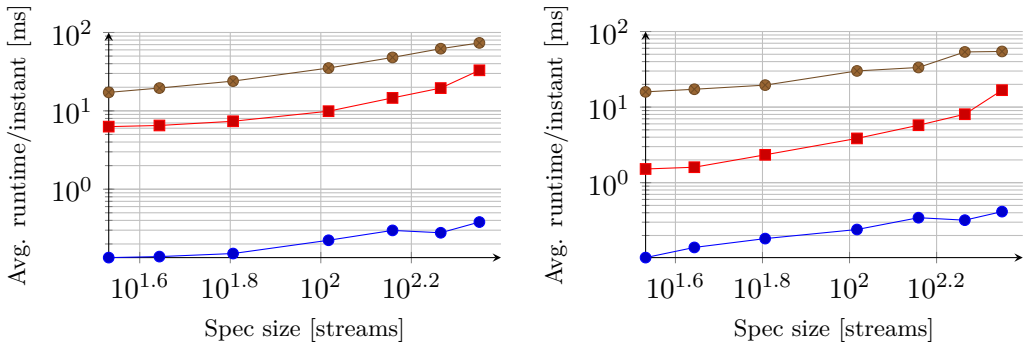
Another interesting aspect in this matter is that by adding the mentioned assumption to the specification, the symbolic approach is able to detect some more heartbeats with certainty. This is because it can conclude that some gaps actually cannot contain a heartbeat, as the last certain heartbeat has been to close. However, if the assumption is not handled as assumption but as filter, i.e. the specification only detects peaks when the assumption is satisfied but does not *require* it to be satisfied,

the overall result is worse. The reason for this is that treating the condition that heartbeats can only occur at a certain distance as an assumption allows advanced reasoning about the values of intermediate streams. Especially a range of the last possible timestamp with a heartbeat can be concluded. Such reasoning is in general not possible if the assumption condition is just used as a filter.

Further, the runtime of the two monitoring approaches has been compared. Therefore variations of the specification have been analyzed where the size of the sliding window in which a heartbeat is detected (originally 100) was reduced, to measure the runtime for different specification sizes. The following table shows the size of the flattened specification (in terms of the streams involved) relative to the length of the window.

Spec. with window of length	5	10	20	40	60	80	100
Number of streams in flattened spec.	34	44	64	104	144	184	224

The runtimes for both approaches (symbolic and interval) per received input event w.r.t. the specification size are shown in figure 6.2 for (a) a trace where 5 % of all input events have been replaced by uncertain intervals ±20% around the exact value and (b) the mentioned trace with five gaps. Additionally the symbolic approach under the presence of the mentioned assumption is depicted. Note that the handling of assumptions in the specification increases the runtime notably strong. This is because to support assumptions in a meaningful way the optimaization capability of Z3 had to be switched on. This was needed to infer bounds of real variables w.r.t. the assumptions during the imperfect rewriting strategy. Without assumption a simpler reasoning about the bounds was sufficient.



(a) Noisy trace

(b) Trace with bursts

Figure 6.2.: Average runtime per instant in ms of the interval (●) vs. symbolic (■) vs. symbolic & assumption (●) approach for input traces with different uncertainty types and different specification sizes (i.e. window sizes).

The figure shows that the average computation time of the interval approach is (not surprisingly) significantly smaller than that of symbolic reasoning. The symbolic

approach without assumptions is up to 90 times slower, depending on the size of the specification. Adding assumptions increases the runtime again by factor 2 to 10.

For the noisy input, the runtime per instant was roughly the same throughout the trace and did not increase over time for all approaches. With a runtime of 73 ms per instant for the symbolic monitoring with assumptions and a window size of 100, the monitoring of the entire trace with 14 heartbeats takes about 200 seconds and would thus be too slow for online monitoring of an ECG signal at this event rate. In the case of the gap trace, the symbolic approach's runtime differed significantly depending on whether or not the current window contained uncertainty bursts requiring advanced symbolic reasoning. In this case the computation time per instant rose at maximum to 1.5 seconds for the symbolic approach with assumptions and a window size of 100 events, which is also too slow for online monitoring. In the specific situation even two uncertainty gaps have been within the considered window. Furthermore the runtime of the symbolic approaches shows non-linear growth in terms of the streams in the flatted specification. For the interval technique the growth is linear.

Case Study 2. Path planning: Combination of discrete and continuous monitoring

In the second (synthetic) case study from [HKLS24] we analyze the behavior of the (full) symbolic backend for a specification which equally contains discrete parts and parts which require continuous calculation. The setting is the following: A battery powered mobile robot system is driving around a building with a number of rooms, where some of them are connected to each other. In room 0 the base station of the robot is located, which is used for charging. As time proceeds the energy level of the robot is continuously going down.

For such a robot system, it is crucial to ensure that no matter what concrete task the robot is performing, it always has enough energy to return to the base station in room 0 for recharging. In this example, we want to create a safety layer around the robot's controller unit that prevents the robot from entering rooms where this property is violated, using the anticipatory symbolic monitoring approach.

We can formulate the final violation of the outlined condition in STL syntax (which can be translated to LOLA syntax, see section 3.4.2) as follows:

$$\varphi_{\text{err}} = \mathcal{F}(\mathit{room} \neq 0 \wedge (\mathit{energy} < 5))$$

I.e. we state that the energy level may not run below threshold 5 (percent) without being in room 0 somewhere in the trace. For representation of the current room we use a real-valued stream but specify via an assumption that it may only take a finite number of concrete values. We thus use it as discrete variable.

In the considered building, it is only possible to go from one room to a selection of other directly connected rooms within one instant. We can encode this in an

assumption, where we specify that if *room* has a particular value at the current instant the value at the subsequent instant may only be within a specific set (of connected rooms).

We specify a further condition that the energy level of the battery decreases by a fixed amount of 5 units per instant:

$$\varphi_{\text{egy}} = \mathcal{G}(\mathit{energy}^{+1} = (\mathit{energy} - 5))$$

Thereby energy^{+1} denotes the value of the energy input at the next instant. Note that actually this cannot be formulated in STL, as STL does not allow access of the input values at previous or subsequent instants. However, we can realize a property like this without problems in LOLA via an offset expression and thus translate this condition to LOLA.

To ensure that the robot does not enter a room from which it cannot get back to the base station without running out of energy, we want to monitor the condition

$$\varphi_{\text{enter}_i} = (\mathit{room}^{+1} = i) \wedge \varphi_{\text{egy}} \wedge \neg\varphi_{\text{err}}$$

for each room i, where room^{+1} denotes the value of the next event on stream *room*, i.e. the next room the robot enters.

What we aim at in this example is to anticipatorily monitor the φ_{enter_i} formulas and prevent the monitor from entering room i if φ_{enter_i} is false. This is because if a perfect recurrent monitor casts the output false for φ_{enter_i}, there is no extension of the current input trace (i.e., no path of the robot) where it can enter room i in the next step and is still able not to run into the error condition φ_{err} when the energy consumption follows φ_{egy}.

A benefit of synthesizing a monitor from this property, rather than programming one manually, is that we can easily add further conditions to our properties. In our example we don't only want to check that the robot can return to the base station but we also want to be sure that throughout the whole trace it remained for at least three steps in a row in a specific room k (e.g. to fulfill some task there). Therefore we can simply adjust the monitored condition to

$$\varphi_{\text{enter}_i} = (\mathit{room}^{+1} = i) \wedge \varphi_{\text{egy}} \wedge \neg\varphi_{\text{err}} \wedge \mathcal{F}(\mathcal{O}(\mathcal{G}_{[0,3]}(\mathit{room} = k)))$$

As pointed out in section 3.4.2, the STL conditions from above (including the access of stream values of the next and previous instant) can be translated to a LOLA specification and be monitored with the symbolic approach from this thesis.

In the following the synthesis time and monitoring time per instant (with and without the presence of uncertainty) of the described specification are analyzed for different numbers of rooms (6-20) in the building. For the measurement a test bench was created which simulates a random walk of the robot through the building, following the instructions of the monitor. This way, the monitor made sure that when the

robot started to run out of energy, it would choose an appropriate way back to room 0 (and with remaining in room k for three instants if still necessary) and stay there. The correctness of the walk was later double-checked to confirm the correctness of the monitor's answers. In the example the perfect rewriting strategy for linear arithmetic was used, which did yield a repeating constraint set in the initial symbolic transformer semantics for all utilized specifications. This is due to the nature of the problem. For every room a minimal energy level can be determined which is sufficient to return to the base room, depending on whether the task has already been fulfilled or not. This is what the repeating symbolic transformer in the initial semantics encodes.

Concerning the synthesis, the monitor creation time was measured for 6,8,12,16 and 20 rooms. The development of the synthesis time is visualized in figure 6.3.

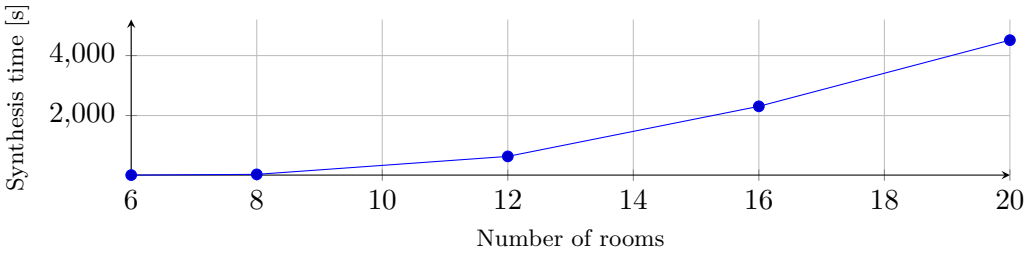


Figure 6.3.: Synthesis time of the symbolic monitor (i.e. computation time of initial semantics) in seconds for the path planning case study with a growing room number.

As one can see the synthesis time increases significantly with the complexity of the specification. While for 6 rooms the computation of the initial semantics is under 14 seconds, the time for 20 rooms is already 4,515 seconds, i.e. roughly $1\frac{1}{4}$ hours, which is pretty high for the complexity of the specification.

On the one hand this is because during the initial semantics computation from backward, the last determined symbolic transformer has to be compared to a growing number of other transformers until the repeating constraint set is found. In fact with a growing complexity of the specification more and more transformers have to be computed from backward until the repeating one is found, as the longest possible way in the building also grows with the specification complexity. Likewise the complexity of the individual symbolic transformers increases with the number of rooms, as information about every room have to be computed. The number of transitions between the rooms (which gives a rough measure of the specification complexity) and the number of computed transformers in the initial semantics (corresponding to the longest way back to room 0 plus three instants lasting in room k) in relation to the number of rooms is listed in the subsequent table. The fact that the synthesis time for 12-20 rooms only shows linear growth is probably due to the fact that the number of computed transformers for these room numbers is almost equal.

Room number in specification	6	8	12	16	20
Number of transitions between rooms	14	22	30	46	55
Number of computed symbolic transformers	9	9	12	13	13

On the other hand, while the SMT backend, Z3, is highly performant in deciding the satisfiability of logical formulas, it is much less sophisticated in other tasks, especially in the simplification of expressions. A first glance at the computed formulas immediately shows that the simplification keeps many superfluous sub-expressions, e.g. $\cdots \vee (x \leq 10) \vee (x \leq 15) \vee \ldots$, where the $(x \leq 10)$ part can obviously be removed from the disjunction. Throughout the computation of the initial semantics this leads to a strongly increasing blow-up in size of the constraint sets which makes all subsequent computations harder.

However note, that the computation of the initial semantics can be done in advance of the actual monitoring process, during monitor synthesis. The relevant measure for the online monitor execution is the computation time per received input instant, because this determines the event rate the monitor can handle. For the example above, the runtime per instant (on average) has been measured for a random walk of size 500, one time without uncertainty; One time with 30 % of the energy input values set to an interval of up to 15% below the actual value and the room input exchanged by a set of three possible rooms around (and including) the current one; And finally one time with 15 % of the input events set to full uncertainty.

The development of the runtimes, again for the different complex specifications with 6 to 20 rooms, is visualized in figure 6.4.

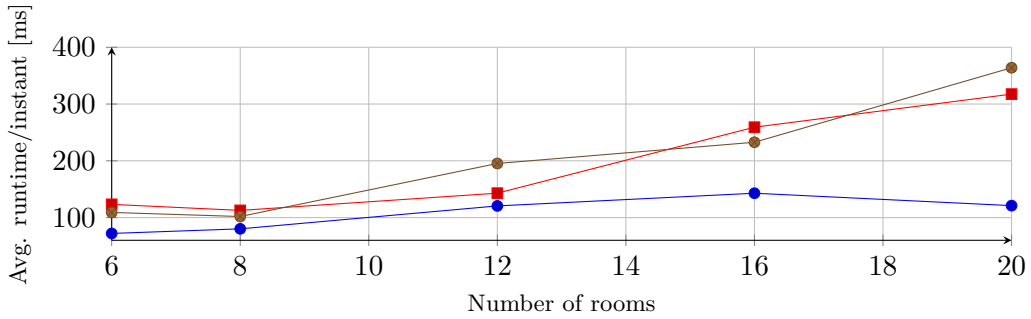


Figure 6.4.: Average runtime of the symbolic monitor per instant in ms for the path planning case study and a growing number of rooms without uncertainty in the input (●) vs. 30 % of all input values replaced by intervals/sets around the exact value (■) vs. 15 % of all input values completely unknown (●).

One can see that the presence of uncertainty is in fact increasing the runtime. Yet in the concrete scenario of a robot moving around rooms and requiring verdicts whether a specific room may still be entered, the runtime for 20 rooms even under

the presence of uncertainty (364 ms per instant) would be sufficient. The runtimes for 15 % full uncertainty and 30 % partial uncertainty are roughly comparable. For the certain case the runtime is not significantly increasing for room sizes above 12 and stabilizes at about 130 ms per instant. This is probably due to the fact that when the room is certainly known, the formula can be reduced directly to a relatively compact representation, no matter how many rooms were originally involved.

Case Study 3. Collision avoidance: Continuous monitoring

Finally the case study from [HKLS24] reflecting a specification with mainly numeric character is presented in this section. The idea is to use an anticipatory LOLA monitor to control a robot to drive around objects. In the particular setting a robot is following a user defined path in an area with obstacles. The LOLA monitor yields information about which steer angles the robot can take to avoid collisions with objects in the way. In this respect it is important that the robot cannot steer arbitrarily strong and thus several monitoring steps have to be anticipated to figure out if a collision is avoidable under a particular steer angle.

In detail the case study is modeled as follows. The specification defines four input streams of type $\mathbb{R}$. On the one hand the steer angle of the robot on stream *angle*. On the other hand streams *dist*, *left* and *right* which indicate the distance and the leftmost and rightmost point of the closest obstacle to the robot in driving direction on a horizon parallel to the robot (see figure 6.5).

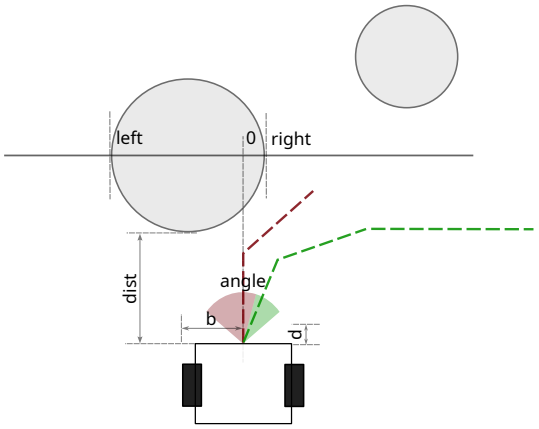


Figure 6.5.: Scheme of the third case study. The monitor figures out by anticipation which steer angles are possible (green) and which would lead to a collision with the closest object (red); (figure from [HKLS24]).

We assume that the obstacles are circular or we presume a circular bounding area around the obstacles. Via assumption we define the maximal steer angle, a maximal size of the objects and other basics, like the value of *left* is lower than the one of

right. As in the case study before the condition of the error scenario, φ_{err}, and also the other properties are given in STL syntax which is then translated to LOLA:

$$\varphi_{err} = \mathcal{F}((dist < d) \wedge (right \geq -b) \wedge (left \leq b))$$

Thereby $d, b \in \mathbb{R}$ are constants which define a bounding box around the robot (see figure 6.5). The formula states that an error is caused if anywhere in the input trace the nearest obstacle is closer than d to the robot and it is neither fully left (right edge left of $-b$) nor fully right (left edge right of b) of the robot. In this case the robot would collide with the object.

Additionally we describe the worst-case approaching behavior of the robot in φ_{app}:

$$\varphi_{app} = (dist - dist^{+1} = v) \wedge (left - left^{+1} = c \cdot angle) \wedge (right - right^{+1} = c \cdot angle)$$

for constants $v, c \in \mathbb{R}$. Thus, we specify that the obstacle gets closer to the robot with a speed of v units per instant and the left and right edges move according to the steer angle, related by a constant c. Note that this relation is of course not accurate. The way the obstacle gets closer in relation to a specific steer angle is a non-linear relation, however, the above equation is sufficient as long as it over-approximates the reality, i.e. overestimates the speed the obstacle gets closer and underestimates the speed it moves out of the robots collision range. Therefore c and the (possibly non-linear) translation between the value of stream *angle* and the actual steer angle must be chosen appropriately.

The given definitions can be used to check whether a collision of the robot is inevitable if a certain steer angle is chosen for the next instant. Therefore, we define φ_{steer,a_i} for a set of specific steer angles $a_i \in \mathbb{R}$ as

$$\varphi_{steer,a_i} = (angle^{+1} = a_i) \wedge \mathcal{G}(\varphi_{app}) \wedge \neg\varphi_{err}.$$

If a recurrent LOLA monitor casts `false` as output for this stream we can conclude that setting the steer angle to a_i in the next step will inevitably lead to a crash - provided the approaching behavior specified in φ_{app}. Note that an alternative to define the streams φ_{steer,a_i} would be to let the monitor cast the whole computed constraint set for each instant and then to determine externally which angles are possible for the next step. Yet for the case study this kind of querying a set of discrete angles seems to be more convenient.

Finally, for additional stability of the approach also the constraint $\mathcal{G}(angle^{+2} \leq a_{max} \wedge angle^{+2} \geq -a_{max})$ was added to the φ_{steer,a_i} properties, where $a_{max} \in \mathbb{R}$ is a constant below the maximal possible steer angle of the robot (which is set by assumption). This leads to a situation where the monitor has more options for the next step than considered in previous anticipations, resulting in more robust guidance of the robot as additional options arise in the current step.

For qualitative evaluation of the case study an integration of the robot operating system ROS[4] and the LOLA monitoring tool has been created and run on a turtle bot robot[5] inside the robot simulation environment gazebo[6] [KH04] (see figure 6.6)[7]. To evaluate the approach the robot was set into an area with several cyclic obstacles around. Furthermore the user could set a path the robot should follow. While driving, the robot's main program loop periodically determined the closest obstacle in the robot's driving direction (by a query to the gazebo environment) and passed it to the LOLA monitor, which determined possible steer angles. The robot was then programmed to chose the steer angle which was closest to the user defined path. This way the robot followed the path given by the user but drove around obstacles if the path led through them or was too close to them.

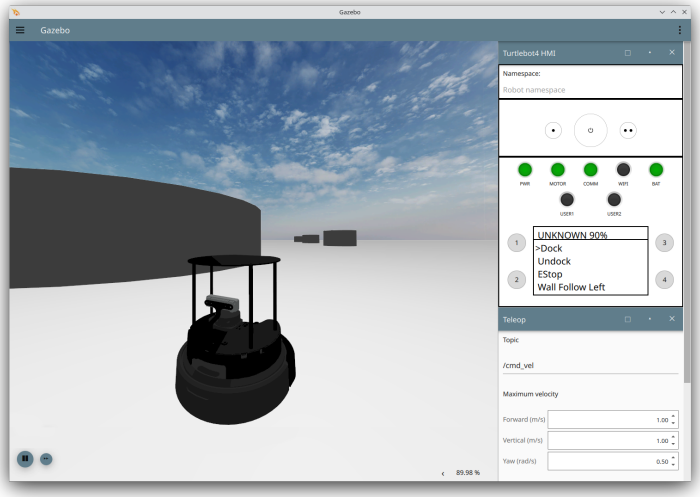


Figure 6.6.: Screenshot of the simulated robot in the gazebo environment, controlled by a LOLA monitor (figure from [HKLS24]).

The example was first run with fully certain inputs. Additionally experiments were carried out where the values of the left and right position of the obstacle were substituted by uncertainty intervals with a margin of $\pm x\%$ (for x from 5 to 40 in steps of 5) of the maximal values around the actual value. Of course, the monitor's tolerance for uncertainty is strongly influenced by how over-approximative φ_{app} is defined. In our case the monitor was still able to steer the robot correctly up to an uncertainty margin of 30% without defining φ_{app} so over-approximative that no movement was possible anymore.

[4] https://www.ros.org/
[5] https://www.turtlebot.com/
[6] https://gazebosim.org/
[7] The simulation in gazebo, which was developed as part of [HKLS24], was done by a co-author and is not the work of the author of this thesis.

Note at this point that it can be a problem that the monitor only observes the closest obstacle to the robot. This is because it could allow steer angles which avoid a collision to the closest object but not to the second closest one. There are two approaches to counter this problem. First, one could duplicate the input streams, φ_{err} and φ_{app} for the second closest obstacle and include these streams in the queries for the steer angles. This would of course increase the complexity of the specification and computation time. Second, one could run two monitors in parallel, one for the closest and one for the second closest obstacle and only allow angles which are allowed by both monitors. However in some situations this second option would allow some angles which ultimately lead to collisions. This is because if the monitors are independent of each other, they can assume different continuations for a given angle that avoid a crash, but are not compatible with each other.

For the quantitative analysis of the case study the traces from the gazebo simulation have been extracted and handed to the monitor in an offline setting (i.e. without the actual simulation running in parallel) to measure the pure runtime of the monitor per instant. As one would expect the runtime increases if the input values contain uncertainty. In this case the constraint set cannot be simplified as much as under presence of certain values. Larger uncertain intervals also slightly increase the runtime, since they require more options for future continuations to be considered. The runtimes for the different uncertainty margins are visualized in figure 6.7.

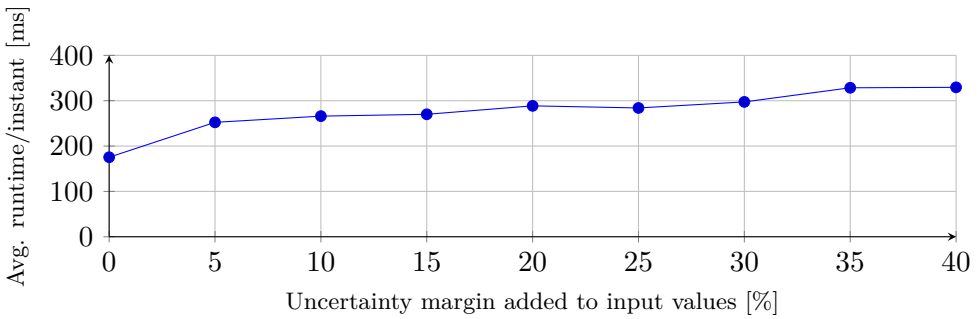


Figure 6.7.: Average runtime per instant in ms of the LOLA monitor with uncertainty of 0 to 40 % around the exact input values.

The runtime is sufficient to execute roughly 3-6 angle queries per second, depending on the chosen uncertainty level. Compared to the second case study, the synthesis time of the monitor was significantly faster as the computed constraint sets were much smaller: It took 8-9 seconds and the initial semantics could be determined after 14 computed states.

6.3. Discussion

Overall, the case studies presented in this chapter have shown that perfect recurrent monitoring of the linear arithmetic LOLA fragment under uncertainty and assumptions is a powerful technique and can be used in several scenarios of practical relevance. Also the perfect rewriting strategy for linear arithmetic based on quantifier elimination has shown to be suitable for perfect and trace-length-independent recurrent online monitoring in a selection of non-trivial use cases, which relied on reasoning about the future.

However the relatively high evaluation and synthesis time of the resulting symbolic monitors compared to other runtime verification tools can be considered a problem of the approach. Already the evaluation time in the first past-only case study has shown to be roughly 10 to 220 times slower than a comparable interval-based solution (yet also significantly more precise). This is despite the fact that an optimized version of the symbolic monitoring approach is used. In the first case study the monitor is even too slow for online monitoring a human heartbeat at the given event rate. In the latter two case studies, the runtime is acceptable for the specific scenarios.

Especially the second case study shows the limits of the full symbolic monitoring approach. The extended graph-search task with 20 nodes and 55 transitions takes already over one hour of synthesis time. This makes it likely that for larger graphs or more complex tasks the symbolic approach is not applicable in practice anymore.

The case study is exemplary for a situation where symbolic reasoning significantly slows down the computation of relatively simple problems. First and foremost, this is because during symbolic reasoning, all contextual information about the problem – that would be the key to an efficient solution – is lost, which increases the computation effort. On the other hand exactly this makes symbolic monitoring very general. Another negative influence on the runtime is – as already mentioned in the second case study – that the computed symbolic expressions are much more complex than they need to be due to the insufficient capabilities of formula simplification in the used SMT backend Z3.

An interesting direction for future work to address this problem would be to investigate symbolic monitoring in more application-specific LOLA fragments. E.g. to reason about spacial-temporal properties it would be possible to enrich the boolean LOLA fragment with simple RCC8 spatial operators [RCC92] which would probably allow for more efficient symbolic computations and better simplifications.

A further not yet discussed drawback of the symbolic LOLA monitoring approach, which became particularly apparent during the evaluation of the second and third case study, is that sometimes small adjustments in the specification can make the difference between being able to perfectly determine the initial symbolic transformer semantics or having to use over-approximations (widening). Basically every statement in the specification that effectively leads to “counting” the number of positions

from the trace end to the beginning leads to a non-stabilizing initial semantics. Counting from the beginning of the trace however is not an issue as the computed symbolic transformers are parametric in the predecessor stream events. However, this "counting" from the end often happens unintentionally. Consider e.g. the two assumptions $s[now] \leq s[-1|v_{max}+1]-1$ and $s[+1|v_{min}] \leq s[now]-1$ where v_{max}, v_{min} denote the possible maximal and minimal value allowed for stream s of type $\mathbb{R}$. Both assumptions look relatively similar. They both state that a value on stream s has to be at least one less than the value before. Yet with the second assumption the computed constraint sets of the initial semantics also contain that in the last step s is at least $v_{min}+1$, in the second last step at least $v_{min}+2$ and so on. This however leads to non-stabilizing constraint sets in the computation of the initial semantics and widening has to be applied. For the first assumption this is not the case, as it contains no reference to v_{min}. It is important to identify such parts of the specification and to communicate to the user that they are the actual cause that no perfect monitoring is possible. Further this example shows the need of precise widening strategies which can identify exactly those constraints and widen or remove them, but not other constraints which are actually stable.

Altogether we can conclude that the symbolic recurrent LOLA monitoring approach is indeed a viable and powerful solution for perfect and sound monitoring of linear arithmetic properties under presence of uncertainty and assumptions. Especially for shielding and guiding the monitored system, the use of perfect, anticipatory monitoring seems promising. For properties of higher complexity however the approach may become intractable, which could probably be solved by usage of more application-specific logics and specialized solvers. In addition, the prototypical implementation presented in this chapter still requires further usability improvements to increase its suitability for practical use.

7

Conclusion and future work

In this final chapter, we want to recapitulate the contents presented in this thesis and discuss directions for future research.

7.1. Summary

The main question that has been addressed in this work has been *whether and how a perfect and efficient monitoring procedure for LOLA specifications, including those with future references, can be constructed in the presence of uncertainties and assumptions.*

Therefore, in the preliminaries chapter, the notion of perfect LTL monitoring, as suggested by the LTL_3 construction, was reviewed, which requires the properties of impartiality (i.e. revealing all possible output valuations w.r.t. the received input prefix) and anticipation (i.e. considering all future continuations of the input prefix for the current output). Also, the standard past LTL monitoring algorithm of Havelund and Roşu has been revisited there, as it serves as a role model for monitoring a correctness property repeatedly with respect to the current trace position instead of the initial one; a concept which was later called recurrent monitoring.

Based on these concepts we have defined a general theory about different kinds of synchronous properties and their (perfect) monitoring in chapter 3. We were able to show that the stream runtime verification language LOLA describes such synchronous properties, particularly pointwise ones. In this thesis, it was further proved that any such property – and thus logic over discretely timed traces or streams – can indeed be expressed as a LOLA specification. Yet, the traditional LOLA monitoring algorithm from the original paper [DSS+05] does not provide a perfect monitor in terms of the previous definitions, as it is not able to consider potential continuations of the currently received input trace. Furthermore, this algorithm is

not able to handle uncertainties (i.e. unknown or noisy input values) nor assumptions (i.e. additional knowledge about the monitored system) which restrict the set of actual possible input continuations.

To remedy this, a novel generic monitoring framework for LOLA specifications based on abstract interpretation was introduced in chapter 4. In addition to the general construction of a LOLA monitor itself, also conditions for its perfectness have been discussed. Compared to other approaches for perfect (i.e. impartial and anticipatory) monitoring of complex properties [CTT19, FMPW23], the novelty of the presented framework on the one hand is, that it is not restricted to specific theories nor to correctness properties, which can only be satisfied or not, but can be used for all kinds of synchronous properties. On the other hand, the approach of efficiently computing the initial specification semantics from back to front, parametric in the predecessor configuration of each instant, and then using this structure for successive monitoring has not been proposed so far.

The introduced theory of LOLA monitoring is then utilized in chapter 5 to build a canonical symbolic LOLA monitor. The monitoring technique from this chapter basically consists of symbolic unrolling of the specification and so-called expression rewriting to keep the resulting symbolic expressions constant in size. This enables efficient trace-length-independent monitoring. In some cases (e.g. boolean LOLA) perfect versions of this expression rewriting can be obtained by applying quantifier elimination. For other fragments (e.g. the linear arithmetic LOLA fragment) this is not possible and over-approximating techniques together with widening have to be chosen, resulting in an imperfect but sound, trace-length-independent monitor.

Finally, in the previous chapter, a prototypical implementation of the symbolic monitoring approach for the linear arithmetic LOLA fragment has been evaluated by means of three practical case studies. In the first, past-only specification, detection of heartbeats in an ECG signal was performed under presence of uncertainties in the input trace and assumptions. The second one was about anticipatory path planning in a room map in presence of an energy consumption constraint. Finally, the third case study dealt with controlling a robotic system to steer around obstacles along a given path. The case studies demonstrated application scenarios where the perfect LOLA monitoring was advantageous to use. Yet they also demonstrated its pitfall: The runtime – of the synthesis as well as of the evaluation – was significantly higher than a similar non-anticipatory and imperfect monitoring approach. Especially for the second case study the synthesis time increased strongly with the specification's complexity and thus made room for optimizations visible.

7.2. Future work

Potential future research can be divided into two groups: theoretical and practical. From a theoretical point of view, the following research questions are of interest.

1. In chapter 3 we have restricted to so-called synchronous properties, i.e. those that assign a property value to every position of an input trace or timed stream over a discrete time domain. Several formalisms however, e.g. M(I)TL, STL or TeSSLa, can also be used over non-discrete and dense time domains. While in this case the monitor inputs are usually still timed traces which e.g. encode a piece-wise constant signal, the corresponding monitor outputs are not necessarily synchronous to the timestamps for which the monitor receives inputs anymore. An extension of properties to this asynchronous setting and analogous monitoring strategies are left open in chapter 3, but are relevant when it comes to the question how the corresponding specifications can perfectly be monitored.

2. Likewise, several formalisms as LTL, TeSSLa or M(I)TL are defined not (only) for finite but also for infinite traces or streams. As the algorithm in this thesis is based on the language LOLA, it is restricted to finite traces, specifically those of a fixed, constant length. To be able to monitor formalisms over infinite input sequences, one would have to adapt the approach to this end. A possible way to go would be to extend the LOLA semantics to infinite streams and investigate whether it is still a canonical formalism for infinite pointwise properties. Further it remains open if the suggested algorithm can – with some modifications – also be used in an infinite setting. Some preliminary considerations about the extension to infinite traces can be found at the end of chapter 5.

3. Further, in chapter 3 we have introduced sound and perfect monitoring strategies for pointwise properties. Yet, for sound but imperfect monitoring, it is usually desirable to have a measure of how coarse the resulting over-approximation is in the worst case. The introduction and elaboration of such a measure and the valuation of the presented sound monitoring approaches was left as future work in this thesis.

4. Finally, regarding uncertainty, the introduced monitoring algorithm was restricted to the case of timestamp-immanent uncertainty, without specific probabilities assigned to each possible value. It is left open whether and how the presented algorithm can be adapted to handle these kinds of uncertainties.

From a practical perspective several directions may be followed to optimize the performance of the basic symbolic monitoring implementation from chapter 5 and to make it applicable to large and complex specifications.

1. While in chapter 6 the expressive linear arithmetic LOLA fragment was used to encode the specific examples, it is unclear whether the use of more application-tailored algebras could lead to better runtimes in synthesis and execution of the monitor. Therefore, it would be necessary to introduce and study algebras with operators for the specific application domains, together with appropriate rewriting and widening strategies for them (see also end of chapter 6).

2. Additionally it should be analyzed to which degree a more sophisticated expression simplification can speed up the symbolic monitoring for linear arithmetic. An extension of the utilized Z3 backend in this direction could also be advantageous for several other use cases. Related to this, also further optimizations of the basic algorithm from chapter 5 would be conceivable, e.g. splitting the constraint set into several independent ones to speed up the symbolic computation and the like.
3. Finally, also advanced widening and sound rewriting strategies for the utilized algebras should be investigated. This would allow for a sound monitoring with little error in case the original specification is not in a shape s.t. it can be monitored in a perfect manner (see also remarks at end of chapter 6). Therefore it might be beneficial to revisit approaches from the field of abstract interpretation where similar issues have to be handled.

Together with the mentioned extensions and optimizations, the presented framework could indeed have potential as a generic monitoring solution in the area of runtime verification, as a general framework to reason about monitorability and constructions, and in practical use.

A

Basic notations

This appendix provides basic notations and introduces fundamental mathematical definitions which are used throughout the whole thesis. Several of them are recapitulated or again defined in the main part of the work.

A.1. Notations

We use the common logical symbols $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$ as operators in symbolic formulas. For reasoning on a meta level the symbols $\Leftrightarrow$ and $\Rightarrow$ and English language are used.

If we define a newly introduced symbol s by some expression φ, we write $s := \varphi$. To define syntactical structures we use EBNF-like syntax: $\varphi ::= \ldots$ where the right hand side may contain several definition alternatives, separated by $|$, and φ for recursive definitions.

A.2. (Ordered) sets, families, tuples, sequences, relations, functions

Note: In this section $\varphi(x)$ represents an expression over x.

We denote **sets** in the common enumerating $(\{s_0, s_1, \ldots, s_n\})$ or descriptive notation $(\{x \mid \varphi(x)\})$, where $\varphi(x)$ can be a comma-separated conjunction of propositions and all free variables in φ apart from x are existentially quantified. For union, intersection, difference, subset relation and containment relation of sets the usual operators $\cup, \cap, \setminus, \subseteq, \in$ are used. To quantify universally or existentially over a set S we use the notations $\forall s \in S.\, \varphi(s)$ and $\exists s \in S.\, \varphi(s)$ respectively. After $s \in S$ we may also note down further conditions on s, separated by comma. We use the notations $\nexists$ and $\not\forall$ to denote that the corresponding quantification does not hold. With $|S|$

we denote the number of elements in S and have $|S| = \infty$ if it is an infinite set. We use 2^S for the power set of S and $\emptyset$ for the empty set. The set $\mathbb{B} := \{\texttt{true}, \texttt{false}\}$ is the boolean domain.

Families over an index set I are indexed collections with elements s_i for every $i \in I$. We denote such a family with $(s_i)_{i \in I}$. We use the notation $s \in (s_i)_{i \in I}$ if there is an $i \in I$ s.t. $s = s_i$. For a family $(s_i)_{i \in I}$ we also use notations like $\bigcup_{i \in I} s_i$ to fold an operation over all elements of a sequence.

With S^* we refer to the set of all finite **sequences** of elements from set S. We use S^ω for infinite sequences, $S^\infty = S^* \cup S^\omega$. With $|s| \in \mathbb{N}$ we refer to the length of a sequence $s \in S^*$ and have $|s| = \infty$ for $s \in S^\omega$. S^n denotes the set of all sequences of size n and $S^{\leq n}$ all sequences of size up to n. As notation of a concrete sequence $\langle s_0, s_1, \dots \rangle$ or alternatively $s_0, s_1, \dots$ and $s_0 s_1 \dots$ is used. With $S(k)$ we denote the k^{th} element in S, starting with 0. The empty sequence is represented by ϵ. We use $S^+ := S^* \backslash \{\epsilon\}$ for all sequences of positive length. The concatenation of a finite with a finite or infinite sequence is denoted by $\langle s_0, s_1, \dots, s_n \rangle \circ \langle s_{n+1}, s_{n+2}, \dots \rangle := \langle s_1, s_2, \dots, s_n, s_{n+1}, s_{n+2}, \dots \rangle$ or simply by $\langle s_0, s_1, \dots, s_n \rangle \langle s_{n+1}, s_{n+2}, \dots \rangle$. Sequences are families of elements over index set $\{0, 1 \dots, n\}$ or $\mathbb{N}$ respectively, thus we also use the notation $(s_i)_{i \in I}$ for sequences.

The set of all **tuples** of sets $S_1, \dots, S_n$ is denoted with $S_1 \times \dots \times S_n$, elements of it are written as $(s_1, \dots, s_n)$. We use $\pi_i((s_1, \dots, s_n)) := s_i$ as selector of the i^{th} element in a tuple. We consider sequences as special case of tuples. For concatenation of two tuples we also use the operator $\cdot \circ \cdot$.

For a binary **relation** R between elements in sets A and B we use, as common, the infix notation $a \; R \; b$ if and only if $(a, b) \in R$ for some $a \in A$, $b \in B$. With $a \not R \; b$ we indicate that relation R does not hold for a, b.

A tuple of a set and a belonging binary relation $(S, \leq)$ is called **(partially) ordered set** (c.f. [DP90]), if and only if for all $a, b, c \in S$

- $a \leq a$ and
- $a \leq b$ and $b \leq c$ imply $a \leq c$ and
- $a \leq b$ and $b \leq a$ implies $a = b$.

If in this thesis we use a partially ordered set $(S, \leq)$ (e.g. for symbolic constraint sets in chapter 5) we implicitly identify all non identical elements $s, s' \in S$ for which we define $s \leq s'$ and $s' \leq s$, i.e. we consider s and s' as different representations of the same element (w.r.t. the order) and write $s = s'$. A tuple of a set and a belonging binary relation $(S, <)$ is called strictly partially ordered set, if and only if $<$ results from the relation of a partial order for S by removing the relation $a < a$ for all a in S. For a partially ordered set $(S, \leq)$ we use the notation $a < b$ if $a \leq b$ and $a \neq b$ and for a strictly partially ordered set $(S, <)$ we use $a \leq b$ if $a < b$ or $a = b$. In this thesis we use the notation $a \leq b \leq c$ to denote that $a \leq b$ and $b \leq c$ (and analogously for $<$). Ordered and strictly ordered sets are called totally ordered

if for every pair $a, b \in S$ with $a \neq b$ the order relations either relate a with b or b with a.

For the set of (total) **functions** from set A to B we use the notation $A \to B$, for partial ones $A \rightharpoonup B$. To define a function f from A to B (written $f : A \to B$) we use the notations $f(a) = \varphi(a)$ or $a \mapsto \varphi(a)$. For $f : B \to C$ and $g : A \to B$ we use the concatenation operator $f \circ g : A \to C$ for the function $x \mapsto f(g(x))$. With f^n for $f : A \to A$, $n \in \mathbb{N}$ we denote the n-fold concatenation of f. For $f : A \to B$, $\mathbf{pic}(f) := \{b \mid b = f(a), a \in A\}$. The above notations are also applicable to partial functions. With $\mathbf{id} : A \to A$ we denote the function $x \mapsto x$ for any set A. For $f : \mathbb{N} \to \mathbb{N}$ we use the Landau notation $\mathcal{O}(f) = \{g \mid g \in \mathbb{N} \to \mathbb{N}, \exists c \in \mathbb{N}.\, \exists n_0 \in \mathbb{N}.\, \forall n \in \mathbb{N}.\, n \geq n_0 \to c \cdot f(n) \geq g(n)\}$.

A.3. Numbers

The following symbols are used to denote specific sets of **numbers** in the thesis:

- $\mathbb{N}$ for the set of natural numbers, including 0.
- $\mathbb{N}^+$ for the set of positive natural numbers.
- $\mathbb{Z}$ for the set of integers.
- $\mathbb{Q}$ for the set of rational numbers.
- $\mathbb{Q}^{\geq 0}$ for the set of non-negative rational numbers.
- $\mathbb{Q}^+$ for the set of positive rational numbers.
- $\mathbb{R}$ for the set of real numbers.
- $\mathbb{R}^{\geq 0}$ for the set of non-negative real numbers.
- $\mathbb{R}^+$ for the set of positive real numbers.

With $\mathbb{I}_{\mathbb{S}} \subseteq 2^{\mathbb{S}}$, where $\mathbb{S}$ is any of the above number sets, we denote the set of all open, half-open and closed bounded and unbounded **intervals** of values for $\mathbb{S}$. We write $[a, b]$ for the closed interval between $a \in \mathbb{S}$ and $b \in \mathbb{S}$, where a and b are included. For half-open and open intervals we use "]" as left delimiter instead if a is excluded and if b is excluded "[" as right delimiter. For unbounded intervals we use the symbols ∞ and $-\infty$ as exclusive upper or lower bound.

For numbers from the above sets we use the common operators $+, -, *, \div, \cdot, \log, <, \leq, >, \geq$. For $a \in \mathbb{R}$ we extend $+$,$-$ for addition and subtraction with infinity: $a + \infty = \infty$ and $a - \infty = -\infty$. Likewise we write $\log(\infty) = \infty$. We use $\lfloor a \rfloor$ and $\lceil a \rceil$ for $a \in \mathbb{R}$ to denote the greatest and smallest integer which is smaller or greater than a, respectively.

A.4. Graphs, trees

A **graph** is a tuple (V, E) of a set of vertices (also called nodes) V and a set of edges E. In case of an unweighted graph we have a relation $E \subseteq V \times V$, indicating a connection between two vertices and in case of a weighted graph $E \subseteq V \times V \times \mathbb{R}$ where the last component denotes the numerical weight of the edge. We call a graph undirected, if for all $(u, v) \in E$ (or $(u, v, w) \in E$) we have that also $(v, u) \in E$ (or $(v, u, w) \in E$ resp.), otherwise directed. For a labeled graph we further add a function $l : V \to L$ to the tuple which assigns a label from a set L to every vertex. A path is a finite or infinite sequence of vertices s.t. all subsequent vertices are connected by edges. A finite path where first and last vertex are equal is called cycle. The distance between two vertices u and v in a graph is the number of elements of the shortest path, s.t. the first element is u and the last one is v. In a weighted graph, the sum of the edge weights of all edges in a path is called weight of the path and is ∞ if this sum does not exist in case of infinite paths.

A **tree** is a directed graph where no cycle exists and with a special vertex called root, from which a unique path to every other vertex in the tree exists. Analogous to graphs we also consider labeled trees in this thesis.

A.5. Finite automata

A **nondeterministic finite automaton (NFA)** (c.f. [HU79]) $A = (Q, \Sigma, \Delta, q_I, F)$ is a 5-tuple of a finite set of states Q, an input alphabet Σ which is finite, a transition relation $\Delta \subseteq Q \times \Sigma \times Q$, an initial state $q_I \in Q$ and a set of finite states $F \subseteq Q$. A sequence of states $q_0, q_1, \ldots, q_{n+1} \in Q^*$ corresponds to a finite sequence (aka word) $w = w_0 w_1 \ldots w_n \in \Sigma^*$ if $q_0 = q_I$ and $(q_i, w_i, q_{i+1}) \in \Delta$ for all $i \in \{0, \ldots, n\}$. The state sequence is called accepting if and only if $q_{n+1} \in F$. The language of A is the set of all words to which an accepting state sequence corresponds.

A **deterministic finite automaton (DFA)** (c.f. [HU79]) is a nondeterministic finite automaton where Δ contains exactly one element (q, s, q') for all pairs $(q, s) \in Q \times \Sigma$. For every NFA there is a DFA with the same language [HU79].

B

Measurement tables

This appendix contains the measured values for all case studies from chapter 6. All experiments were run on a Linux machine (kernel version 6.5.3) with an Intel i7-8550U CPU running at 1.80 GHz base frequency and with 8 GB of RAM.

B.1. Case Study 1. ECG signal analysis: Past-only monitoring

For a detailed description see section 6.2.1, case study 1.

Spec. size [streams]	**Interval [ms]**	**Symbolic [ms]**	**Symbolic w/ ass. [ms]**
34	0.134	6.29	17.208
44	0.138	6.507	19.537
64	0.151	7.364	23.963
104	0.223	9.894	35.186
144	0.298	14.626	48.039
184	0.278	19.523	62.119
224	0.381	32.914	73.635

Table B.1.: Avg. runtimes per instant in ms for case study 1 with differently complex specifications, where 20% of the values are exchanged by ranges of ±20% around the actual values. Measurements for the interval backend, the past-only symbolic backend and the past-only symbolic backend with an additional assumption in the specification.

Spec. size [streams]	Interval [ms]	Symbolic [ms]	Symbolic w/ ass. [ms]
34	0.102	1.518	15.909
44	0.139	1.601	17.228
64	0.182	2.337	19.536
104	0.24	3.852	30.106
144	0.344	5.756	33.442
184	0.319	8.048	53.702
224	0.414	16.743	54.475

Table B.2.: Avg. runtimes per instant in ms for case study 1 with differently complex specifications, where 5 full uncertainty gaps of 6-19 events are inserted in the trace. Measurements for the interval backend, the past-only symbolic backend and the past-only symbolic backend with an additional assumption in the specification.

B.2. Case Study 2. Path planning: Combination of discrete and continuous monitoring

For a detailed description see section 6.2.1, case study 2.

Rooms in spec.	Synthesis time [s]
6	13.635
8	37.413
12	636.885
16	2306.874
20	4514.926

Table B.3.: Synthesis time in seconds for case study 2 with different room numbers, full symbolic backend.

Rooms in spec.	No uncertainty [ms]	30% noisy [ms]	15% unknown [ms]
6	72.073	123.322	109.063
8	80.169	112.658	101.891
12	120.56	142.983	195.306
16	142.922	259.192	232.69
20	121.064	317.497	363.948

Table B.4.: Avg. runtimes per instant in ms for case study 2 with different room numbers, full symbolic backend. Measurements for full certainty; 30% of the energy input values exchanged by ranges of up to 15% below the real value and 15% of room inputs set to a set of three possible rooms; and 15% of the input values fully unknown.

B.3. Case Study 3. Collision avoidance: Continuous monitoring

For a detailed description see section 6.2.1, case study 3.

Injected uncertainty [%]	**Runtime per instant [ms]**
0	175.338
5	252.287
10	265.971
15	270.028
20	288.653
25	284.018
30	297.351
35	328.504
40	329.476

Table B.5.: Runtime per instant in ms for case study 3, full symbolic backend. Measurements for different amounts of uncertainty. Input values exchanged by intervals of $\pm x\%$ around actual values.

Bibliography

[ADFdB13] Azzeddine Amiar, Mickaël Delahaye, Yliès Falcone, and Lydie du Bousquet. Compressing microcontroller execution traces to assist system analysis. In Gunar Schirner, Marcelo Götz, Achim Rettberg, Mauro Cesar Zanella, and Franz J. Rammig, editors, *Embedded Systems: Design, Analysis and Verification - 4th IFIP TC 10 International Embedded Systems Symposium, IESS 2013, Paderborn, Germany, June 17-19, 2013. Proceedings*, volume 403 of *IFIP Advances in Information and Communication Technology*, pages 139–150. Springer, 2013.

[AFH91] Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. The benefits of relaxing punctuality. In Luigi Logrippo, editor, *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 19-21, 1991*, pages 139–152. ACM, 1991.

[AH91] Rajeev Alur and Thomas A. Henzinger. Logics and models of real time: A survey. In J. W. de Bakker, Cornelis Huizing, Willem P. de Roever, and Grzegorz Rozenberg, editors, *Real-Time: Theory in Practice, REX Workshop, Mook, The Netherlands, June 3-7, 1991, Proceedings*, volume 600 of *Lecture Notes in Computer Science*, pages 74–106. Springer, 1991.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[BFFR18] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to runtime verification. In Ezio Bartocci and Yliès Falcone, editors, *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 1–33. Springer, 2018.

[BG92] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[BHKZ11] David A. Basin, Matús Harvan, Felix Klaedtke, and Eugen Zalinescu. MONPOLY: monitoring usage-control policies. In Sarfraz Khurshid and Koushik Sen, editors, *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, volume 7186 of *Lecture Notes in Computer Science*, pages 360–364. Springer, 2011.

[BKR10] Udi Boker, Orna Kupferman, and Adin Rosenberg. Alternation removal in büchi automata. In Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis, editors, *Automata, Languages and Programming, 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part II*, volume 6199 of *Lecture Notes in Computer Science*, pages 76–87. Springer, 2010.

[BKV13] Andreas Bauer, Jan-Christoph Küster, and Gil Vegliach. From propositional to first-order monitoring. In Axel Legay and Saddek Bensalem, editors, *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*, volume 8174 of *Lecture Notes in Computer Science*, pages 59–75. Springer, 2013.

[BLS06a] Andreas Bauer, Martin Leucker, and Christian Schallhart. Monitoring of real-time properties. In S. Arun-Kumar and Naveen Garg, editors, *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science, 26th International Conference, Kolkata, India, December 13-15, 2006, Proceedings*, volume 4337 of *Lecture Notes in Computer Science*, pages 260–272. Springer, 2006.

[BLS06b] Andreas Bauer, Martin Leucker, and Jonathan Streit. SALT - structured assertion language for temporal logic. In *ICFEM*, volume 4260 of *Lecture Notes in Computer Science*, pages 757–775. Springer, 2006.

[BLS10] Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing LTL semantics for runtime verification. *J. Log. Comput.*, 20(3):651–674, 2010.

[BNK16] Andrew Butterfield, Gerard Ekembe Ngondi, and Anne Kerr. *A Dictionary of Computer Science*. Oxford University Press, 2016.

[BRH10] Howard Barringer, David E. Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: from eagle to ruler. *J. Log. Comput.*, 20(3):675–706, 2010.

[BS14] Laura Bozzelli and César Sánchez. Foundations of boolean stream runtime verification. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, volume 8734 of *Lecture Notes in Computer Science*, pages 64–79. Springer, 2014.

[Büc90] J. Richard Büchi. *On a Decision Method in Restricted Second Order Arithmetic*, pages 425–435. Springer New York, New York, NY, 1990.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.

[CGK+18] Edmund M. Clarke, Orna Grumberg, Daniel Kroening, Doron A. Peled, and Helmut Veith. *Model checking, 2nd Edition*. MIT Press, 2018.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski, editors, *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 84–96. ACM Press, 1978.

[CHL+18] Lukas Convent, Sebastian Hungerecker, Martin Leucker, Torben Scheffel, Malte Schmitz, and Daniel Thoma. Tessla: Temporal stream-based specification language. In Tiago Massoni and Mohammad Reza Mousavi, editors, *Formal Methods: Foundations and Applications - 21st Brazilian Symposium, SBMF 2018, Salvador, Brazil, November 26-30, 2018, Proceedings*, volume 11254 of *Lecture Notes in Computer Science*, pages 144–162. Springer, 2018.

[CHS+18] Lukas Convent, Sebastian Hungerecker, Torben Scheffel, Malte Schmitz, Daniel Thoma, and Alexander Weiss. Hardware-based runtime verification with embedded tracing units and stream processing. In Christian Colombo and Martin Leucker, editors, *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*, volume 11237 of *Lecture Notes in Computer Science*, pages 43–63. Springer, 2018.

[CJ12] Bob F Caviness and Jeremy R Johnson. *Quantifier elimination and cylindrical algebraic decomposition*. Springer Science & Business Media, 2012.

[Cou21] Patrick Cousot. *Principles of abstract interpretation*. The MIT Press, 2021.

[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, Munich, Germany, January 21-23, 1987*, pages 178–188. ACM Press, 1987.

[CTT19] Alessandro Cimatti, Chun Tian, and Stefano Tonetta. Assumption-based runtime verification with partial observability and resets. In Bernd Finkbeiner and Leonardo Mariani, editors, *Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal, October 8-11, 2019, Proceedings*, volume 11757 of *Lecture Notes in Computer Science*, pages 165–184. Springer, 2019.

[CTT21] Alessandro Cimatti, Chun Tian, and Stefano Tonetta. Assumption-based runtime verification of infinite-state systems. In Lu Feng and Dana Fisman, editors, *Runtime Verification - 21st International Conference, RV 2021, Virtual Event, October 11-14, 2021, Proceedings*, volume 12974 of *Lecture Notes in Computer Science*, pages 207–227. Springer, 2021.

[CVWY90] Costas Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory efficient algorithms for the verification of temporal properties. In Edmund M. Clarke and Robert P. Kurshan, editors, *Computer Aided Verification, 2nd International Workshop, CAV '90, New Brunswick, NJ, USA, June 18-21, 1990, Proceedings*, volume 531 of *Lecture Notes in Computer Science*, pages 233–242. Springer, 1990.

[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 Simone André da Costa Cavalheiro and José Luiz Fiadeiro, editors, *Formal Methods: Foundations and Applications - 20th Brazilian Symposium, SBMF 2017, Recife, Brazil, November 29 - December 1, 2017, Proceedings*, volume 10623 of *Lecture Notes in Computer Science*, pages 179–196. Springer, 2017.

[Dij72] Edsger W. Dijkstra. *Chapter I: Notes on Structured Programming*, page 1–82. Academic Press Ltd., GBR, 1972.

[DLS08] Wei Dong, Martin Leucker, and Christian Schallhart. Impartial anticipation in runtime-verification. In Sung Deok Cha, Jin-Young Choi, Moonzoo Kim, Insup Lee, and Mahesh Viswanathan, editors, *Automated Technology for Verification and Analysis, 6th International Symposium, ATVA 2008, Seoul, Korea, October 20-23, 2008. Proceedings*,

volume 5311 of *Lecture Notes in Computer Science*, pages 386–396. Springer, 2008.

[DLT14] Normann Decker, Martin Leucker, and Daniel Thoma. Monitoring modulo theories. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 341–356. Springer, 2014.

[dMB08] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[DP90] Brian A Davey and Hilary A Priestley. *Introduction to lattices and order*. Cambridge university press, 1990.

[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 *12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23-25 June 2005, Burlington, Vermont, USA*, pages 166–174. IEEE Computer Society, 2005.

[EH97] Conal Elliott and Paul Hudak. Functional reactive animation. In *ICFP*, pages 263–273. ACM, 1997.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of algebraic specification 1: Equations and initial semantics*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1985.

[FFS+19] Peter Faymonville, Bernd Finkbeiner, Malte Schledjewski, Maximilian Schwenger, Marvin Stenger, Leander Tentrup, and Hazem Torfah. Streamlab: Stream-based monitoring of cyber-physical systems. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 421–431. Springer, 2019.

[FFST16] Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah. A stream-based specification language for network monitoring. In Yliès Falcone and César Sánchez, editors, *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30,*

2016, Proceedings, volume 10012 of *Lecture Notes in Computer Science*, pages 152–168. Springer, 2016.

[FFST17] Peter Faymonville, Bernd Finkbeiner, Maximilian Schwenger, and Hazem Torfah. Real-time stream-based monitoring. *CoRR*, abs/1711.03829, 2017.

[FKRT21] Yliès Falcone, Srdan Krstic, Giles Reger, and Dmitriy Traytel. A taxonomy for classifying runtime verification tools. *Int. J. Softw. Tools Technol. Transf.*, 23(2):255–284, 2021.

[FMPW23] Paolo Felli, Marco Montali, Fabio Patrizi, and Sarah Winkler. Monitoring arithmetic temporal properties on finite traces. In *AAAI*, pages 6346–6354. AAAI Press, 2023.

[GDS20] Felipe Gorostiaga, Luis Miguel Danielsson, and César Sánchez. Unifying the time-event spectrum for stream runtime verification. In Jyotirmoy Deshmukh and Dejan Nickovic, editors, *Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6-9, 2020, Proceedings*, volume 12399 of *Lecture Notes in Computer Science*, pages 462–481. Springer, 2020.

[GH05] Allen Goldberg and Klaus Havelund. Automated runtime verification with eagle. In Ulrich Ultes-Nitsche, Juan Carlos Augusto, and Joseph Barjis, editors, *Modelling, Simulation, Verification and Validation of Enterprise Information Systems, Proceedings of the 3rd International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems, MSVVEIS 2005, In conjunction with ICEIS 2005, Miami, FL, USA, May 2005*. INSTICC Press, 2005.

[GL87] Thierry Gautier and Paul Le Guernic. SIGNAL: A declarative language for synchronous programming of real-time systems. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture, Portland, Oregon, USA, September 14-16, 1987, Proceedings*, volume 274 of *Lecture Notes in Computer Science*, pages 257–277. Springer, 1987.

[GL02] Dimitra Giannakopoulou and Flavio Lerda. From states to transitions: Improving translation of LTL formulae to büchi automata. In Doron A. Peled and Moshe Y. Vardi, editors, *Formal Techniques for Networked and Distributed Systems - FORTE 2002, 22nd IFIP WG 6.1 International Conference Houston, Texas, USA, November 11-14, 2002, Proceedings*, volume 2529 of *Lecture Notes in Computer Science*, pages 308–326. Springer, 2002.

[GO01] Paul Gastin and Denis Oddoux. Fast LTL to büchi automata translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification, 13th International Conference, CAV 2001,*

Paris, France, July 18-22, 2001, Proceedings, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2001.

[GO03] Paul Gastin and Denis Oddoux. LTL with past and two-way very-weak alternating automata. In Branislav Rovan and Peter Vojtás, editors, *Mathematical Foundations of Computer Science 2003, 28th International Symposium, MFCS 2003, Bratislava, Slovakia, August 25-29, 2003, Proceedings*, volume 2747 of *Lecture Notes in Computer Science*, pages 439–448. Springer, 2003.

[Gor22] Felipe Gorostiaga. *Theory and Practice of Stream Runtime Verification for Sequences and Real-Time Event Based Systems*. PhD thesis, Technical University of Madrid, Spain, 2022.

[GPMS20] Dimitra Giannakopoulou, Thomas Pressburger, Anastasia Mavridou, and Johann Schumann. Generation of formal requirements from structured natural language. In *REFSQ*, volume 12045 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2020.

[GPSS80] Dov M. Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In Paul W. Abrahams, Richard J. Lipton, and Stephen R. Bourne, editors, *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, USA, January 1980*, pages 163–173. ACM Press, 1980.

[GS18] Felipe Gorostiaga and César Sánchez. Striver: Stream runtime verification for real-time event-streams. In Christian Colombo and Martin Leucker, editors, *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*, volume 11237 of *Lecture Notes in Computer Science*, pages 282–298. Springer, 2018.

[GS21a] Felipe Gorostiaga and César Sánchez. Nested monitors: Monitors as expressions to build monitors. In Lu Feng and Dana Fisman, editors, *Runtime Verification - 21st International Conference, RV 2021, Virtual Event, October 11-14, 2021, Proceedings*, volume 12974 of *Lecture Notes in Computer Science*, pages 164–183. Springer, 2021.

[GS21b] Felipe Gorostiaga and César Sánchez. Stream runtime verification of real-time event streams with the striver language. *Int. J. Softw. Tools Technol. Transf.*, 23(2):157–183, 2021.

[GV13] Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *Proc. of the 23rd Int'l Joint Conf. on Artificial Intelligence (IJCAI'13)*, pages 854–860. IJCAI/AAAI, 2013.

[HKLS24] Raik Hipler, Hannes Kallwies, Martin Leucker, and César Sánchez. General anticipatory runtime verification. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part II*, volume 14682 of *Lecture Notes in Computer Science*, pages 133–155. Springer, 2024.

[HL11] Martin Hofmann and Martin Lange. *Automatentheorie und Logik*. eXamen.press. Springer, 2011.

[Hod93] Wilfrid Hodges. *Model theory*, volume 42 of *Encyclopedia of mathematics and its applications*. Cambridge University Press, 1993.

[HPU17] Klaus Havelund, Doron Peled, and Dogan Ulus. First order temporal logic monitoring with bdds. In *FMCAD*, pages 116–123. IEEE, 2017.

[HR02] Klaus Havelund and Grigore Roşu. Synthesizing monitors for safety properties. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.

[HS20] Thomas A. Henzinger and N. Ege Saraç. Monitorability under assumptions. In Jyotirmoy Deshmukh and Dejan Nickovic, editors, *Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6-9, 2020, Proceedings*, volume 12399 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2020.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[IEE98] IEEE. IEEE standard for software verification and validation. *IEEE Std 1012-1998*, pages 1–80, 1998.

[Ins13] Project Management Institute. A guide to the project management body of knowledge (pmbok guide). Project Management Institute, 2013.

[KH04] Nathan P. Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems, Sendai, Japan, September 28 - October 2, 2004*, pages 2149–2154. IEEE, 2004.

[KHF19] Sean Kauffman, Klaus Havelund, and Sebastian Fischmeister. Monitorability over unreliable channels. In *RV*, volume 11757 of *Lecture Notes in Computer Science*, pages 256–272. Springer, 2019.

[KHF21] Sean Kauffman, Klaus Havelund, and Sebastian Fischmeister. What can we monitor over unreliable channels? *Int. J. Softw. Tools Technol. Transf.*, 23(4):579–600, 2021.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[KLS22a] Hannes Kallwies, Martin Leucker, and César Sánchez. Symbolic runtime verification for monitoring under uncertainties and assumptions. In Ahmed Bouajjani, Lukás Holík, and Zhilin Wu, editors, *Automated Technology for Verification and Analysis - 20th International Symposium, ATVA 2022, Virtual Event, October 25-28, 2022, Proceedings*, volume 13505 of *Lecture Notes in Computer Science*, pages 117–134. Springer, 2022.

[KLS+22b] Hannes Kallwies, Martin Leucker, Malte Schmitz, Albert Schulz, Daniel Thoma, and Alexander Weiss. Tessla - an ecosystem for runtime verification. In Thao Dang and Volker Stolz, editors, *Runtime Verification - 22nd International Conference, RV 2022, Tbilisi, Georgia, September 28-30, 2022, Proceedings*, volume 13498 of *Lecture Notes in Computer Science*, pages 314–324. Springer, 2022.

[KLS23] Hannes Kallwies, Martin Leucker, and César Sánchez. General anticipatory monitoring for temporal logics on finite traces. In Panagiotis Katsaros and Laura Nenzi, editors, *Runtime Verification - 23rd International Conference, RV 2023, Thessaloniki, Greece, October 3-6, 2023, Proceedings*, volume 14245 of *Lecture Notes in Computer Science*, pages 106–125. Springer, 2023.

[KLSS22] Hannes Kallwies, Martin Leucker, César Sánchez, and Torben Scheffel. Anticipatory recurrent monitoring with uncertainty and assumptions. In Thao Dang and Volker Stolz, editors, *Runtime Verification - 22nd International Conference, RV 2022, Tbilisi, Georgia, September 28-30, 2022, Proceedings*, volume 13498 of *Lecture Notes in Computer Science*, pages 181–199. Springer, 2022.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Boston, third edition, 1997.

[Koy90] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real Time Syst.*, 2(4):255–299, 1990.

[KPD22] Peeyush Kushwaha, Rahul Purandare, and Matthew B. Dwyer. Optimal finite-state monitoring of partial traces. In Thao Dang and Volker Stolz, editors, *Runtime Verification - 22nd International Conference, RV 2022, Tbilisi, Georgia, September 28-30, 2022, Proceedings*, volume

13498 of *Lecture Notes in Computer Science*, pages 124–142. Springer, 2022.

[KS16] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View, Second Edition*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016.

[Leu11] Martin Leucker. Teaching runtime verification. In Sarfraz Khurshid and Koushik Sen, editors, *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, volume 7186 of *Lecture Notes in Computer Science*, pages 34–48. Springer, 2011.

[Leu12] Martin Leucker. Sliding between model checking and runtime verification. In Shaz Qadeer and Serdar Tasiran, editors, *Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers*, volume 7687 of *Lecture Notes in Computer Science*, pages 82–87. Springer, 2012.

[Lov78] Donald W. Loveland. *Automated theorem proving: a logical basis*, volume 6 of *Fundamental studies in computer science*. North-Holland, 1978.

[LPZ85] Orna Lichtenstein, Amir Pnueli, and Lenore D. Zuck. The glory of the past. In Rohit Parikh, editor, *Logics of Programs, Conference, Brooklyn College, New York, NY, USA, June 17-19, 1985, Proceedings*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218. Springer, 1985.

[LS07] Martin Leucker and César Sánchez. Regular linear temporal logic. In *Proc. of the 4th Int'l Colloquium on Theoretical Aspects of Computing (ICTAC'07)*, volume 4711 of *LNCS*, pages 291–305. Springer, 2007.

[LS09] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Log. Algebraic Methods Program.*, 78(5):293–303, 2009.

[LSS+18] Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, and Alexander Schramm. Tessla: runtime verification of non-synchronized real-time streams. In Hisham M. Haddad, Roger L. Wainwright, and Richard Chbeir, editors, *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*, pages 1925–1933. ACM, 2018.

[LSS+19] Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, and Daniel Thoma. Runtime verification for timed event streams with partial information. In Bernd Finkbeiner and Leonardo Mariani, editors, *Runtime Verification - 19th International Conference, RV 2019, Porto,*

Portugal, October 8-11, 2019, Proceedings, volume 11757 of *Lecture Notes in Computer Science*, pages 273–291. Springer, 2019.

[M+56] Edward F Moore et al. Gedanken-experiments on sequential machines. *Automata studies*, 34:129–153, 1956.

[Mag16] Andrea Maglie. *Reactive Java Programming*. Apress, Berkeley, CA, 2016.

[Mar03] Nicolas Markey. Temporal logic with past is exponentially more succinct, concurrency column. *Bull. EATCS*, 79:122–128, 2003.

[Mea55] George H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955.

[MH84] Satoru Miyano and Takeshi Hayashi. Alternating finite automata on omega-words. In Bruno Courcelle, editor, *CAAP'84, 9th Colloquium on Trees in Algebra and Programming, Bordeaux, France, March 5-7, 1984, Proceedings*, pages 195–210. Cambridge University Press, 1984.

[Min01] Antoine Miné. The octagon abstract domain. In Elizabeth Burd, Peter Aiken, and Rainer Koschke, editors, *Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE'01, Stuttgart, Germany, October 2-5, 2001*, page 310. IEEE Computer Society, 2001.

[Mis99] Mishap Investigation Board. Mars Climate Orbiter Mishap Investigation Board Phase I Report November 10, 1999, 1999.

[MN04] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In Yassine Lakhnech and Sergio Yovine, editors, *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings*, volume 3253 of *Lecture Notes in Computer Science*, pages 152–166. Springer, 2004.

[MNP06] Oded Maler, Dejan Nickovic, and Amir Pnueli. From MITL to timed automata. In Eugene Asarin and Patricia Bouyer, editors, *Formal Modeling and Analysis of Timed Systems, 4th International Conference, FORMATS 2006, Paris, France, September 25-27, 2006, Proceedings*, volume 4202 of *Lecture Notes in Computer Science*, pages 274–289. Springer, 2006.

[MP79] Zohar Manna and Amir Pnueli. The modal logic of programs. In Hermann A. Maurer, editor, *Automata, Languages and Programming, 6th Colloquium, Graz, Austria, July 16-20, 1979, Proceedings*, volume 71 of *Lecture Notes in Computer Science*, pages 385–409. Springer, 1979.

[MP91] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems*. Springer-Verlag, 1991.

[MP92] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems - specification*. Springer, 1992.

[Nip10] Tobias Nipkow. Linear quantifier elimination. *J. Autom. Reason.*, 45(2):189–212, 2010.

[ØH07] Peter Øhrstrøm and Per Hasle. *Temporal logic: From ancient ideas to artificial intelligence*, volume 57. Springer Science & Business Media, 2007.

[Øhr19] Peter Øhrstrøm. *The Significance of the Contributions of A.N.Prior and Jerzy Łoś in the Early History of Modern Temporal Logic*. Logic and Philosophy of Time. Aalborg Universitetsforlag, 1 edition, 2019.

[Ore44] Oystein Ore. Galois connexions. *Transactions of the American mathematical society*, 55:493–513, 1944.

[OW06] Joël Ouaknine and James Worrell. On metric temporal logic and faulty turing machines. In Luca Aceto and Anna Ingólfsdóttir, editors, *Foundations of Software Science and Computation Structures, 9th International Conference, FOSSACS 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25-31, 2006, Proceedings*, volume 3921 of *Lecture Notes in Computer Science*, pages 217–230. Springer, 2006.

[PGMN10] Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot: A hard real-time runtime monitor. In Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Roşu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 2010.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[Pre29] M Presburger. Über die vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervortritt. In *Comptes Rendus du I. congrès de Mathématiciens des Pays Slaves*, pages 92–101, 1929.

[Pri58] A. N. Prior. The syntax of time-distinctions. *Franciscan Studies*, 18(2):105–120, 1958.

[PZ06] Amir Pnueli and Aleksandr Zaks. PSL model checking and run-time verification via testers. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, volume 4085 of *Lecture Notes in Computer Science*, pages 573–586. Springer, 2006.

[RCC92] David A. Randell, Zhan Cui, and Anthony G. Cohn. A spatial logic based on regions and connection. In Bernhard Nebel, Charles Rich, and William R. Swartout, editors, *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92). Cambridge, MA, USA, October 25-29, 1992*, pages 165–176. Morgan Kaufmann, 1992.

[Ric53] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, 74(2):358–366, 1953.

[RRS14] Thomas Reinbacher, Kristin Y. Rozier, and Johann Schumann. Temporal-logic based runtime observer pairs for system health management of real-time systems. In *Proc. 20th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14)*, volume 8413 of *LNCS*, pages 357–372. Springer, 2014.

[RU71] Nicholas Rescher and Alasdair Urquhart. *Temporal logic*. Springer Vienna, Vienna, 1971.

[RY20] Xavier Rival and Kwangkeun Yi. *Introduction to static analysis: an abstract interpretation perspective*. Mit Press, 2020.

[SB00] Fabio Somenzi and Roderick Bloem. Efficient büchi automata from LTL formulae. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 248–263. Springer, 2000.

[SBS+11] Scott D. Stoller, Ezio Bartocci, Justin Seyster, Radu Grosu, Klaus Havelund, Scott A. Smolka, and Erez Zadok. Runtime verification with state estimation. In Sarfraz Khurshid and Koushik Sen, editors, *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, volume 7186 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 2011.

[Sch22] Torben Scheffel. *Expressiveness and complexity of stream-based specification languages*. PhD thesis, University of Lübeck, Germany, 2022.

[Sch24] Malte Schmitz. *Efficient Implementation of Stream Transformations.* PhD thesis, University of Lübeck, Germany, 2024.

[SL10] César Sánchez and Martin Leucker. Regular linear temporal logic with past. In *Proc. of the 11th Int'l Conf. on Verification, Model Checking, and Abstract Interpretation, (VMCAI'10)*, volume 5944 of *LNCS*, pages 295–311. Springer, 2010.

[SQ18] Yassamine Seladji and Zheng Qu. Polyhedron over-approximation for complexity reduction in static analysis. *Int. J. Comput. Math. Comput. Syst. Theory*, 3(4):215–229, 2018.

[SSA+19] César Sánchez, Gerardo Schneider, Wolfgang Ahrendt, Ezio Bartocci, Domenico Bianculli, Christian Colombo, Yliès Falcone, Adrian Francalanza, Srdan Krstic, João M. Lourenço, Dejan Nickovic, Gordon J. Pace, José Rufino, Julien Signoles, Dmitriy Traytel, and Alexander Weiss. A survey of challenges for runtime verification from advanced application domains (beyond software). *Formal Methods Syst. Des.*, 54(3):279–335, 2019.

[Str80] Gilbert Strang. Linear algebra and its applications. *Academic Press, New York*, 1980.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955.

[THK23] Rania Taleb, Sylvain Hallé, and Raphaël Khoury. Uncertainty in runtime verification: A survey. *Computer Science Review*, 50:100594, 2023.

[Var95] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In Faron Moller and Graham M. Birtwistle, editors, *Logics for Concurrency - Structure versus Automata (8th Banff Higher Order Workshop, Banff, Canada, August 27 - September 3, 1995, Proceedings)*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer, 1995.

[Var11] Moshe Y. Vardi. The rise and fall of linear time logic. *2nd Int'l Symp. on Games, Automata, Logics and Formal Verification*, 2011.

[WAH19] Masaki Waga, Étienne André, and Ichiro Hasuo. Symbolic monitoring against specifications parametric in time and data. In *Proc. of CAV'19(1)*, volume 11561 of *LNCS*, pages 520–539. Springer, 2019.

[Wol81] Pierre Wolper. Temporal logic can be more expressive. In *22nd Annual Symposium on Foundations of Computer Science, Nashville, Tennessee, USA, 28-30 October 1981*, pages 340–348. IEEE Computer Society, 1981.

[Wol00] Pierre Wolper. Constructing automata from temporal logic formulas: A tutorial. In Ed Brinksma, Holger Hermanns, and Joost-Pieter Katoen, editors, *Lectures on Formal Methods and Performance Analysis, First EEF/Euro Summer School on Trends in Computer Science, Berg en Dal, The Netherlands, July 3-7, 2000, Revised Lectures*, volume 2090 of *Lecture Notes in Computer Science*, pages 261–277. Springer, 2000.

[ZJLS00] Jun Zhang, Karl Henrik Johansson, John Lygeros, and Shankar Sastry. Dynamical systems revisited: Hybrid systems with zeno executions. In *HSCC*, volume 1790 of *Lecture Notes in Computer Science*, pages 451–464. Springer, 2000.

Index