Faculty of Natural Science and Technology I
Department of Computer Science, Saarland University

Bachelors's Program in Computer Science

Bachelor's Thesis

Optimizing LOLA Specifications

---

submitted by

Mark Timon Hüneberg

on 16.10.2015

Supervisor
Prof. Bernd Finkbeiner, Ph.D.

Reviewers
Prof. Bernd Finkbeiner, Ph.D.
Prof. Dr.-Ing. Holger Hermanns

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, ...
(Datum / Date) (Unterschrift / Signature)

Contents

Abstract

LOLA is a stream-based specification language for the online monitoring of synchronous systems. For this, LOLA offers two main functions. On the one hand it can serve as a verification tool and check, whether a single execution path satisfies a certain property. On the other hand it allows for the collection of statistics about the system execution. The monitoring algorithm can become inefficient, if LOLA specifications contain streams that depend on their own future values. In this thesis we will present methods to remove such forward-recursion from LOLA specifications in order to optimize the performance of the monitoring algorithm. We will use an automata-theoretic approach to optimize specifications, which are supposed to check properties, and we will use a mathematical approach to optimize specification that collect statistics. We will also present our implementation of these methods and illustrate experimental results to demonstrate the effectiveness of our optimization procedures.

1 Introduction

In recent times, the role of engineered systems in everyday life grew heavily in importance. Especially if those systems operate in safety-critical environments, verification becomes a crucial part of the development process. *Runtime Monitoring* is a widely used method that verifies, whether a single execution path of a system satisfies a specification. Runtime monitoring can be seen as an alternative to formal verification methods (e.g. *Model Checking*), where the correctness of every possible execution path is verified. Such methods actually prove correctness, but unfortunately they do not scale very well for complex systems and are only able to verify the correctness of a formal model of the actual system. Runtime monitoring verifies the correctness of a single execution path and thus it scales much better. Here the system is usually run in a simulation setup and emits internal data in the form of an execution trace, which is then processed by a *Monitor*. Thus the actual system is checked and not merely a formal model of the system. In general there are two different ways of connecting the system execution with the monitor.
The first one is called *Offline Monitoring*. Here the system is run until it terminates and its complete execution trace is stored. When the execution of the system has finished, the stored trace can be checked for violations of the specification. Especially when checking for the satisfaction of temporal properties, this approach gives the advantage of virtually having random access to every position of the trace. As an example, comparing the value of a variable at the start and at the end of the execution would be very easy. The obvious downside of this approach is the possibly large amount of memory needed for storing the whole execution trace, depending on the length of the system run and the amount of data monitored.
An alternative is *Online Monitoring*. Here the execution trace is checked on-the-fly, while it is being generated by the system. This give the possibility of detecting violations of specifications early and reacting accordingly, in order to prevent them from cascading into more critical errors. Usually the system and the monitor are linked so that the execution of the system is halted until the current data chunk is processed by the monitor. Thus the monitor should be as time-efficient as possible. It should also be memory-efficient because memory requirements linear in the trace size would be impractical for very long system runs. Memory requirements constant in the trace size however, would take away the option of random access to earlier trace positions, thus specifications should optimally be defined in a way where they can be checked inductively.
In this thesis we will focus on the online monitoring of *Synchronous Systems*. Those are systems whose execution is defined by a single discrete clock and where all changes happen synchronously and instantaneously with every clock tick. Since the execution trace of such a system can easily be separated into a finite amount of data chunks, each associated with a discrete point in time, online monitoring can be easily applied.
We will consider the specification language LOLA, where specifications are formulated by defining *output streams*, which associate each execution step with a value. LOLA offers the possibility of using streams to compute the truth values of certain properties, but streams can also be used to compute statistics about the system execution. Therefore LOLA does not only serve as a verification device, but also as a powerful tool for the collection of information about the the monitored system. By allowing the combination of different data types, LOLA offers a high expressiveness, including context-free properties. All of this makes LOLA a very powerful tool for the industrial testing and profiling of synchronous systems.
To enable the formulation of temporal properties, streams in LOLA can be defined so that they depend on their own past or future values. The latter case can make the online monitor-

ing algorithm very inefficient, because the current value of the stream cannot be computed, until the referenced execution step is reached. In order to achieve a good performance for the online monitoring algorithm, LOLA streams should be defined so that they can be computed incrementally, i.e. by only referencing the past. However, this is impractical, because the formulation of specifications is often more easier and intuitive when the future is referenced. This motivates the goal of this thesis: We want to find optimization methods, which automatically transform arbitrary LOLA specifications into equivalent specifications that are efficiently monitorable.
In this thesis we show such optimization methods for two classes of LOLA specifications, on the one hand strictly boolean specifications that check, whether a property is satisfied, and on the other hand statistical measures. As we prove later, strictly boolean LOLA specifications have exactly regular expressiveness. Thus we map such specifications to language acceptance problems, which we represent as alternating automata. We then demonstrate that equivalent efficiently monitorable LOLA specifications can be synthesized from alternating automata.
Further, we reduce the problem of optimizing statistical measures to the problem of solving a system of recurrence equations. We then use this to construct simple rewriting rules, which transform specifications that match certain patterns into equivalent efficiently monitorable statistics.
We have implemented the optimization algorithm for boolean LOLA specifications in Java and conducted a series of experiments to evaluate the changes in the efficiency of the monitoring algorithm. The implementation resulted in a compiler that reads an arbitrary LOLA specification and returns an equivalent optimized specification, which is guaranteed to be efficiently monitorable if the input specification is strictly boolean.

1.1 Related Work

LOLA, which was first presented in [1], is a specification language that can be used for Stream Runtime Verification (SRV), i.e. the use of streams to verify, whether temporal properties are satisfied. SRV in general is further examined in [5].
Other stream-based languages for synchronous systems are ESTEREL [4] and SIGNAL [9]. Unlike LOLA however, these languages are mainly designed for the synthesis of systems and are thus fundamentally different from LOLA. In synthesis languages, it makes no sense for streams to reference the future, because in an actual run of the synthesized system the future is not known.
LOLA is also able to gather statistics about the system execution. Formal methods for the collection of such statistics were already proposed in [6].
Runtime Monitoring was initially developed for temporal logic properties, which are usually specified in LTL [15]. Boolean LOLA specifications can be seen as an extension of LTL by interpreting streams as recursive LTL equations. It was shown that extending LTL with past operators does not add new expressiveness [14], which also holds for boolean LOLA specifications.
Another monitoring formalism called EAGLE is presented in [3]. EAGLE is a temporal logic that allows the definition of recursive equations and is thus closely related to the strictly boolean subset of LOLA.
It was shown in [10] that the truth values of past temporal properties can be easily computed inductively. Efficiently monitorable specifications in LOLA follow the same principle.

1.2 Outline

In chapter 2 we will present the specification language LOLA. We will formally define its syntax and semantics according to [1] and give an overview on the corresponding online

monitoring algorithm. Then we will introduce dependency graphs in order to define efficiently monitorable and forward-recursive specifications.
Chapter 3 will deal with a strictly boolean subset of LOLA that is expressive enough to verify regular properties. We will interpret such specifications as language acceptors (where a language is a subset of the possible system execution paths) and use automata to capture the corresponding acceptance conditions. We will then show how to synthesize efficient specifications directly from such automata and arrive at a complete optimization procedure. Finally we will briefly describe some possible extensions for this procedure.
Chapter 4 will focus on statistical measures in LOLA. We will present rewriting rules for simple statistics and gradually extend these rules by considering more complex statistics.
In chapter 5 we will describe our implementation of the algorithm presented in chapter 3. We will then display some experimental results and show how the memory and time consumption of the monitoring algorithm compares for optimized and non-optimized specifications.
Chapter 6 will conclude this thesis and briefly summarize the open problems that this thesis does not solve and that could be a subject of future work.

2 LOLA

2.1 Overview

LOLA [1] is a specification language for the online monitoring of synchronous systems, i.e. systems whose time is defined by a single discrete clock. During the monitoring process, the monitored system ejects its data in the form of streams, i.e. arrays of a certain type T that associate each execution step with a value in T. For example, a stream could describe the value of a register in the system at every step of the execution. Streams that are generated directly by the monitored system are called *input streams*. The set of input streams is also called the *execution trace*. The values of these input streams can be accessed by using the associated *independent stream variables*.
LOLA offers the definition of *dependent stream variables*, which describe the so-called *output streams*. The output streams always have the same length as the input streams and their values are computed by evaluating the corresponding stream variables at each time step. Each stream variable is associated with a *stream expression*, which describes how its value is computed. Stream expressions can reference both dependent and independent stream variables with a relative time offset, which allows the description of temporal properties.
While it is theoretically possible to use LOLA with arbitrary types, we will only discuss the types *integer* and *bool* here. Both types can be used individually to compute either incremental statistics or the truth value of a temporal property, but they can also be combined by using constraints and if-expressions, which results in a very high expressiveness that is at least context-free [1].

2.2 Syntax

We first define the general structure of a LOLA specification.

Definition 2.1 (Specification [1])**.** A LOLA specification is given by a set of variable definitions

$$\begin{aligned} s_1 &= e_1(s_1, ..., s_n, t_1, ..., t_m) \\ &\vdots \\ s_n &= e_1(s_1, ..., s_n, t_1, ..., t_m) \end{aligned}$$

where $s_1, ...s_n$ are dependent stream variables, $t_1, ..., t_m$ are independent stream variables and $e_1, ..., e_n$ are stream expressions that depend on $s_1, ..., s_n$ and $t_1, ..., t_m$. Each stream variable has a type T and its corresponding stream expression must have the same type.
A LOLA specification can also contain an arbitrary number of triggers, which are defined by

$$\textbf{trigger } e$$

where e is a boolean LOLA expression. When the expression associated with a trigger evaluates to true, a notification will be created by the trigger.

Triggers in LOLA serve as a way to generate notifications when properties are violated by the monitored system. Note that the property has to be negated before it can be encoded in the trigger expression, because the trigger waits for it to become true.
We also need to specify how LOLA expressions are built.

Definition 2.2 (Syntax [1])**.** A LOLA expression of type T can have one of the following forms:

- c, where c is a constant of type T
- s, where s is a stream variable
- $f(e_1, ..., e_k)$, where $f : T_1 \times ... \times T_k \to T$ is a k-ary operator and for all $1 \leq i \leq k$, e_i is a LOLA expression of type T_i
- $ite(b, e_1, e_2)$, where b is a boolean LOLA expression and e_1, e_2 are LOLA expressions of type T
- $e[i, c]$, where e is a LOLA expression of type T, c is a constant of type T and i is an integer (that can be negative)

Here $ite(b, e_1, e_2)$ is called an *if-expression* and $e[i, c]$ is called an *offset-expression*. An offset-expression simply evaluates its expression at the relative time i and returns c as default value, if this time is not defined. Expressions of the first two forms are called *atomic expressions*.

2.3 Semantics

In order to evaluate LOLA specifications, we use evaluation models. An evaluation model assigns to each dependent stream variable of type T, a stream of values in T.

Definition 2.3 (Semantics [1])**.** Let S be a LOLA specification that consists of the independent stream variables $t_1, ..., t_m$ and the dependent stream variables $s_1, ..., s_n$. Additionally, let $\tau_1, ..., \tau_m$ be the input streams with length N. Then we call the tuple $\langle \sigma_1, ..., \sigma_n \rangle$ of streams with length N an *evaluation model* for S, if for each variable definition

$$s_i = e_i(s_1, ..., s_n, t_1, ..., t_m),$$

the *associated equations* given by

$$\sigma_i(j) = val(e_i)(j) \quad \forall 0 \leq j < N$$

are satisfied and σ_i has the same type as s_i.

Here *val* denotes the evaluation function, which is inductively defined as follows

- $val(c)(j) = c$, if c is a constant
- $val(t_i)(j) = \tau_i(j)$
- $val(s_i)(j) = \sigma_i(j)$
- $val(f(e_1, ..., e_k))(j) = f(val(e_1)(j), ..., val(e_k)(j))$
- $val(ite(b, e_1, e_2))(j) = if\ val(b)(j)\ then\ val(e_1)(j)\ else\ val(e_2)(j)$

- $val(e[k,c])(j) = \begin{cases} val(e)(j+k) & \text{if } 0 \leq j+k < N \\ c & \text{otherwise} \end{cases}$

Note that a specification can have multiple or none evaluation models. We say that a specification is *well-defined* if it has exactly one evaluation model. A specification is obviously not well-defined if it contains a non-terminating recursive variable definition. However, we are not yet able to formally describe this property. Therefore we will now introduce a method to capture the dependencies between stream variables in the form of a graph. We will later use this dependency graph to detect important properties in the corresponding specification.

Definition 2.4 (Dependency Graph [1])**.** Let S be a LOLA specification. The *dependency graph* for S is a weighted and directed multigraph $G = (V, E)$, where V is the vertex set $\{s_1, ..., s_n, t_1, ..., t_m\}$ given by the set of independent and dependent stream variables, and $E \subseteq V \times \mathbb{Z} \times V$ is the set of weighted edges. The tuple $\langle s_i, w, s_j \rangle$ is an edge in G, if and only if the right hand side of the equation for $\sigma_i(k)$ for some k has $\sigma_j(k+w)$ as a subexpression. Analogously the tuple $\langle s_i, w, t_j \rangle$ is an edge in G, if and only if the right hand side of the equation for $\sigma_i(k)$ for some k has $\tau_j(k+w)$ as a subexpression.

Informally, the dependency graph has an edge with weight w between two stream variables, if the source variable can only be evaluated at time k, after the destination variable was evaluated at time $k+w$. We will now introduce the concept of walks (single cyclic traversals of the graph) so that we can make more precise statements about cyclic dependencies.

Definition 2.5 (Walk [1])**.** Given a weighted directed multigraph $G = (V, E)$, we call a sequence $v_1, e_1, v_2, ..., v_k, e_k, v_{k+1}$ of vertices and edges a *walk* in G, iff $k \geq 1$ and $e_i = \langle v_i, v_{i+1}, w_i \rangle$ for all $1 \leq i \leq k$. The total weight of the walk is then given by $\sum_{i=1}^{k} w_i$. A walk is called *closed* if $v_1 = v_{k+1}$.

The dependency graph can now be used to describe the property we mentioned earlier, which says that every recursion in the specification terminates. We call specifications that satisfy this property well-formed.

Definition 2.6 (Well-Formed [1])**.** A LOLA specification S is called *well-formed* if there is no closed walk with weight 0 in its dependency graph.

As shown in [1], every well-formed LOLA specification is also well-defined.

Theorem 2.1 ([1])**.** *Every LOLA specification that is well-formed is also well-defined.*

The converse does generally not hold. Note that for every well-formed specification the corresponding evaluation model is given directly by the evaluation function *val*. Since we will only discuss well-formed specifications in the later chapters, we will strongly use this fact when proving that two streams are equal.

Example 2.1. Consider a system whose components communicate by sending requests and grants. We want to define a well-formed LOLA specification that verifies, whether each such

request is eventually followed by a grant. Our LOLA specification uses two boolean input streams *grant* and *request* and is defined as follows:

$$\begin{aligned} reqgrant =& ite(request, evgrant, true) \\ evgrant =& grant \vee evgrant[1, false] \\ \textbf{trigger}\ & (\neg reqgrant) \end{aligned}$$

The variable *evgrant* evaluates to *true* whenever *grant* is true somewhere in the future. The variable *reqgrant* evaluates to *true*, if *request* being true implies that *evgrant* is also true. The trigger generates a notification if *reqgrant* evaluates to false in some point, or in other words it generates a notification as soon as a request is encountered that is not followed by a grant. Note that this specification does not check the (context-free) property 'every request has a matching grant', but rather the (regular) property given by the LTL formula

$$\square(request \rightarrow \Diamond(grant)).$$

We will also use the dependency graph to formalize the idea of forward-recursion.

Definition 2.7 (Forward-Recursive)**.** A LOLA specification S is called *forward-recursive* if there is a positive cycle in its dependency graph. A stream variable s in S is called *forward-recursive*, if there is a cycle in the dependency graph of S that contains s.

2.4 Monitoring Algorithm

The LOLA online monitoring algorithm tries to build an evaluation model for the given specification, one execution step at a time. For this, the algorithm maintains two sets: A set R that stores *resolved equations*, i.e. equations of the form $\sigma_i(j) = c$ or $\tau_i(j) = c$, and a set U that stores *unresolved equations*. In other words R serves as a lookup for the previous variable evaluations that are still needed and U describes which variables still need to be evaluated and for which trace position this needs to happen.
At the start of the algorithm both sets are empty. For each execution step j, the following steps happen in order:

1. The current input stream valuation is added to the set U of resolved equations, i.e. as equations of the form $\tau_i(j) = c$.
2. The variable definitions for the current execution step are added to the set R of unresolved equations, i.e. as equations of the form $\sigma_i(j) = val(e_i)(j)$.
3. Each equation in U is then simplified as much as possible. If an equation is solved, i.e. has a constant on the right hand side, it is removed from U and added R. The simplification includes partial evaluation rules and substitutions by solutions in R. In this step also trigger notifications are generated.
4. Finally all equations, which will not be needed in the future, are removed from R. The decision, whether an equation is still needed in the future, can be made by looking at the offsets which are used in the specification to reference the underlying stream variable.

2.5 Efficiently Monitorable Specifications

The LOLA online monitoring algorithm is very efficient for specifications that contain no forward-recursion. If this is the case, every equation in R is removed after a constant amount

of time and thus the memory requirements of the monitoring algorithm are constant in the trace size. It can become much more inefficient, if the specification contains forward-recursively defined stream variables. Consider for example the stream variable definition

$$s = s[1, false]$$

In order to evaluate s at time j, the value of s at time $j+1$ is needed, which in turn needs the value of s at time $j+2$ and so on. In fact, the unresolved equations for s would be contained in U until the end of the algorithm and a new instance is added for each execution step. Thus the memory requirements would be linear in the trace size. We can now introduce the notion of efficiently monitorable specifications.

Definition 2.8. We call a LOLA specification *efficiently monitorable* if the worst case memory requirements for the online monitoring algorithm are constant in the trace size.

As mentioned before this is always the case for specifications without forward-recursion. Thus results the following theorem from [1]:

Theorem 2.2 ([1])**.** *If the dependency graph for a LOLA specification S has no positive cycles, then S is efficiently monitorable.*

The converse of this theorem is in general no true. The only reason for this, however, is the use of partial evaluation in the monitoring algorithm (partial evaluation here includes the rewriting rules for if-expressions with a literal as condition). For example, the following specification obviously has a positive cycle in its dependency graph:

$$s = false \wedge s[1, true]$$

However, in order to evaluate s at time j, the value of s at time $j+1$ is not needed. The partial evaluation rules for the conjunction operator allow for an immediate evaluation to $false$ and thus each equations stays in the set U for only one execution step. We now prove this observation.

Theorem 2.3. *If no partial evaluation is used, the converse of Theorem 2.2 holds.*

Proof. We proof by contradiction that a LOLA specification with positive cycles in its dependency graph can only be efficiently monitorable, if partial evaluation is used. Let $S = \{s_1 = e_1, ..., s_k = e_k\}$ be a an efficiently monitorable LOLA specification, where w.l.o.g. s_1 is part of a closed walk with positive weight w in the dependency graph. Also assume that no partial evaluation is used.
Since the specification is efficiently monitorable, the number of unresolved equations in the memory is bounded by a constant c, which only depends on the size of the specification. We choose the trace length N so that $N > wk+1$. Let $\langle \sigma_1, ..., \sigma_k \rangle$ be an evaluation model for the specification S. At the first position of the trace, the equation $\sigma_1(0) = e_1(0)$ is added to the set U of unresolved equations. Because S has a closed walk with weight w in its dependency graph and every subexpression needs to be evaluated, the above equation remains unresolved until the value of $\sigma_1(w)$ is resolved. In general, for every $i \in \mathbb{N}$ it holds that at time $w * i$, the set $\{\sigma_1(j) = e_1(j) | j \in \mathbb{N} \wedge j \leq i\}$ is a subset of U if $i * w < N - 1$. This can be proved by induction (*). Thus for $i = k$, there are at least $k + 1$ unresolved equations in the memory, which contradicts the assumption.

Proof for (*): Proof by induction over $\mathbb{N}$.
Basis Case: Let $i = 0$. The equation $\sigma_1(0) = e_1(0)$ is added to U. Thus the set $\{\sigma_1(0) = e_1(0)\}$ is a subset of U.
Inductive Step: Assume the claim holds for i-1. Thus at time $(i-1)w$ the set $\{\sigma_1(0) = e_1(0), ..., \sigma_1((i-1)w) = e_1((i-1)w)$ is a subset of U. Because s_1 is part of a closed walk with positive weight w in the dependency graph and every subexpression has to be evaluated, the value of $\sigma_1((i-1)w)$ (and consequently all previous equations) remain unresolved until the value of $\sigma_1(i*w)$ is resolved. Thus at time $i*w$, $\{\sigma_1(0) = e_1(0), ..., \sigma_1((i-1)w) = e_1((i-1)w)$ is still a subset of U. Now the equation $\sigma_1(i*w) = e_1(i*w)$ is added to U and therefore the claim holds for i. □

In the later chapters we will try to optimize LOLA specifications so that they become efficiently monitorable. In practice, it is often not possible to detect, whether partial evaluation will be applied for a LOLA specification. Therefore, by optimization we do not necessarily mean then transformation of not efficiently monitorable into efficiently monitorable specifications, but rather the transformation of specifications with positive cycles in their dependency graphs into specifications without positive cycles, which are by Theorem 2.2 always efficiently monitorable. Thus we may transform specifications that are already efficiently monitorable into different (but equivalent) efficiently monitorable specifications. We will present these transformations in the following chapters.

3 Optimizing Boolean Specifications

In this chapter we focus on the part of LOLA that is able to verify regular properties. This is done by using triggers, which are activated if an associated boolean LOLA expression becomes true. Triggers can be seen as language recognizers, where the accepted language is the set of input stream valuations, for which no trigger is activated at any point in time. As this notion suggests, the verification of regular properties in LOLA can be mapped to a language acceptance problem. Therefore we are able to use automata to capture the regular property encoded by the specification. More specifically, we will build an alternating automaton that is equivalent to a given trigger, and then, by a series of translations, build an equivalent deterministic automaton. By creating a LOLA specification that simulates the runs in this DFA, we can directly synthesize an equivalent efficiently monitorable specification, since the active state in a DFA can be determined by only the previous state and the current input.
It is interesting to note that, as a consequence, we can synthesize efficiently monitorable LOLA specifications directly from every language-recognizing formalism, that has at most regular expressiveness. An example for such a formalism is LTL [15], which is a popular tool to describe specifications for Model Checking and Monitoring.

3.1 Background

In this section we will introduce the different formalisms (mostly automata) that will be used in the course of this chapter.

3.1.1 Propositional Logic

First, we will recall the *first-order propositional logic* (*PL*). A *PL-formula* is built by using boolean connectives to specify a property over a set of *atomic propositions*.

Definition 3.1 (PL-Formula)**.** Let AP be a set of atomic propositions. A PL-formula φ over AP is defined as follows:

$$\varphi ::= \varphi \wedge \varphi | \varphi \vee \varphi | \neg\varphi | p \in AP | true | false$$

We also need to define the semantics of PL-formulas. We say that a formula φ over AP is *modeled* by a subset S of AP, if the formula evaluates to true when $p \in S$ are interpreted as $true$ and $p \in AP \backslash S$ are interpreted as $false$. If S models φ we write $S \models \varphi$. If S does not model φ, then we write $S \not\models \varphi$.

Definition 3.2 (Model)**.** Let AP be a set of atomic propositions and $S \subseteq AP$ be a subset. Further, let φ and ψ be PL-formulas over AP and $p \in AP$. Then:

$$\begin{aligned}
&S \models true && \\
&S \not\models false && \\
&S \models \varphi \wedge \psi && \text{iff } S \models \varphi \text{ and } S \models \psi \\
&S \models \varphi \wedge \psi && \text{iff } S \models \varphi \text{ or } S \models \psi \\
&S \models \neg\varphi && \text{iff } S \not\models \varphi \\
&S \models p && \text{iff } p \in S.
\end{aligned}$$

Now we can use PL-formulas to define recursive alternating automata.

3.1.2 Recursive Two-Way Alternating Automata (R2AA)

Alternating automata [16] can be seen as a generalization of non-deterministic automata. While non-deterministic automata can have existential transitions (that can reach one out of multiple states), alternating automata can also have universal transitions (that reach multiple states at the same time). Consequently, a run in an alternating automaton is a tree rather than a sequence of states, and each path through this tree must be accepting.
An (equally expressive) further generalization of alternating automata are Two-Way Alternating Automata [17], which add the possibility of moving backward on the input. We now introduce recursively defined two-way alternating automata as first presented in [13] and call these *Recursive Two-Way Alternating Automata* or R2AAs. More specifically we will use R2AAs over finite models as in [7].

Definition 3.3 (R2AA [7])**.** A *recursive two-way alternating automaton* (R2AA) for an alphabet $\Sigma = 2^{AP}$, where AP is a set of atomic propositions, can be defined in the following way:

$$\mathcal{A} ::= \varepsilon_{\mathcal{A}} | \langle v, \mathcal{A}, f, g \rangle | \mathcal{A} \wedge \mathcal{A} | \mathcal{A} \vee \mathcal{A},$$

In this notation $\varepsilon_{\mathcal{A}}$ denotes the empty automaton and $\langle v, \mathcal{A}, f, g \rangle$ is a node in the automaton (where v is a PL-formula over AP, $f \in \{acc, rej\}$ denotes, whether the node is accepting or rejecting, and $g \in \{\rightarrow, \leftarrow, \downarrow\}$ denotes the direction of the node). We usually write a node as $n = \langle v, \mathcal{A}, f, g \rangle$, where $\mathcal{A}$ is called the *successor automaton* of n. Every node has a direction $g \in \{\leftarrow, \rightarrow, \downarrow\}$. $\leftarrow$ denotes a *past node*, $\rightarrow$ denotes a *future node* and $\downarrow$ denotes an *atomic node*. Future nodes and past nodes serve to move the current position on the input forward or backward by one, while atomic nodes simply check that a PL-formula is satisfied by the current position of the input. Now we need to clarify when a word over 2^{AP} is accepted by a R2AA. In order to describe runs in R2AAs we use trees as defined in [13].

Definition 3.4 (Tree [7])**.** A *tree* is defined in the following way:

$$T ::= \varepsilon_T | T \cdot T | \langle \langle v, \mathcal{A}, f, g \rangle, T \rangle$$

Here ε_T is the empty tree, $T \cdot T$ is the composition of two trees and $\langle \langle v, \mathcal{A}, f, g \rangle, T \rangle$ is a node in the tree (similar to a node in a R2AA). Note that each tree node $\langle \langle v, \mathcal{A}, f, g \rangle, T \rangle$ is associated with a R2AA node $\langle v, \mathcal{A}, f, g \rangle$. We can now use such trees to describe runs in R2AAs as was done in [7].

Definition 3.5 (Run [7])**.** For a word $w = w_0 ... w_{n-1} \in \Sigma^*$, let $|w|$ denote the length of w. We say that a tree T is a run of w in a R2AA $\mathcal{A}$ at time i, if one of the following holds:

- $\mathcal{A} = \varepsilon_{\mathcal{A}}$ and $T = \varepsilon_T$
- $\mathcal{A} = \langle v, \mathcal{A}', f, \downarrow \rangle$ and $T = \langle \langle v, \mathcal{A}', f, \downarrow \rangle, \varepsilon_T \rangle$ and $w_i \models v$
- $\mathcal{A} = \langle v, \mathcal{A}', f, \rightarrow \rangle$ and $\begin{cases} T = \langle \langle v, \mathcal{A}', f, \rightarrow \rangle, T' \rangle \text{ and} \\ \quad w_i \models v \text{ and } T' \text{ is a run of } w \text{ in } \mathcal{A}' \text{ at } i+1 & \text{, if } i < n-1 \\ T = \langle \langle v, \mathcal{A}', f, \rightarrow \rangle, \varepsilon_T \rangle \text{ and } w_i \models v & \text{, if } i = n \end{cases}$

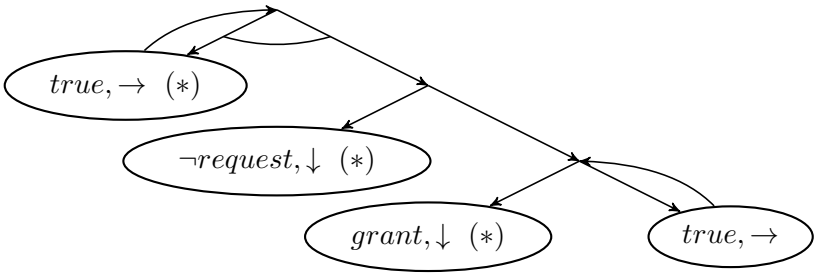

Figure 3.1: R2AA for $\square(request \rightarrow \Diamond grant)$

- $\mathcal{A} = \langle v, \mathcal{A}', f, \leftarrow \rangle$ and $\begin{cases} T = \langle \langle v, \mathcal{A}', f, \leftarrow \rangle, T' \rangle \text{ and} \\ \quad w_i \models v \text{ and } T' \text{ is a run of } w \text{ in } \mathcal{A}' \text{ at } i-1 & \text{, if } i > 0 \\ T = \langle \langle v, \mathcal{A}', f, \leftarrow \rangle, \varepsilon_T \rangle \text{ and } w_i \models v & \text{, if } i = 0 \end{cases}$
- $\mathcal{A} = \mathcal{A}_1 \wedge \mathcal{A}_2$ and $T = T_1 \cdot T_2$, where T_1 is a run of w in $\mathcal{A}_1$ and T_2 is a run of w in $\mathcal{A}_2$
- $\mathcal{A} = \mathcal{A}_1 \vee \mathcal{A}_2$ and T_1 is a run of w in $\mathcal{A}_1$ or T_2 is a run of w in $\mathcal{A}_2$

We say that $\mathcal{A}$ *accepts* w at time i, if there is a run T of w in $\mathcal{A}$ at time i such that every branch in T ends with an accepting node.

The cyclic definition of R2AAs allows for the construction of automata that depend on themselves in the same way as non-well-formed LOLA streams depend on themselves. Therefore we define a class of R2AAs, whose runs terminate for every input. We say that such automata are *progressing* [8].

Definition 3.6 (Progressing [8])**.** A R2AA is called *progressing*, if no subautomaton can reach itself with the same time.

Example 3.1. Consider the R2AA $\mathcal{A}$, defined as

- $\mathcal{A} = \langle true, \mathcal{A}, acc, \rightarrow \rangle \wedge \mathcal{A}'$
- $\mathcal{A}' = \langle \neg request, \varepsilon_{\mathcal{A}}, acc, \downarrow \rangle \vee \mathcal{A}''$
- $\mathcal{A}'' = \langle grant, \varepsilon_{\mathcal{A}}, acc, \downarrow \rangle \vee \langle true, \mathcal{A}'', rej, \rightarrow \rangle$

$\mathcal{A}$ accepts an input, if and only if the LTL formula $\square(request \rightarrow \Diamond grant)$ is satisfied by the input. The automaton $\mathcal{A}$ is illustrated in figure 3.1. In the graphical representation, an arc between two transitions denotes a conjunction, and (*) denotes that the node is accepting.

3.1.3 Classical Two-Way Alternating Automata (2AA)

We will later use a construction from [8] that allows to transform two-way alternating automata into equivalent non-deterministic automata. However, this construction does not use recursive automata, which demands the introduction of a non-recursive definition for alternating automata. In the following we will simply recall the 2AA-Definitions from [8]. However, while they use automata over finite and infinite words, we will confine their definitions to finite words.

Definition 3.7 (2AA [8])**.** A two-way alternating automaton (2AA) can be defined as follows: $A = (Q, \Sigma, \Delta, I, F)$, where

- Q is a set of states,
- Σ is the alphabet,
- $\Delta : Q \to 2^{2^Q \times 2^\Sigma \times 2^Q}$ is a transition function, that maps each state to a disjunction over tuples of
  - a conjunction of past states,
  - a set of signs in Σ, and
  - a conjunction of future states,
- $I \in 2^{2^Q}$ is the initial state, which is given by a disjunction over conjunctions of states, and
- $F \subseteq Q$ is a set of accepting states.

Note that, while R2AAs make progress on the input when reaching nodes, 2AAs make progress on the input when taking transitions between states. The initial state is a boolean formula over states (or more formally a formula in $\mathbb{B}^+(Q)$). In our definition, we denote such formulas by their disjunctive normal forms, which are represented by nested sets. The transition function Δ maps each state to such a formula, but represents each conjunction set by a tuple $(\overleftarrow{X}, \alpha, \overrightarrow{X})$. α simply denotes the set of signs in the alphabet Σ that can be used to reach this conjunction of states. The actual reachable states need to be separated into two sets $\overleftarrow{X}$ and $\overrightarrow{X}$, because we are dealing with two-way automata. If a 2AA is currently in state q and there is a tuple $(\overleftarrow{X}, \alpha, \overrightarrow{X}) \in \Delta(q)$ such that the current position of the input word is an element of α, then q can simultaneously reach the states in both $\overleftarrow{X}$ (by moving backward on the input) and $\overrightarrow{X}$ (by moving forward on the input). We can now describe this formally by defining runs in 2AAs. Instead of recursively defined trees, we will use Q-forests to represent runs.

Definition 3.8 (Q-Forest [8])**.** For a 2AA $A = (Q, \Sigma, \Delta, I, F)$, a *$Q$-Forest* $(V, E, \sigma, \vartheta)$ is a tuple, where V is a set of nodes, $E \subseteq V \times V$ describes the edges between nodes, $\sigma : V \to Q$ maps each node to a state in Q, and $\vartheta : V \to \mathbb{N}$ maps each node to a time in $\mathbb{N}$. For each node $x \in V$, $\overleftarrow{E}(x) = \{y \in E(x) | \vartheta(y) = \vartheta(x) - 1\}$ denotes the set of left children of x and $\overrightarrow{E}(x) = \{y \in E(x) | \vartheta(y) = \vartheta(x) + 1\}$ denotes the set of right children of x.

A Q-forest is basically a set of $(Q, \mathbb{N})$-labeled trees. We can now use Q-forests to describe runs of words in Σ^*.

Definition 3.9 (Run [8])**.** Given a 2AA $A = (Q, \Sigma, \Delta, I, F)$, a word $w \in \Sigma^*$ and a Q-forest $F = (V, E, \sigma, \vartheta)$, we call F a run of w in A, if all of the following holds:

- $\vartheta(Q) \subseteq \{-1, 0, ..., |w| - 1, |w|\}$
- $\sigma(\Gamma) \in I$ and $\vartheta(\Gamma) \subseteq \{0\}$, where Γ is the set of roots in the forest
- for every node x in V so that $\vartheta(x) \in \{-1, |w|\}$, the set of children is empty ($E(x) = \emptyset$)
- for every node x in V so that $0 \leq \vartheta(x) \leq |w| - 1$, there is a subset $\alpha \subseteq \Sigma$ such that $w_{\vartheta(x)} \in \alpha$ and $(\sigma(\overleftarrow{E}(x)), \alpha, \sigma(\overrightarrow{E}(x))) \in \Delta(\sigma(x))$.

We say that a run is *accepting* if for every node x with $\vartheta(x) \in \{-1, |w|\}$ it holds that $\sigma(x) \in F$.

These definitions also allow to define the notion of progressing automata much more formally:

Definition 3.10 (Progressing [8])**.** A run $(V, E, \sigma, \vartheta)$ has a *loop*, if for two nodes $x, y \in V$ in the same branch it holds that $\sigma(x) = \sigma(y)$ and $\vartheta(x) = \vartheta(y)$.
A 2AA is called *progressing* if all runs are loop-free.

3.1.4 NFAs and DFAs

Finally we will recall the definitions of *Deterministic Finite Automata* (DFAs) and *Non-Deterministic Finite Automata* (NFAs).

Definition 3.11. A non-deterministic finite automaton (NFA) can be defined as follows:
$N = (Q, \Sigma, \Delta, s_0, F)$, where

- Q is the set of states,
- Σ is the alphabet,
- $\Delta : Q \times \Sigma \to 2^Q$ is the transition function,
- $s_0 \in Q$ is the initial state, and
- F is the set of accepting states.

We call $s_0 \xrightarrow{w_0} s_1 \xrightarrow{w_1} ... \xrightarrow{w_{n-1}} s_n$ a run of $w = w_0...w_{n-1} \in \Sigma^*$, if $s_{i+1} \in \Delta(s_i, w_i)$ is satisfied for all i with $0 \leq i < n$. The run is called *accepting* if $s_n \in F$. We say that a NFA N *accepts* a word w if there is an accepting run of w in N.

Definition 3.12. A deterministic finite automaton (DFA) can be defined as follows:
$A = (Q, \Sigma, \delta, s_0, F)$, where

- Q is the set of states,
- Σ is the alphabet,
- $\delta : Q \times \Sigma \to Q$ is the transition function,
- $s_0 \in Q$ is the initial state, and
- F is the set of accepting states.

We call $q_0 \xrightarrow{w_0} q_1 \xrightarrow{w_1} ... \xrightarrow{w_{n-1}} q_n$ a run of $w = w_0...w_{n-1} \in \Sigma^*$, if $\delta(q_i, w_i) = q_{i+1}$ is satisfied for all i with $0 \leq i < n$. The run is called *accepting* if $q_n \in F$. We say that a state q_0 *accepts* a word w if there is an accepting run of w in A starting from q_0. We can define the *language* of a state q_0 as $\mathcal{L}(q_0) = \{w \in \Sigma^* | q_0 \text{ accepts } w\}$. The language of A is then defined as $\mathcal{L}(A) = \mathcal{L}(s_0)$

3.2 Overview

Before we show the actual algorithm, we will first give an overview on the strategy that will be used. This strategy is also depicted in figure 3.2. The original LOLA specification contains a set of triggers, each of which specifies a set of acceptance conditions for the input. Our first goal is to create an equivalent alternating automaton for each trigger, which will be translated into an equivalent DFA. It would also be possible to build an automaton that combines the acceptance conditions of all triggers. However, we chose to build a set of automata, because each trigger usually defines a distinct property and this distinction should

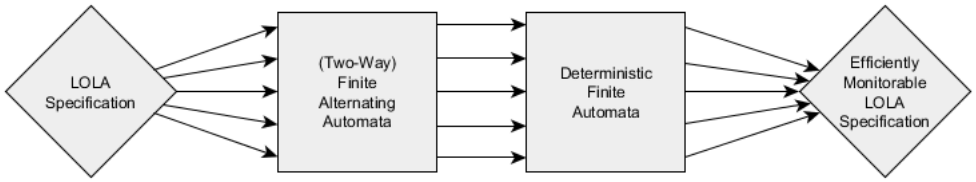

Figure 3.2: Translation Strategy

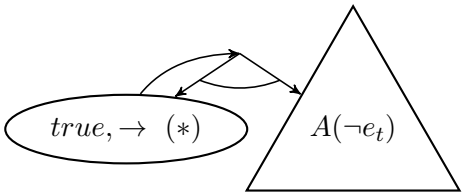

Figure 3.3: R2AA for a trigger

not be lost. Finally the set of DFAs will be transformed into a single efficiently monitorable LOLA specification that contains a separate trigger for each DFA (and thus for each trigger in the original specification).
As the first automata-theoretic representation of the original LOLA specification we chose R2AAs, because this allows to recursively build a R2AA for each LOLA expression, which only accepts an input if the expression evaluates to true under that input. We can then build a R2AA for each trigger by building the automaton for the recursive trigger expression. Finally we transform each R2AA into a non-recursive 2AA and use the algorithm from [8], which constructs an equivalent NFA. Each NFA can then simply be translated into an equivalent DFA by using the classic subset construction. For a 2AA with n, the constructed DFA has a $\mathcal{O}(2^{2^{2n}})$ states, because two subset constructions are used. However, since our final representation is a DFA, we can simply minimize this DFA by using the Myhill-Nerode theorem.

3.3 LOLA to R2AA

We are now ready to build an equivalent R2AA for a LOLA specification, or more specifically for each trigger in a LOLA specification. Each such trigger has the form

trigger e_t

where e_t is the trigger expression. The equivalent R2AA should accept an input, if the trigger is never activated or in other words, if e_t does not evaluate to *true* for any suffix of the input. To accomplish this we construct the R2AA depicted in figure 3.3. It is a conjunction of the automaton $A(\neg e_t)$ and an accepting future node that simply moves the position on the input forward by 1 and loops back to the conjunction. Thus it accepts an input at a time i, if and only if the input is accepted by the automaton $A(\neg e_t)$ at every time j so that $j \geq i$. However, for this R2AA to actually encode the trigger, the automaton $A(\neg e_t)$ has to accept an input at time i exactly if the LOLA expression $\neg e_t$ evaluates to *true* under the same input at time i. In other words we need to show how to construct an automaton $A(e)$ for every LOLA expression e, so that $A(e)$ accepts an input at time i, if e evaluates to *true* at time i under that input. Fortunately this construction is very easy if the LOLA expression satisfies certain properties:

Definition 3.13 (R2AA-Compatible)**.** We call a LOLA expression *R2AA-compatible* if all of the following properties are satisfied:

1. The expression is of boolean type and does not depend on any non-boolean expression
2. It does not contain any if-expressions
3. It does not contain any offset-expression with an offset bigger than 1 or smaller than -1
4. It contains only negations in front of atomic expressions

As we will show in the next paragraphs, every boolean LOLA expression can be rewritten as an equivalent R2AA-compatible expression. For a R2AA-compatible LOLA expression it is then easy to recursively build a R2AA (that has the same structure as the syntax tree of the expression).

3.3.1 Removing If-Expressions

We will first show how to remove if-expressions from a LOLA-expression. This is very easy for boolean expressions because then the if-expressions simply becomes a boolean property that can be described by a boolean formula. We simply rewrite each if-expression $ite(e_1, e_2, e_3)$ into the boolean expression $(\neg e_1 \vee e_2) \wedge (e_1 \vee e_3)$. This equivalence is trivial and not proven here.

3.3.2 Unfolding Offsets

Alternating automata can only move one position forward or backward at a time on the input, therefore we cannot directly build such automata for LOLA expressions with offsets bigger than 1 or smaller than -1. We solve this problem by unfolding the offset-expressions. Here unfolding means the transformation of a k-offset-expression into a k-long series of 1-offset-expressions (and the same for negative offsets). In the following we will only prove this transformation for positive offsets, but the method for negative offsets works completely analogously and is thus not shown here. We first introduce a notation for a series of n offset-expressions with offset k.

[Offset-Series]

Definition 3.14. Let the expression $e[k, c]^n$ with $n \in \mathbb{N}$ be defined as

- $e[k, c]^0 = e$
- $e[k, c]^n = (e[k, c]^{n-1})[k, c]$

We now use this notation to show that every offset-expression with offset k can be rewritten as a series of offset-expressions with offset 1.

Theorem 3.1 (Offset Unfolding)**.** *Given a well-formed LOLA expression e and a constant c, for every $k \in \mathbb{N}$ the expressions $e[1, c]^k$ and $e[k, c]$ are equivalent.*

Proof. In order to prove the theorem, we need to show that $val(e[1, c]^k)(j) = val(e[k, c])(j)$ holds for every $j, k \in \mathbb{N}$ with $0 \leq j < N$, where N is the trace size. We prove this by induction over $k \in \mathbb{N}$.

- *Base case:* Let $k = 0$. Then:

$$\begin{aligned} val(e[1, c]^k)(j) &= val(e[1, c]^0)(j) \\ &\stackrel{\text{def}}{=} val(e)(j) \end{aligned}$$

Also:

$$\begin{aligned} val(e[k,c])(j) &= val(e[0,c])(j) \\ &\stackrel{\text{def}}{=} \begin{cases} val(e)(j+0) & \text{if } 0 \leq j+0 \leq N \\ c & \text{otherwise} \end{cases} \\ &= val(e)(j) \qquad \text{(because } 0 \leq j < N\text{)} \end{aligned}$$

- *Induction step:* Let $k \in \mathbb{N}_{>0}$. Assume the claim holds for all $k' < k$. Then:

$$\begin{aligned} val(e[1,c]^k)(j) &\stackrel{\text{def}}{=} val((e[1,c]^{k-1})[1,c])(j) \\ &\stackrel{\text{def}}{=} \begin{cases} val(e[1,c]^{k-1})(j+1) & \text{if } 0 \leq j+1 < N \\ c & \text{otherwise} \end{cases} \end{aligned}$$

  There are now two possible cases:

  - *Case 1:* $j+1 \geq N$. Then we get

$$val(e[1,c]^k)(j) = c.$$

    Since $k > 0$ and $j+1 \geq N$, it follows that $j+k \geq N$. We get

$$\begin{aligned} val(e[k,c])(j) &\stackrel{\text{def}}{=} \begin{cases} val(e)(j+k) & \text{if } 0 \leq j+k < N \\ c & \text{otherwise} \end{cases} \\ &= c \end{aligned}$$

    Thus the claims holds.

  - *Case 2:* $j+1 < N$. Then by using the induction hypthesis (IH) we get

$$\begin{aligned} val(e[1,c]^k)(j) &\stackrel{\text{def}}{=} val(e[1,c]^{k-1})(j+1) \\ &\stackrel{\text{IH}}{=} val(e[k-1,c])(j+1) \\ &= \begin{cases} val(e)(j+1+k-1) & \text{if } 0 \leq j+1+k-1 < N \\ c & \text{otherwise} \end{cases} \\ &= \begin{cases} val(e)(j+k) & \text{if } 0 \leq j+k < N \\ c & \text{otherwise} \end{cases} \\ &\stackrel{\text{def}}{=} val(e[k,c])(j) \end{aligned}$$

    Thus the claim holds.

□

3.3.3 Pushing Negations Down

Negations on a high level cannot easily be encoded in alternating automata. Therefore we will need to bring each LOLA expression into a form where negations only appear in front of atomic expressions. In order to do this we introduce a unary operator similar to the dual in [16], which recursively pushes as single negation down.

Definition 3.15. For a LOLA expression e, let $\bar{e}$ denote the following:

- $\overline{(e_1 \wedge e_2)} = \overline{e_1} \vee \overline{e_2}$
- $\overline{(e_1 \vee e_2)} = \overline{e_1} \wedge \overline{e_2}$
- $\overline{\neg e} = e$
- $\overline{true} = false$
- $\overline{false} = true$
- $\overline{t} = \neg t$, if t is an input stream
- $\overline{s} = \neg s$, if s is an output stream
- $\overline{e[1, c]} = \overline{e}[1, \neg c]$
- $\overline{e[-1, c]} = \overline{e}[1, \neg c]$

In order to be able to use them interchangeably, we need to show that the values of $\neg e$ and $\overline{e}$ are always the same for every LOLA expression e.

Theorem 3.2. *Let N be the trace length and e be a boolean LOLA expression. For every input stream valuation it holds that $val(\overline{e})(i) = val(\neg e)(i)$ for all $0 \leq i < N$.*

Proof. We prove the theorem by induction over the structure of e.

- *Base cases:*
  - Let $e = true$, then $val(\overline{true})(i) \stackrel{\text{def}}{=} val(false)(i) = val(\neg true)(i)$.
  - Let $e = false$, then $val(\overline{false})(i) \stackrel{\text{def}}{=} val(true)(i) = val(\neg false)(i)$.
  - Let $e = t$, where t is an input stream. Then $val(\overline{t})(i) = val(\neg t)(i)$ by definition.
  - Let $e = s$, where s is an output stream. Then $val(\overline{s})(i) = val(\neg s)(i)$ by definition.
- *Induction Step:*
  - Let $e = e_1 \wedge e_2$. Then

$$\begin{aligned} val(\overline{e_1 \wedge e_2})(i) &\stackrel{\text{def}}{=} val(\overline{e_1} \vee \overline{e_2})(i) \\ &\stackrel{\text{def}}{=} val(\overline{e_1})(i) \vee val(\overline{e_2})(i) \\ &\stackrel{\text{IH}}{=} val(\neg e_1)(i) \vee val(\neg e_2)(i) \\ &\stackrel{\text{def}}{=} \neg val(e_1)(i) \vee \neg val(e_2)(i) \\ &= \neg(val(e_1)(i) \wedge val(e_2)(i)) \\ &\stackrel{\text{def}}{=} \neg(val(e_1 \wedge e_2)(i)) \\ &\stackrel{\text{def}}{=} val(\neg(e_1 \wedge e_2))(i) \end{aligned}$$

– Let $e = e_1 \vee e_2$. Then

$$\begin{aligned} val(\overline{e_1 \vee e_2})(i) &\stackrel{\text{def}}{=} val(\overline{e_1} \wedge \overline{e_2})(i) \\ &\stackrel{\text{def}}{=} val(\overline{e_1})(i) \wedge val(\overline{e_2})(i) \\ &\stackrel{\text{IH}}{=} val(\neg e_1)(i) \wedge val(\neg e_2)(i) \\ &\stackrel{\text{def}}{=} \neg val(e_1)(i) \wedge \neg val(e_2)(i) \\ &= \neg(val(e_1)(i) \vee val(e_2)(i)) \\ &\stackrel{\text{def}}{=} \neg(val(e_1 \vee e_2)(i)) \\ &\stackrel{\text{def}}{=} val(\neg(e_1 \vee e_2))(i) \end{aligned}$$

– Let $e = \neg e'$. Then

$$\begin{aligned} val(\overline{\neg e'})(i) &\stackrel{\text{def}}{=} val(e')(i) \\ &= \neg(\neg(val(e')(i))) \\ &\stackrel{\text{def}}{=} \neg(val(\neg e')(i)) \\ &\stackrel{\text{def}}{=} val(\neg(\neg e'))(i) \end{aligned}$$

– Let $e = e'[1, c]$. We proof $val(\overline{e})(i) = val(\neg e)(i)$ for all $0 \leq i < N$ by backwards induction over $\{0, .., N-1\}$

  * *Base Case:* Let $i = N - 1$. Then

$$\begin{aligned} val(\overline{e'[1, c]})(N-1) &\stackrel{\text{def}}{=} val(\overline{e'}[1, \neg c](N-1) \\ &\stackrel{\text{def}}{=} \neg c \\ &\stackrel{\text{def}}{=} \neg val(e'[1, c])(N-1) \\ &\stackrel{\text{def}}{=} val(\neg(e'[1, c]))(N-1) \end{aligned}$$

  * *Induction Step:* Let $0 \leq i < N - 1$. Then

$$\begin{aligned} val(\overline{e'[1, c]})(i) &\stackrel{\text{def}}{=} val(\overline{e'}[1, \neg c](i) \\ &\stackrel{\text{def}}{=} val(\overline{e'})(i+1) \\ &\stackrel{\text{IH}}{=} val(\neg e')(i+1) \\ &= \neg(val(e')(i+1)) \\ &\stackrel{\text{def}}{=} \neg(val(e'[1, c])(i)) \\ &\stackrel{\text{def}}{=} val(\neg(e'[1, c]))(i) \end{aligned}$$

– Let $e = e'[-1, c]$. We proof $val(\overline{e})(i) = val(\neg e)(i)$ for all $0 \leq i < N$ by induction over $\{0, .., N-1\}$

  * *Base Case:* Let $i = 0$. Then

$$\begin{aligned} val(\overline{e'[-1, c]})(N) &\stackrel{\text{def}}{=} val(\overline{e'}[-1, \neg c](0) \\ &\stackrel{\text{def}}{=} \neg c \\ &\stackrel{\text{def}}{=} \neg val(e'[-1, c])(0) \\ &\stackrel{\text{def}}{=} val(\neg(e'[-1, c]))(0) \end{aligned}$$

∗ *Induction Step:* Let $0 < i < N$. Then

$$\begin{aligned} val(\overline{e'[-1,c]})(i) &\stackrel{\text{def}}{=} val(\overline{e'}[-1,\neg c](i) \\ &\stackrel{\text{def}}{=} val(\overline{e'})(i-1) \\ &\stackrel{\text{IH}}{=} val(\neg e')(i-1) \\ &= \neg(val(e')(i-1)) \\ &\stackrel{\text{def}}{=} \neg(val(e'[-1,c])(i)) \\ &\stackrel{\text{def}}{=} val(\neg(e'[-1,c]))(i) \end{aligned}$$

□

3.3.4 LOLA Expression to R2AA

We can now use the previous results to show that we can build an equivalent R2AA for every possible LOLA expression. This R2AA should accept an input stream valuation, if and only if the expression evaluates to true under this valuation. In order to make the automaton recognize input stream valuations, we define its alphabet as the powerset of the input stream variables. The i-th letter of a word w then encodes the set of input streams, which are true at execution step i.

Theorem 3.3. *Let S be a well-formed and strictly boolean LOLA specification, $\tau = \{\tau_1, ..., \tau_m\}$ be a set of boolean input streams of length N and $\mathcal{T} = \{t_1, ..., t_m\}$ be the corresponding input stream variables. For every LOLA expression e in S there is an equivalent progressing R2AA $A(e)$ over the alphabet $2^{\mathcal{T}}$, so that the following holds for all $i \in \{0, ..., N-1\}$:*
$val(e)(i) = true$, iff $A(e)$ accepts the word $w = w_0...w_{N-1}$ at time i, where
$w_k = \{t_j \in \mathcal{T} | val(t_j)(k) = true\}$.

Construction: We prove the theorem by showing how to recursively construct the R2AA $A(e)$ for any expression e.
Let e_s be the expression corresponding to a stream s. Then we can transform a well-formed LOLA expression to a progressing R2AA in the following way:

- $A(e_1 \wedge e_2) = A(e_1) \wedge A(e_2)$
- $A(e_1 \vee e_2) = A(e_1) \vee A(e_2)$
- $A(\neg e) = A(\overline{e})$, if e is not a constant, output stream variable or input stream variable
- $A(\neg s) = A(\overline{e_s})$, where s is an output stream variable
- $A(s) = A(e_s)$, where s is an output stream variable
- $A(t) = \langle t, \varepsilon_{\mathcal{A}}, acc, \downarrow \rangle$, where t
- $A(\neg t) = \langle \neg t, \varepsilon_{\mathcal{A}}, acc, \downarrow \rangle$, where t is an input stream variable
- $A(c) = \langle c, \varepsilon_{\mathcal{A}}, acc, \downarrow \rangle$, where c is a constant
- $A(e[1, true]) = \langle true, A(e), acc, \rightarrow \rangle$
- $A(e[1, false]) = \langle true, A(e), rej, \rightarrow \rangle$
- $A(e[-1, true]) = \langle true, A(e), acc, \leftarrow \rangle$

- $A(e[-1, false]) = \langle true, A(e), rej, \leftarrow \rangle$

Proof. The property that the automaton is progressing follows directly from the fact that the specification is well-formed. We show the equivalence of e and $A(e)$ by induction over the structure of e.

- *Base Cases:*
  - Let $e = t_j$, where t_j is an input stream variable. Then $A(e)$ is the alternating automaton $\langle t_j, \varepsilon_{\mathcal{A}}, acc, \downarrow \rangle$. Obviously this atomic node accepts w at time i, if and only if $t_j \in w_i$. This is per definition the case exactly when $val(t_j)(i) = true$.
  - Let $e = \neg t_j$, where t_j is an input stream variable. Then $A(e)$ is the alternating automaton $\langle \neg t_j, \varepsilon_{\mathcal{A}}, acc, \downarrow \rangle$. Obviously this atomic node accepts w at time i, if and only if $t_j \notin w_i$. This is per definition the case exactly when $val(t_j)(i) = false \Leftrightarrow val(\neg t_j)(i) = true$.
  - Let $e = true$. Then $A(e)$ is the alternating automaton $\langle true, \varepsilon_{\mathcal{A}}, acc, \downarrow \rangle$. Then $A(e)$ accepts w at time i, if and only if $w_i \models true$. Since this is always the case, $A(e)$ accepts every input. In other words, $A(e)$ accepts w at time i, if and only if $true = true \Leftrightarrow val(true)(i) = true$.
  - Let $e = false$. Then $A(e)$ is the alternating automaton $\langle false, \varepsilon_{\mathcal{A}}, acc, \downarrow \rangle$. Then $A(e)$ accepts w at time i, if and only if $w_i \models false$. Since this is never the case, $A(e)$ accepts no input. In other words, $A(e)$ accepts w at time i, if and only if $false = true \Leftrightarrow val(false)(i) = true$.
- *Induction Step:*
  - Let $e = e_1 \wedge e_2$. Then $A(e)$ is the alternating automaton $A(e_1) \wedge A(e_2)$. Thus $A(e)$ accepts the word w at time i, if and only if both $A(e_1)$ and $A(e_2)$ accept w at time i. By induction it follows that this is the case, iff $val(e_1)(i) = true \wedge val(e_2)(i) = true \Leftrightarrow val(e_1 \wedge e_2)(i) = true$.
  - Let $e = e_1 \vee e_2$. Then $A(e)$ is the alternating automaton $A(e_1) \vee A(e_2)$. Thus $A(e)$ accepts the word w at time i, if and only if either $A(e_1)$ or $A(e_2)$ accept w at time i. By induction it follows that this is the case, iff $val(e_1)(i) = true \vee val(e_2)(i) = true \Leftrightarrow val(e_1 \vee e_2)(i) = true$.
  - Let $e = \neg e'$. Then $A(e)$ is the alternating automaton $A(\overline{e'})$. By induction it holds that $A(\overline{e'})$ accepts w at time i, if and only if $val(\overline{e'})(i) = true$. From Theorem 3.2 it follows that this is equivalent to $val(\neg e')(i) = true$.
  - Let $e = s$, where s is an output stream variable. Then $A(e)$ is the alternating automaton $A(e_s)$, where e_s is the stream expression corresponding to output stream s. By induction it follows that $A(e_s)$ accepts w at time i, if and only if $val(e_s)(i) = true \Leftrightarrow val(s)(i) = true$.
  - Let $e = \neg s$, where s is an output stream variable. Then $A(e)$ is the alternating automaton $A(\overline{e_s})$, where e_s is the stream expression corresponding to output stream s. By induction it holds that $A(\overline{e_s})$ accepts w at time i, if and only if $val(\overline{e_s})(i) = true$. From Theorem 3.2 it follows that this is equivalent to $val(\neg e_s)(i) = true \Leftrightarrow val(\neg s)(i) = true$.
  - Let $e = e'[1, true]$. Then $A(e)$ is the alternating automaton $\langle true, A(e'), acc, \rightarrow \rangle$. We proof the theorem by backwards induction over $\{0, ..., N-1\}$.
    - *Base case:* Let $i = N - 1$. Since $i = |w| - 1$, $\langle true, A(e'), acc, \rightarrow \rangle$ accepts w at time $N-1$, if and only if $w_{N-1} \models true$. In other words, $A(e)$ accepts w at time $N-1$, if and only if $true = true \Leftrightarrow val(e'[1, true])(N-1) = true$.

  - *Induction step:* Let $0 \leq i < N-1$. Since $i < |w|-1$, $\langle true, A(e), acc, \rightarrow \rangle$ accepts w at time i, if and only if $w_i \models true$ and $A(e')$ accepts w at time $i+1$. The first condition is trivially true, the second condition is by induction equivalent to $val(e')(i+1) = true \Leftrightarrow val(e'[1, true])(i) = true$.

- Let $e = e'[1, false]$. Then $A(e)$ is the alternating automaton $\langle true, A(e'), rej, \rightarrow \rangle$. We proof the theorem by backwards induction over $\{0, ..., N-1\}$.
  - *Base case:* Let $i = N-1$. Since the node is rejecting and $i = |w|-1$, no input will be accepted. In other words $A(e)$ accepts w at time $N-1$, if and only if $false = true \Leftrightarrow val(e'[1, false])(N-1) = true$.
  - *Induction step:* Let $0 \leq i < N-1$. Then $i < |w|-1$. Thus $\langle true, A(e), rej, \rightarrow \rangle$ accepts w at time i, if and only if $w_i \models true$ and $A(e')$ accepts w at time $i+1$. The first condition is trivially true, the second condition is by induction equivalent to $val(e')(i+1) = true \Leftrightarrow val(e'[1, false])(i) = true$.
- Let $e = e'[-1, true]$. Then $A(e)$ is the alternating automaton $\langle true, A(e'), acc, \leftarrow \rangle$. We proof the theorem by induction over $\{0, ..., N-1\}$.
  - *Base case:* Let $i = 0$. Then $\langle true, A(e'), acc, \leftarrow \rangle$ accepts w at time i, if and only if $w_i \models true$. Since this is always the case, $A(e)$ accepts every input at time 0. In other words, $A(e)$ accepts w at time i, if and only if $true = true \Leftrightarrow val(e'[-1, true])(0) = true$.
  - *Induction Step:* Let $0 < i < N$. Then $\langle true, A(e'), acc, \leftarrow \rangle$ accepts w at time i, if and only if $w_i \models true$ and $A(e')$ accepts w at time $i-1$. The first condition is trivially true, the second condition is by induction equivalent to $val(e')(i-1) = true \Leftrightarrow val(e'[-1, true])(i) = true$
- Let $e = e'[-1, false]$. Then $A(e)$ is the alternating automaton $\langle true, A(e'), rej, \leftarrow \rangle$. We proof the theorem by induction over $\{0, ..., N-1\}$.
  - *Base case:* Let $i = 0$. Since $\langle true, A(e'), rej, \leftarrow \rangle$ is rejecting, no input is accepted by $A(e)$ at time 0. In other words, $A(e)$ accepts w at time i, if and only if $false = true \Leftrightarrow val(e'[-1, false])(0) = true$.
  - *Induction Step:* Let $0 < i < N$. Then $\langle true, A(e'), rej, \leftarrow \rangle$ accepts w at time i, if and only if $w_i \models true$ and $A(e')$ accepts w at time $i+1$. The first condition is trivially true, the second condition is by induction equivalent to $val(e')(i-1) = true \Leftrightarrow val(e'[-1, true])(i) = true$

□

3.3.5 Trigger to R2AA

As it is now shown that every boolean LOLA expression has an equivalent R2AA, we can use this knowledge to build an R2AA for actual triggers. This construction was already described in the beginning of the section and we will now simply formalize it. A trigger is only activated when the associated expression becomes true at any time point, so the automaton needs to verify the complement. Therefore it has to check that the negated trigger expression is always true. We can encode the 'always' by adding a conjunction and a *true*-future-node to the automaton.

Theorem 3.4. *Given a strictly boolean LOLA specification with input stream variables $\mathcal{T} = \{t_1, ..., t_m\}$ and a trigger t associated with an expression e_t, there is progressing R2AA $A(t)$ so that the following holds:*
t will be activated for an input of length N, if and only if $A(t)$ rejects the word $w = w_0...w_{N-1}$ at time 0, where $w_i = \{t_j \in \mathcal{T} | val(t_j)(i) = true\}$.

Construction: The R2AA $A(t)$ can be constructed like this:

$$A(t) = \langle true, A(t), acc, \rightarrow \rangle \wedge A(\neg e_t),$$

where $A(\neg e_t)$ is the progressing R2AA that accepts an input at time i, if and only if $val(\neg e_t)(i) = true$. It follows directly that $A(t)$ is also progressing.

Proof. The trigger t will be activated if $val(e_t)(i) = true$ for any $i \in \{0, ..., N-1\}$. Thus is suffices to show that $val(\neg e_t)(j) = true$ for all $j \geq i$, if and only if $A(t) = \langle true, A(t), acc, \rightarrow \rangle \wedge A(\neg e_t)$ accepts w at time i. The case where we look at the whole input is then simply a special case, where $i = 0$. We can show this by backwards induction of i over $\{0, ..., N-1\}$, i.e. the length of the input streams.

- *Base case:* Let $i = N - 1$. Then $A(t)$ accepts w at time i, if and only if $A(\neg e_t)$ accepts w at time i. With theorem 3.3 it follows that this is equivalent to $val(\neg e_t)(i) = true$. Since there is no j with $i < j < N$, the claim is proven.
- *Induction step:* Let $i < N - 1$. Then the R2AA $A(t)$ accepts w at time i, if and only if $\langle true, A(t), acc, \rightarrow \rangle$ accepts w at time $i+1$ and $A(\neg e_t)$ accepts w at time i. The second condition is by theorem 3.3 the same as $val(\neg e_t)(i) = true$. The first condition is by induction the same as $val(\neg e_t)(j) = true \ \forall j, i < j < N$. Thus the claim is proven.

□

3.4 R2AA to DFA

In the last section it was shown that it is possible to build an equivalent R2AA for every trigger in a boolean LOLA specification, so that the semantics of the LOLA specification are encoded completely in an automata-theoretic representation. Therefore this section will not deal with LOLA at all, but instead with automata over sets of atomic propositions. More specifically, we will show how to transform a progressing R2AA into an equivalent DFA. This transformation will be presented in multiple steps. First we will translate the R2AA into an equivalent 2AA. Then we will use the algorithm from [8] to transform that 2AA into an equivalent NFA. Finally we will use the known methods to build an equivalent DFA.

3.4.1 R2AA to 2AA

We will first construct an equivalent 2AA for each R2AA. Therefore we define the set of states as the non-atomic nodes and add an initial state that represents the root of the 2AA. We then simply translate the acceptance conditions. In order to define the transition function Δ, we define a help function $\overline{\Delta}$ which maps a node and a direction to the states that can be reached in that direction.

Theorem 3.5. *For every progressing R2AA $\mathcal{A}$ over 2^{AP}, there is an equivalent progressing 2AA $\mathcal{A}'$.*

Construction: Let the 2AA $\mathcal{A}'$ be defined as $\mathcal{A}' = (Q, 2^{AP}, \Delta, I, F)$, where

- $Q = \{\langle v, \mathcal{A}_{succ}, f, g \rangle \in nodes(\mathcal{A}) | g \neq \downarrow\} \cup \{s_0\}$ is the set of states, given by the non-atomic nodes in $\mathcal{A}$ and a unique initial state s_0,
- Δ is the transition relation defined as
  - $\Delta(\langle v, \mathcal{A}_{succ}, f, g \rangle) = \overline{\Delta}(\mathcal{A}_{succ})$
  - $\Delta(s_0) = \overline{\Delta}(\mathcal{A})$

  where $\overline{\Delta}$ is defined as

  - $\overline{\Delta}(\varepsilon_{\mathcal{A}}) = \{(\emptyset, \Sigma, \emptyset)\}$
  - $\overline{\Delta}(\langle v, \mathcal{A}_{succ}, f, \downarrow\rangle) = \{(\emptyset, \{a \in \Sigma | a \models v\}, \emptyset)\}$
  - $\overline{\Delta}(\langle v, \mathcal{A}_{succ}, f, \leftarrow\rangle) = \{(\{\langle v, \mathcal{A}_{succ}, f, g\rangle\}, \{a \in \Sigma | a \models v\}, \emptyset)\}$
  - $\overline{\Delta}(\langle v, \mathcal{A}_{succ}, f, \rightarrow\rangle) = \{(\emptyset, \{a \in \Sigma | a \models v\}, \{\langle v, \mathcal{A}_{succ}, f, g\rangle\})\}$
  - $\overline{\Delta}(\mathcal{A}_1 \vee \mathcal{A}_2) = \overline{\Delta}(\mathcal{A}_1) \cup \overline{\Delta}(\mathcal{A}_2)$
  - $\overline{\Delta}(\mathcal{A}_1 \wedge \mathcal{A}_2) = \{(\overleftarrow{X}_1 \cup \overleftarrow{X}_2, \alpha_1 \cap \alpha_2, \overrightarrow{X}_1 \cup \overrightarrow{X}_2) | (\overleftarrow{X}_1, \alpha_1, \overrightarrow{X}_1) \in \overline{\Delta}(\mathcal{A}_1)$ and $(\overleftarrow{X}_2, \alpha_1, \overrightarrow{X}_2) \in \overline{\Delta}(\mathcal{A}_2)\}$,

- $I = \{\{s_0\}\}$ contains only the unique initial state s_0, and
- $F = \{\langle v, \mathcal{A}_{succ}, f, g\rangle \in Q \rangle | f = acc\}$ is the set of accepting nodes

Proof. We now need to proof that $\mathcal{A}$ and $\mathcal{A}'$ accept the same language. This is not very easy, therefore we will first prove a weaker claim, which says that for all i with $0 \leq i < |w|$ a state $\langle v, \mathcal{A}_{succ}, f, g\rangle$ in the 2AA $\mathcal{A}'$ accepts an input $w = w_0...w_{n-1}$ at time i, if and only if the R2AA $\mathcal{A}_{succ}$ accepts w at time i. We can proof this by induction over the structure of $\mathcal{A}_{succ}$.

- *Base Cases:*
  - Consider a state $q = \langle v, \varepsilon_{\mathcal{A}}, f, g\rangle$ in $\mathcal{A}'$. The R2AA $\varepsilon_{\mathcal{A}}$ accepts every input at any time with the empty run ε_T. It holds that $\Delta(q) = \overline{\Delta}(\mathcal{A}_{succ}) = \{(\emptyset, \Sigma, \emptyset)\}$. Thus q does also accept every input at any time i with $0 \leq i < |w|$.
  - Consider a state $q = \langle v, \mathcal{A}_{succ}, f, g\rangle$ with $\mathcal{A}_{succ} = \langle v', \mathcal{A}'_{succ}, f, \downarrow\rangle$. The R2AA $\mathcal{A}_{succ}$ accepts w at time i, if and only if $w_i \models v'$. It holds that $\Delta(q) = \overline{\Delta}(\mathcal{A}_{succ}) = \{(\emptyset, \{a \in \Sigma | a \models v'\}, \emptyset)\}$. Thus q does also accept every w at time i, if $w_i \models v'$ and $0 \leq i < |w|$.
- *Induction Step:*
  - Consider a state $q = \langle v, \mathcal{A}_{succ}, f, g\rangle$ with $\mathcal{A}_{succ} = \langle v', \mathcal{A}'_{succ}, f', \rightarrow\rangle$. We proof the claim by backward induction of i over $\mathbb{N}$.
    - *Base Case:* Let $i = |w| - 1$. Then the node $\mathcal{A}_{succ}$ accepts w at time i, if and only if $w_i \models v'$ and $f' = acc$. It holds that $\Delta(q) = \overline{\Delta}(\mathcal{A}_{succ}) = \{(\emptyset, \{a \in \Sigma | a \models v'\}, \mathcal{A}_{succ})\}$. Thus q can reach the state corresponding to $\mathcal{A}_{succ}$ by moving the position on the input forward by 1, if $w_i \models v'$. This (partial) run can only be accepting, if the state corresponding to $\mathcal{A}_{succ}$ is accepting, which is equivalent to $f' = acc$. Thus the claim holds.
    - *Induction Step:* Let $i < |w| - 1$. Then the node $\mathcal{A}_{succ}$ accepts w at time i, if and only if $w_i \models v'$ and $\mathcal{A}'_{succ}$ accepts w at time $i + 1$. It holds that $\Delta(q) = \overline{\Delta}(\mathcal{A}_{succ}) = \{(\emptyset, \{a \in \Sigma | a \models v'\}, \mathcal{A}_{succ})\}$. Thus q can reach the state corresponding to $\mathcal{A}_{succ}$ by moving the position on the input forward by 1, if $w_i \models v'$. This (partial) run is accepting, if the state corresponding to $\mathcal{A}_{succ}$ accepts w at time $i+1$. By induction, this is the case exactly when the R2AA $\mathcal{A}'_{succ}$ accepts w at time $i + 1$. Thus the claim holds.
  - Consider a state $q = \langle v, \mathcal{A}_{succ}, f, g\rangle$ with $\mathcal{A}_{succ} = \langle v', \mathcal{A}'_{succ}, f', \leftarrow\rangle$. We proof the claim by induction of i over $\mathbb{N}$.
    - *Base Case:* Let $i = 0$. Then the node $\mathcal{A}_{succ}$ accepts w at time i, if and only if $w_i \models v'$ and $f' = acc$. It holds that $\Delta(q) = \overline{\Delta}(\mathcal{A}_{succ}) = \{(\mathcal{A}_{succ}, \{a \in \Sigma | a \models v'\}, \emptyset)\}$. Thus q can reach the state corresponding to $\mathcal{A}_{succ}$ by moving the position on the input backward by 1, if $w_i \models v'$. This (partial) run can only be accepting, if the state corresponding to $\mathcal{A}_{succ}$ is accepting, which is equivalent to $f' = acc$. Thus the claim holds.

  - *Induction Step:* Let $i > 0$. Then the node $\mathcal{A}_{succ}$ accepts w at time i, if and only if $w_i \models v'$ and $\mathcal{A}'_{succ}$ accepts w at time $i-1$. It holds that $\Delta(q) = \overline{\Delta}(\mathcal{A}_{succ}) = \{(\mathcal{A}_{succ}, \{a \in \Sigma | a \models v'\}, \emptyset)\}$. Thus q can reach the state corresponding to $\mathcal{A}_{succ}$ by moving the position on the input backward by 1, if $w_i \models v'$. This (partial) run is accepting, if the state corresponding to $\mathcal{A}_{succ}$ accepts w at time $i-1$. By induction, this is the case exactly when the R2AA $\mathcal{A}'_{succ}$ accepts w at time $i-1$. Thus the claim holds.

- Consider a state $q = \langle v, \mathcal{A}_{succ}, f, g\rangle$ with $\mathcal{A}_{succ} = \mathcal{A}_1 \vee \mathcal{A}_2$. Then the R2AA $\mathcal{A}_{succ}$ accepts w at time i, if and only if $\mathcal{A}_1$ or $\mathcal{A}_2$ accepts w at time i. Now consider the hypothetical states $q_1 = \langle v, \mathcal{A}_1, f, g\rangle$ and $q_2 = \langle v, \mathcal{A}_2, f, g\rangle$ in the 2AA $\mathcal{A}'$. Each of those states only differs to q in its outgoing transitions. However, the set of outgoing transitions from q is simply the union of the sets of outgoing transitions from q_1 and q_2. Each transition that can be chosen in q can also be chosen in either q_1 or q_2. Conversely, if q_1 or q_2 take a transition, this transition can also be taken in q. Per induction the state q_j with $j \in \{1, 2\}$ accepts w a time i, if and only if the R2AA $\mathcal{A}_j$ accepts w at time i. Thus q accepts w at time i, if and only if $\mathcal{A}_{succ}$ accepts w at time i, and the claim holds.
- Consider a state $q = \langle v, \mathcal{A}_{succ}, f, g\rangle$ with $\mathcal{A}_{succ} = \mathcal{A}_1 \wedge \mathcal{A}_2$. Then the R2AA $\mathcal{A}_{succ}$ accepts w at time i, if and only if both $\mathcal{A}_1$ and $\mathcal{A}_2$ accept w at time i. Now consider the hypothetical states $q_1 = \langle v, \mathcal{A}_1, f, g\rangle$ and $q_2 = \langle v, \mathcal{A}_2, f, g\rangle$ in the 2AA $\mathcal{A}'$. Each of those states only differs to q in its outgoing transitions. It holds that $\Delta(q) = \overline{\Delta}(\mathcal{A}_{succ}) = \{(\overleftarrow{X}_1 \cup \overleftarrow{X}_2, \alpha_1 \cap \alpha_2, \overrightarrow{X}_1 \cup \overrightarrow{X}_2) | (\overleftarrow{X}_1, \alpha_1, \overrightarrow{X}_1) \in \overline{\Delta}(\mathcal{A}_1)$ and $(\overleftarrow{X}_2, \alpha_1, \overrightarrow{X}_2) \in \overline{\Delta}(\mathcal{A}_2)\} = \{(\overleftarrow{X}_1 \cup \overleftarrow{X}_2, \alpha_1 \cap \alpha_2, \overrightarrow{X}_1 \cup \overrightarrow{X}_2) | (\overleftarrow{X}_1, \alpha_1, \overrightarrow{X}_1) \in \Delta(q_1)$ and $(\overleftarrow{X}_2, \alpha_1, \overrightarrow{X}_2) \in \Delta(q_2)$. Assume a transition $(\overleftarrow{X_1} \cup \overleftarrow{X_2}, \alpha_1 \cap \alpha_2, \overrightarrow{X_1} \cup \overrightarrow{X_2})$ is taken from state q. Then the (partial) run is accepting, if all states in $\overleftarrow{X_1} \cup \overleftarrow{X_2}$ accept w at time $i-1$, all states in $\overrightarrow{X_1} \cup \overrightarrow{X_2}$ accept w at time $i+1$, and $w_i \in \alpha_1 \cap \alpha_2$. The state q_j with $j \in \{1, 2\}$ can take the transition $(\overleftarrow{X_j}, \alpha_j, \overrightarrow{X_j})$, which builds an accepting partial run, if all states in $\overleftarrow{X_j}$ accept w at time $i-1$, all states in $\overrightarrow{X_j}$ accept w at time $i+1$, and $w_i \in \alpha_j$. Thus q accepts w at time i, if and only if both q_1 and q_2 would accept w at time i. Per induction this is the case, if both $\mathcal{A}_1$ and $\mathcal{A}_2$ accept w at time i, which is equivalent to $\mathcal{A}_succ$ accepting w at time i.

We now proved that a state $q = \langle v, \mathcal{A}_{succ}, f, g\rangle$ accepts an input w at time i, if and only if the R2AA $\mathcal{A}_{succ}$ accepts w at time i. We can use this to prove that $\mathcal{A}$ and $\mathcal{A}'$ are equivalent. Therefore consider the hypothetical state $q = \langle v, \mathcal{A}, rej, g\rangle$ in $\mathcal{A}'$ (here v and g are irrelevant). This state does only accept w at time 0, if $\mathcal{A}$ accepts w at time 0 (this follows from the previous claim). It holds that $\Delta(s_0) = \overline{\Delta}(\mathcal{A}) = \Delta(q)$. Also, both q and s_0 are non-accepting states. Thus it follows that the state s_0 accepts w at time 0, if and only if the R2AA $\mathcal{A}$ accepts w at time 0, which means that $\mathcal{A}$ and $\mathcal{A}'$ accepts the same set of inputs.
The fact that $\mathcal{A}'$ is progressing follows directly from the fact that $\mathcal{A}$ is progressing. □

3.4.2 2AA to NFA

We can now apply the work from [8] to remove the alternation and bidirectionality from the 2AA, in order to arrive at an equivalent NFA. This is done by defining transitions between pairs of state sets, so that each transitions is associated with a sequence of three state sets.

Theorem 3.6 ([8])**.** *For every progressing 2AA over finite words, there is an equivalent NFA.*

Construction: For a progressing 2AA $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$, let the NFA $N_{\mathcal{A}}$ be defined as $N_{\mathcal{A}} = (Q', \Sigma, \Delta', I', F')$, where

- $Q' = 2^Q \times 2^Q$ is the set of states given by a set over tuples of subsets of Q,
- Σ is the same,
- $\Delta'((\overleftarrow{X}, X), \alpha) = \{(X, \overrightarrow{X}) | \exists (\overleftarrow{Y}, \alpha, \overrightarrow{Y}) \in \bigotimes_{q \in X} \Delta(q)$ with $\overleftarrow{Y} \subseteq \overleftarrow{X}$ and $\overrightarrow{Y} \subseteq \overrightarrow{X}\}$ is the transition function, where $A \otimes B = \{(\overleftarrow{X}_1 \cup \overleftarrow{X}_2, \alpha_1 \cap \alpha_2, \overrightarrow{X}_1 \cup \overrightarrow{X}_2) | (\overleftarrow{X}_1, \alpha_1, \overrightarrow{X}_1) \in A$ and $(\overleftarrow{X}_2, \alpha_1, \overrightarrow{X}_2) \in B\}$,
- $I' = \{(X, Y) \in Q' \times Q' | X \subseteq F$ and $\exists Z \in I$ with $Z \subseteq Y\}$ is the initial state, and
- $F' = \{(X, Y) \in Q' \times Q' | Y \subseteq F\}$

The equivalence of this construction is proven in [8] for very-weak 2AAs over finite and infinite words. An automaton is called very-weak, if there is a partial order over the state set, so that each state can only reach states that are smaller with respect to this order. This property is needed to show that infinite runs converge. However, since we use progressing automata over finite words here, this property is not needed and the here shown construction is a special case of the construction in [8].

3.4.3 NFA to DFA

The equivalence of DFAs and NFAs is well-known, therefore we simply use the classic subset construction.

Theorem 3.7. *For every NFA $N = (Q, \Sigma, \Delta, s_0, F)$, there is a DFA $A_N = (Q', \Sigma, \delta, s_0', F')$ so that N and A_N accept the same language.*

Construction: Given a NFA $N = (Q, \Sigma, \Delta, s_0, F)$, we can create an equivalent DFA A_N in the following way:
$A_N = (Q', \Sigma, \delta, s_0', F')$, where

- $Q' = 2^Q$ is the powerset of Q,
- Σ is the same,
- δ is defined as $\delta(S, \alpha) = \bigcup_{s \in S} \Delta(s, \alpha)$ for all $S \in Q'$ and $\alpha \in \Sigma$,
- $s_0' = \{s_0\}$, and
- $F' = \{S \in Q' | S \cap F \neq \emptyset\}$

Starting from a 2AA with n states, the application of the previously shown construction leads to a DFA with $\mathcal{O}(2^{2^{2n}})$ states. It makes sense to minimize the resulting DFA as much as possible so that the LOLA specification, which is generated later, is as small as possible. We will show the method used for this in chapter 5.

3.5 DFA to LOLA

Finally we will show how to synthesize an efficiently monitorable LOLA specification that is equivalent to a given DFA. Such a specification can be built by creating a LOLA stream variable for each state in the DFA. A stream associated with a state s is defined so that it only becomes true if there is a transition $s' \xrightarrow{\alpha} s$ such that the stream associated with s' was true at the previous execution step and exactly the set of input streams given by α is

true at the current execution step. When referencing other streams at a previous execution step, only references to the stream associated with the initial state have the default value *true*. The trigger simply checks that no stream that is not associated with an accepting state evaluates to true in the final execution step. If the DFA has bad states (i.e. states that lead to every input being rejected once they are reached), then we can additionally check in the trigger that a stream associated with such a state never becomes true.

Theorem 3.8. *For every DFA $A = (Q, \Sigma, \delta, q_0, F)$ where $\Sigma = 2^{AP}$ for a set AP of atomic propositions, there is an efficiently monitorable LOLA specification, which is equivalent if the atomic propositions are interpreted as input stream variables.*

Construction: Given a DFA $A = (Q, \Sigma, \Delta, q_0, F)$, where $\Sigma = 2^{AP}$ for a set AP of boolean LOLA input stream variables, and let $\bot_A$ denote the set of bad states defined as $\bot_A = \{q \in Q | \mathcal{L}(q) = \emptyset\}$. Additionally, let the function $trans : Q \times Q \to 2^{\Sigma}$ be defined as $trans(q, q') = \{\alpha \in \Sigma | \delta(q, \alpha) = q'\}$. We can then create a LOLA specification in the following way:
We regard AP as the set of input stream variables and define the boolean output stream

$$\begin{aligned} s_q = & s_{q_0}[-1, true] \wedge \Big(\bigvee_{\alpha \in trans(q_0, q)} \Big(\bigwedge_{t \in \alpha} t \wedge \bigwedge_{t \in AP/\alpha} \neg t \Big) \Big) \\ & \vee \bigvee_{q' \in Q/s_0} \Big(s_{q'}[-1, false] \wedge \Big(\bigvee_{\alpha \in trans(q', q)} \Big(\bigwedge_{t \in \alpha} t \wedge \bigwedge_{t \in AP/\alpha} \neg t \Big) \Big) \Big) \end{aligned}$$

for each state $q \in Q$. The trigger is the defined as

$$\begin{aligned} & s_t = \bigvee_{q \in \bot_A} s_q \vee (ended \wedge \bigvee_{q \in Q/F} s_q) \\ & \textbf{trigger } s_t \end{aligned}$$

where the stream *ended* is defined as

$$ended = false[1, true]$$

Proof. The LOLA specification that is created here is obviously well-formed. It is also efficiently monitorable because the only positive offset appears in *ended*, which can always be resolved in the next execution step.
Now we need to prove that the here created LOLA specification is equivalent to the automaton. As earlier we do this by defining a word in each position as the set of input streams that evaluate to true at that execution step.
Before we can prove the actual equivalence, we first need to prove a weaker statement, which says that the output stream variables $\{s_q\}$ 'bisimulate' the runs in A. We do this by showing that each stream variable s_q evaluates to true, if and only if the corresponding state q is currently active in the run for the automaton.
More formally, we want to first show the following: Let $\mathcal{T} = \{t_1, ..., t_m\}$ be a set of boolean LOLA stream variables, N be the input stream length and $A = (Q, \Sigma, \Delta, q_0, F)$ be a DFA with $\Sigma = 2^{\mathcal{T}}$. Then A is in state q after reading the word $w = w_0 ... w_{n-1}$, where $0 \leq n < N$ and $w_i = \{t \in \mathcal{T} | val(t)(i) = true\}$, if and only if $val(s_q)(n-1) = true$ and $val(s_{q'})(n-1) = false$ for all $q' \in Q/\{q\}$. We prove this by induction over the length of w.

- *Base case:* Let $n = |w| = 1$. Assume that A is in state q after reading w. Then the corresponding run of w in A has the form $q_0 \xrightarrow{w_0} q$ and it follows that $w_0 \in trans(q_0, q)$. Thus

$$val\Big(\bigvee_{\alpha \in trans(q_0, q)} \Big(\bigwedge_{t \in \alpha} t \wedge \bigwedge_{t \in AP/\alpha} \neg t \Big) \Big)(0)$$

is $true$ and since $val(s_{q_0})(0) = true$, it follows that $val(s_q)(0) = true$. Since $val(s_q[-1, false])(0)$ always evaluates to false, the value of any other stream variable $s_{q'}$ can only become true when $val(\bigvee_{\alpha \in trans(q_0,q')} (\bigwedge_{t \in \alpha} t \wedge \bigwedge_{t \in AP/\alpha} \neg t))(0)$ is true. This only happens, if $w_0 \in trans(q_0, q')$. Since $w_0 \in trans(q_0, q)$ already holds, that would mean that A is non-deterministic, which is a contradiction. Thus, $val(s_{q'})(0) = false$ for every $q' \neq q$.

- *Induction step:* Let $n = |w| > 1$. Assume that A is in state q after reading w. Then the corresponding run of w in A has the form $q_0 \xrightarrow{w_0} ... \xrightarrow{w_{n-2}} q' \xrightarrow{w_{n-1}} q$. Thus A is in state q' after reading the word $w' = w_0...w_{n-2}$. By induction this means that $val(s_{q'})(n-2) = true$ and $val(s_{q''})(n-2) = false$ for all $q'' \neq q'$. Obviously it also holds that $w_{n-1} \in trans(q', q)$. Consequently

$$val(\bigvee_{\alpha \in trans(q',q)} (\bigwedge_{t \in \alpha} t \wedge \bigwedge_{t \in AP/\alpha} \neg t))(N)$$

evaluates to true. Since by induction $val(s_{q'}[-1, true])(n-1)$ also is true, it follows that $val(s_q)(n-1)$ evaluates to true. Since per induction $val(s_{q''}[-1, false])(n-1)$ is false for all $q'' \neq q'$, the value of any other stream variable $s_{q''}$ can only become true when $val(\bigvee_{\alpha \in trans(q',q'')} (\bigwedge_{t \in \alpha} t \wedge \bigwedge_{t \in AP/\alpha} \neg t))(n-1)$ is true. This only happens, if $w_{n-1} \in trans(q', q'')$. Since $w_{n-1} \in trans(q', q)$ already holds, that would mean that A is non-deterministic, which is a contradiction. Thus, $val(s_{q''})(n-1) = false$ for every $q'' \neq q$.

We know now that the created output streams behave in the same way as the automaton. The only thing left is to show that the set of stream valuations accepted by the trigger and the language of the automaton are the same. Obviously the trigger accepts an input stream valuation if s_t evaluates to $false$ in each execution step. Hence we will now show that s_t only evaluates to $true$ at any execution step, if the DFA accepts the input stream valuation.

- '$\Rightarrow$': Assume A accepts w. Then the run of w in A has the form $q_0 \xrightarrow{w_0} ... \xrightarrow{w_{N-1}} q_N$, where $q_N \in F$. Obviously for all i with $0 \leq i < N-1$, q_i is not in $\bot_A$, because a bad state can by definition not be part of an accepting run. Thus it follows directly that $val(s_t)(i) = false$ for all $0 \leq i < N$. With the previously shown claim it holds that $val(s_{q_N})(N-1) = true$ and $val(s_{q'})(N-1) = false$ for all $q' \neq q_N$. Since q_N is an accepting state, $val(\bigvee_{q \in Q/F} s_q)(N-1)$ evaluates to $false$ and thus $val(s_t)(N-1)$ is $false$.
- '$\Leftarrow$': Assume A does not accept w. Then the run of w in A has the form $q_0 \xrightarrow{w_0} ... \xrightarrow{w_{N-1}} q_N$, where $q_N \in Q/F$. With the previously shown claim it holds that $val(s_{q_N})(N-1) = true$. Thus $val(\bigvee_{q \in Q/F} s_q)(N-1)$ evaluates to $false$ and $val(s_t)(N-1)$ evaluates to $true$.

□

Note that the checking for bad states in the trigger is actually redundant if one just cares about acceptance. Especially when checking safety properties however, it makes the trigger activate earlier when the property is violated. If the trigger would not check for bad states, it would always be activated in the last execution step, even if the property was irrecoverably violated at an earlier execution step, which would be impractical.

It is also interesting to note that this theorem shows that efficiently monitorable LOLA specifications are at least as powerful as DFAs in regards to language acceptance. Together with the previous theorems in this chapter, this shows that boolean LOLA specifications can recognize exactly the regular languages.

3.6 Extensions

3.6.1 Dealing with Non-Boolean Streams

In the previous sections we showed how to transform strictly boolean LOLA specifications into equivalent efficiently monitorable LOLA specifications, but as soon as streams of other types appear, the earlier described methods become inapplicable. However, since this chapter deals with the verification of properties, which in LOLA is done by using triggers, the top-level expression is still of boolean type. Thus streams of other types can only appear as part of constraints, which are boolean expressions. We can use this fact to manipulate a specification so that the methods from the earlier sections are still applicable. This can be done by creating a new stream variable for each constraint subexpression that is encountered when following the dependency graph of the specification. In the algorithm, the newly created stream variables are then treated exactly like boolean input stream variables so that only a strictly boolean specification is left.
As an example consider the following LOLA specification:

$$\begin{aligned} s &= id \geq 0 \wedge s[1, true] \\ &\textbf{trigger } \neg s \end{aligned}$$

This is a simple specification, which checks that an integer input stream named id never has a value smaller than 0 (actually the future self reference here is redundant, but we keep it for the sake of a more complex example). We now use the method described above to remove the integer subexpression $id \geq 0$.

$$\begin{aligned} id\_constr &= id \geq 0 \\ s &= id\_constr \wedge s[1, true] \\ &\textbf{trigger } \neg s \end{aligned}$$

During the optimization algorithm we simply treat id_constr as a boolean input stream variable. We can then either add id_constr to the optimized specification or even substitute each reference to id_constr by the expression $id \geq 0$ in the optimized specification.
This method may work out for simple specifications like in the example, but it is not as efficient for more complex ones. Consider for example the LOLA specification given by

$$\begin{aligned} s_1 &= id + s_1[1, 0] \\ s_2 &= s_1 < 1000 \wedge s_2[1, false] \\ &\textbf{trigger } \neg s_2 \end{aligned}$$

As before we have an integer stream s_1 that appears inside a constraint in a boolean stream s_2. If we tried to create a new boolean stream for the constraint however, this new stream would still contain positive cycles that would not be removed during the optimization algorithm. Unfortunately specifications like this can in general not be made completely efficiently monitorable by our algorithms. This is because LOLA with integer streams is at least context-free (a stack can be emulated by an integer stream) [1], while we only use DFAs and other equally powerful automata here. It may be possible to capture the semantics of such specifications by using more powerful formalisms like pushdown automata, but we will not investigate that

further in this thesis. However we can still apply the earlier described method to make the specification 'more efficiently monitorable' by removing at least some positive offsets. If we would create a stream for the constraint in the example, and then optimize the specification, the positive cycle for s_2 would be removed, but the positive cycle in s_1 would remain. Thus the resulting specification would not be efficiently monitorable, but it would still have less worst-case memory requirements.
Note that this also means that not necessarily every boolean stream will be translated. If a boolean stream with a positive cycle is referenced by an integer stream, then the boolean stream would be considered to be independent from the rest of the specification and not be captured in the automaton. There is also the case that a boolean stream, which would usually be translated, is also referenced by an integer stream. In this case it will be encoded in the automaton, but the original stream will still be preserved in the optimized specification, so that the integer stream can reference it.

3.6.2 Adjusting the Monitoring Algorithm

The previously shown optimization algorithm uses a series of automata transformations. The complexity of those constructions leads to a double-exponential blow-up in the state set. Fortunately we can use minimization techniques to heavily reduce the state set size of the DFA. However, the constructed LOLA specification is often still a lot more complex than the original specification, which leads to increased time requirements for the overall monitoring procedure (see chapter 5).
It is possible to reduce these additional time requirements, by merging the monitoring algorithm with the optimization algorithm that was presented in this chapter. The basic idea is to give the DFA, which is generated by the optimization algorithm, directly to the monitoring algorithm (instead of the LOLA specification). Each step in the algorithm then only considers the relevant part of this DFA, i.e. the current state and all states that are reachable in one step. This removes unnecessary computations for non-relevant states, which happen if the optimization and monitoring algorithms are separated. However, it adds the need to build the DFA in the beginning of every monitoring task, which may use a lot of time and memory for complex LOLA specifications. Hence this approach may become unfavorable, if a specification is reused often for multiple monitoring tasks.

4 Optimizing Statistical Measures

4.1 Problem

In addition to the verification of properties, LOLA also offers the collection of incremental statistics. The former one can be regarded as a language acceptance problem and its acceptance conditions can be easily captured by automata. However, this is not as simple for statistics. Unfortunately LOLA does not yet offer a mechanism similar to the trigger-keyword for statistics, thus statistics can only be collected by marking certain streams to have their last value printed at the end of the execution. Since only their last value will be printed, it does not make a lot of sense to define those streams in a forward manner. Consider for example the forward-defined sum over an input stream t, given by the stream $s = t + s[1, 0]$. The value printed to the user would then only be the last value of s, i.e. the last value of t, while the actual sum over all values of t, would be stored in the first position of s. This could be prevented by defining a keyword which takes a position i and a stream s, and prints the value of s at position i to the user. While this is not yet possible in LOLA, in this chapter we will still take a look at forward-recursive incremental statistics and how to represent them non-recursively, so that we can easily translate them into backward-recursive streams.

4.2 Simple Aggregates

In this section we will take a look at simple streams of the form $s = e \circ s[1, c]$, where $\circ$ is a binary operator like $+$ or max, e is an expression that does not depend on s and c is a constant. An example is the sum over the input stream t defined as $s = t + s[1, 0]$. The value of s at time i is then obviously given by the recurrence equations

$$s_i = \begin{cases} t_i + s_{i+1} & \text{for } 0 \leq i < N \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

where N is the input stream length. In general, the task of translating an incremental statistics in LOLA can always be solved by bringing the corresponding recurrence into a closed form. If we unfold the above recurrence for $i = 0$, we get

$$s_0 = t_1 + (t_2 + ... + (t_{N-2} + (t_{N-1} + 0))...)$$

Because of the associativity of $+$ this is obviously the same as writing

$$s_0 = (...((0 + t_1) + t_2) + ... + t_{N-2}) + t_{N-1}$$

which is equal to the $(N - 1)$-th value of the unfolded version of

$$s'_i = \begin{cases} s'_{i-1} + t_i & \text{for } 0 \leq i < N \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

This is in turn the recurrence associated with the stream $s' = s'[-1, 0] + t$, which is an efficiently monitorable LOLA stream that has the same value at position $N - 1$ as s has at position 0. Thus we used the associativity of $+$ to make s efficient. We know want to generalize this idea. Therefore we first define generalizations of the forward recurrence in (4.1) and the backward recurrence in (4.2).

Definition 4.1. Let $\circ$ be an arbitrary binary operator for a type T, $e : \mathbb{N} \to T$ be a function and c be a constant of type T. Then

$$\overset{n}{\underset{i=k}{\uparrow}} \circ_c e(i) = \begin{cases} c & \text{, if } k > n \\ e(k) \circ (\overset{n}{\underset{i=k+1}{\uparrow}} \circ_c e(i)) & \text{, otherwise} \end{cases}$$

and

$$\overset{n}{\underset{i=k}{\downarrow}} \circ_c e(i) = \begin{cases} c & \text{, if } k > n \\ (\overset{n-1}{\underset{i=k}{\downarrow}} \circ_c e(i)) \circ e(n) & \text{, otherwise} \end{cases}$$

Now we can prove that these two definitions are the same for associative operators that are commutative for c. We have not talked about the c yet because it was the neutral element in our previous example, but depending on the direction of the recurrence it is added to the total sum from different sides.

Theorem 4.1. *For all $k, n \in \mathbb{N}_{\geq 0}$ the following holds: If $\circ$ is an associative operator, which is commutative for c, then $\overset{n}{\underset{i=k}{\uparrow}} \circ_c e(i) = \overset{n}{\underset{i=k}{\downarrow}} \circ_c e(i)$.*

Proof. At first we make a case distinction:

- $n < k$. Then obviously $\overset{n}{\underset{i=k}{\uparrow}} \circ_c e(i) = \overset{n}{\underset{i=k}{\downarrow}} \circ_c e(i) = c$.
- $n \geq k$. We proof the statement by induction over $n \in \mathbb{N}_{\geq 0}$.
  - *Base case:* Let $n = 0$. Since $n \geq k$ holds, it follows that $k = 0$. Then:

$$\begin{aligned} \overset{n}{\underset{i=k}{\uparrow}} \circ_c e(i) &= e(0) \circ c && \text{Definition of } \overset{n}{\underset{i=k}{\uparrow}} \circ_c e(i) \text{ for } n = k = 0 \\ &= c \circ e(0) && \text{Commutativity for } c \\ &= \overset{n}{\underset{i=k}{\downarrow}} \circ_c e(i) && \text{Definition of } \overset{n}{\underset{i=k}{\downarrow}} \circ_c e(i) \text{ for } n = k = 0 \end{aligned}$$

  - *Induction step:* Let $k \leq n$ and $0 < n$. We proof the statement by backwards induction over $k \in \{0, ..., n\}$:
    - *Base case:* Let $k = n$. Then

$$\begin{aligned} \overset{n}{\underset{i=k}{\downarrow}} \circ_c e(i) &= \overset{n-1}{\underset{i=n}{\downarrow}} \circ_c e(i) \circ e(n) && \text{Definition of } \overset{n}{\underset{i=k}{\downarrow}} \circ_c e(i) \\ &= c \circ e(n) && \text{Definition of } \overset{n}{\underset{i=k}{\downarrow}} \circ_c e(i) \text{ for } k > n \\ &= e(n) \circ c && \text{Commutativity for } c \\ &= e(n) \circ \overset{n}{\underset{i=n+1}{\uparrow}} \circ_c e(i) && \text{Definition of } \overset{n}{\underset{i=k}{\uparrow}} \circ_c e(i) \text{ for } k > n \\ &= \overset{n}{\underset{i=k}{\uparrow}} \circ_c e(i) && \text{Definition of } \overset{n}{\underset{i=k}{\uparrow}} \circ_c e(i) \end{aligned}$$

∗ *Induction step:* Let $k < n$. Then

$$
\begin{aligned}
\underset{i=k}{\overset{n}{\downarrow}} \circ_c e(i) &= \underset{i=k}{\overset{n-1}{\downarrow}} \circ_c e(i) \circ e(n) && \text{Definition of } \underset{i=k}{\overset{n}{\downarrow}} \circ_c e(i) \\
&= \underset{i=k}{\overset{n-1}{\uparrow}} \circ_c e(i) \circ e(n) && \text{Induction} \\
&= (e(k) \circ (\underset{i=k+1}{\overset{n-1}{\uparrow}} \circ_c e(i))) \circ e(n) && \text{Definition of } \underset{i=k}{\overset{n}{\uparrow}} \circ_c e(i) \\
&= e(k) \circ ((\underset{i=k+1}{\overset{n-1}{\uparrow}} \circ_c e(i)) \circ e(n)) && \text{Associativity of } \circ \\
&= e(k) \circ ((\underset{i=k+1}{\overset{n-1}{\downarrow}} \circ_c e(i)) \circ e(n)) && \text{Induction} \\
&= e(k) \circ (\underset{i=k+1}{\overset{n}{\downarrow}} \circ_c e(i)) && \text{Definition of } \underset{i=k}{\overset{n}{\downarrow}} \circ_c e(i) \\
&= e(k) \circ (\underset{i=k+1}{\overset{n}{\uparrow}} \circ_c e(i)) && \text{Induction} \\
&= \underset{i=k}{\overset{n}{\uparrow}} \circ_c e(i) && \text{Definition of } \underset{i=k}{\overset{n}{\uparrow}} \circ_c e(i)
\end{aligned}
$$

□

Since $\underset{i=k}{\overset{n}{\uparrow}} \circ_c e(i)$ and $\underset{i=k}{\overset{n}{\downarrow}} \circ_c e(i)$ only differ in their definition, we can introduce a new notation $\underset{i=k}{\overset{n}{\circ_c}} e(i)$, if $\circ$ is associative and commutative for c. We can now apply this concept on LOLA streams, as we did in the sum example.

Theorem 4.2. *Let s be a LOLA stream variable that is part of a well-formed specification. If the stream expression of s has the form $e \circ s[1, c]$ (respectively $s[1, c] \circ e$), where e does not depend on s, then the following holds: If $\circ$ is commutative for c and associative, then a new stream variable $s' = s'[-1, c] \circ e$ (respectively $s' = e \circ s'[-1, c]$) can be created, such that for input stream length N, $val(s')(N-1) = val(s)(0) = \underset{i=0}{\overset{N-1}{\circ_c}} val(e)(i)$.*

Proof. We can define new stream $\overline{s} = ite(true, e \circ s[1, c], c)$ and $\overline{s'} = ite(true, s[-1, c] \circ e, c)$ (respectively $\overline{s} = ite(true, s[1, c] \circ e, c)$ and $\overline{s'} = ite(true, e \circ s[-1, c], c)$), so that $val(s)(k) = val(\overline{s})(k)$ and $val(s')(k) = val(\overline{s'})(k)$ for all $0 \leq k < N$. Then we just need to proof the equivalence of $\overline{s}$ and $\overline{s'}$. This is a special case of Theorem 4.4, which we will proof later. □

Example 4.1. Assume we have an output stream variable $s = t + s[1, 0]$, where t is an input stream. Since $+$ is associative and commutative for 0 and s is not referenced in the left part, we can create a new stream $s' = s'[-1, 0] + t$, so that $val(s)(0) = val(s')(N-1)$ (where N is the length of the stream t). The underlying formula would be $\underset{i=0}{\overset{N-1}{+_0}} val(t)(i)$, which is the same as $\sum_{i=0}^{N-1} val(t)(i)$.

4.3 More Complex Aggregates

Now we will consider streams that are not on the form which was presented in the previous section. In some cases the streams can simply be transformed into such a form, for example $s = t + (t' + s[1, 0])$ is the same as $s = (t + t') + s[1, 0]$. In order to automate this, one has

to provide a set of transformations for each operator, which move the subexpression $s[1, c]$ upwards in the syntax tree.

4.3.1 Arbitrary Positive Offsets

There are also cases in which the stream can not be brought into such a form. An obvious example would be a stream s containing the subexpression $s[i, 0]$ with $i > 0$, like in the stream $s = t + s[2, 0]$. Note that this stream cannot be simply rewritten into $s' = s'[-2, 0] + t$, because those would not be equal for inputs with length N such that $(N - 1) \bmod 2 \neq 0$. However, there is another way:

Theorem 4.3. *Let s be a LOLA stream variable that is part of a well-formed specification. If the with s associated stream expression has the form $e \circ s[i, c]$ (respectively $s[i, c] \circ e$), where e does not depend on s, then the following holds: If $\circ$ is commutative for c and associative, then $i + 1$ new stream variables*

$$s_j = \begin{cases} s_0[-i, c] \circ e & \text{if } j = 0 \\ s_0[-j, c] & \text{if } 0 < j < i \end{cases}$$
$$count = 1 + count[-1, 0]$$

(respectively $s_0 = e \circ s_0[-i, c]$) can be created, so that we can (by using if-expressions) create a stream variable s' such that

$$s' = s_i \quad \text{if} \quad (count - 1) \bmod i = 0.$$

Then for an input of length N it holds that $val(s')(N - 1) = val(s)(0) = \overset{\lfloor N-1/i \rfloor}{\underset{j=0}{\circ_c}} val(e)(j \cdot i)$.

Proof. Let $k \in \mathbb{N}$ be the biggest number that satisfies $k * i < N$. Then obviously $val(s)(k * i) = val(e)(i * k) \circ c$ holds. Thus the remaining positions of the input are irrelevant to compute the value of S at time 0 and theorem 4.2 can be applied, which implies that $val(s)(0) = val(s_0)(k * i)$. By definition it holds that $val(s')(N - 1) = val(s_0)(k * i)$ and thus $val(s')(N - 1) = val(s)(0)$. □

4.3.2 Multiple Self References

Another possibility is the appearance of multiple self references in a single stream, both of which have a positive offset (one positive and one negative offset would not make sense because the stream would not be well-formed). Take for example the fibonacci-like stream $s = t + s[1, 0] + s[2, 0]$. In the closed form of the associated recurrence equations the values of t at later positions would have a much bigger weight than the values of t at earlier positions. This is the same for streams with multiple self references that have all the same offset, e.g. $s = t + 2 \cdot s[1, 0] = t + s[1, 0] + s[1, 0]$. For these streams, solving the recurrence equations becomes much harder since the recursion tree is non-linear. In the closed form each reference to another stream has to be weighted with a certain value which depends on the referenced position. Automating this process of finding a closed form for the recurrence equations can only be done with sophisticated mathematical methods as in [2], which are outside of the scope of this thesis.

4.3.3 Indirect Self References

It can also happen, that a stream references itself indirectly, i.e. by referencing another stream that references it, so that there would be a cycle of references. This becomes problematic

when any stream has multiple of such cycles. An example for this would be the LOLA streams

$$s = s'[1,0] + t$$
$$s' = s'[1,0] + s[1,0] + t$$

Here, s at time i would depend on twice the value of s at time $i+2$ and we could apply the same argumentation as in the previous paragraph. If however, there is only a single cycle of references, the sum of their offsets is positive and they all have the same default value, then our streams would look like this:

$$s_1 = s_2[i_1, c] \circ e_1$$
$$s_2 = s_3[i_2, c] \circ e_2$$
$$\dots$$
$$s_n = s_1[i_n, c] \circ e_n$$

If $\circ$ is associative, then we can rewrite any of those streams to a new stream

$$\begin{aligned} s'_i =& e_{i+1}[i_i, c] \circ e_{i+2}[i_i + i_{i+1}, c] \circ \dots \circ e_n[i_i + \dots + i_n, c] \\ & \circ e_1[i_i + \dots + i_n + i_1, c] \circ \dots \circ e_{i-1}[i_i + \dots + i_n + i_1 + \dots + i_{i-1}] \circ s'_i[i_1 + \dots + i_n, c] \end{aligned}$$

Since no e_i depends on s, we can rewrite the stream as:

$$s'_i = e'_i \circ s'_i[i', c]$$

where $i' > 0$ and we can then translate this stream with our previous results.

4.3.4 Expressions with Offsets

As another scenario, consider a stream s that references an expression, which contains a reference to s, at an offset. An example for this would be the stream

$$s = (t + s[1,0])[1,0]$$

We can then simply rewrite the referenced expression as a new stream and reference this stream instead of the expression. Applied on the example, it would look like this:

$$s = s'[1,0]$$
$$s' = t + s[1,0]$$

We would have a cycle of references and can now follow the same approach as in the previous paragraph.

4.3.5 Nested Incremental Statistics

Finally, consider a stream that references other incremental statistics, for example a stream that computes the sum over partial sums. Unfortunately it is not possible to simply translate the referenced streams and just use the translated ones instead, because these are in general not equal to the original streams if they are referenced at the same time. To solve this problem, we just treat those streams as if they are input streams and do not translate them at all. As a consequence, there may be some streams that can not be made efficiently monitorable with our methods, but that can be made 'more efficiently monitorable' by removing at least some of the future offsets.

4.4 Dealing with If-Expressions

In the last section we presented a few methods to optimize some more complex aggregates. If-expressions are a little harder to translate because they have no concrete mathematical counterpart that we can use to capture their semantics. At first we will think about the case, where the root of the syntax tree is an if-expression, the condition does not contain a self reference and both alternative and consequence have the same offset for their self reference (if there is one). In the following we will give a translation method for if-expressions in a certain form, which is an extension of the one shown in section 4.2.

Theorem 4.4. *Let s be a LOLA stream variable that is part of a well-formed specification. If the stream expression of s has the form*

$$s = ite(e_1, e_2 \circ s[1, c], e_3 \circ s[1, c]),$$

respectively

$$s = ite(e_1, s[1, c] \circ e_2, s[1, c] \circ e_3),$$

where e_1, e_2, e_3 do not depend on s, then the following holds: If $\circ$ is commutative for c and associative, then a new stream variable

$$s' = ite(e_1, s[-1, c] \circ e_2, s[-1, c] \circ e_3),$$

respectively

$$s' = ite(e_1, e_2 \circ s'[-1, c], e_3 \circ' s[-1, c]),$$

can be created, such that for an input of length N, $val(s')(N-1) = val(s)(0) = \overset{N-1}{\underset{i=0}{\circ_c}} val(ite(e_1, e_2, e_3))(i)$.

Proof. Assume w.l.o.g that $s = ite(e_1, e_2 \circ s[1, c], e_3 \circ s[1, c])$. First we prove by backwards induction over $k \in \{0, ..., N-1\}$ that
$val(s)(k) = \overset{N-1}{\underset{i=k}{\uparrow \circ_c}} val(ite(e_1, e_2, e_3))(i)$:

- *Base case:* Let $k = N - 1$. Then

$$\begin{aligned}
val(s)(k) &= \text{if } val(e_1)(k) \text{ then } val(e_2 \circ s[1, c])(k) \\
&\quad \text{else } val(e_3 \circ s[1, c])(k) && \text{Definition of } val \\
&= \text{if } val(e_1)(k) \text{ then } val(e_2)(k) \circ c \\
&\quad \text{else } val(e_3)(k) \circ c && \text{Definition of } val \text{ for } k + 1 > N - 1 \\
&= val(ite(e_1, e_2, e_3))(k) \circ c && \text{Definition of } val \\
&= val(ite(e_1, e_2, e_3))(k) \\
&\quad \circ \left(\overset{N-1}{\underset{i=k+1}{\uparrow \circ_c}} val(ite(e_1, e_2, e_3))(i) \right) && \text{Definition of } \overset{n}{\underset{i=k}{\uparrow \circ_c}} \text{ for } k > n \\
&= \overset{N-1}{\underset{i=k}{\uparrow \circ_c}} val(ite(e_1, e_2, e_3))(i) && \text{Definition of } \overset{n}{\underset{i=k}{\uparrow \circ_c}} \text{ for } k \leq n
\end{aligned}$$

- *Induction Step:* Let $0 \leq k < N-1$. Then

$$\begin{aligned}
val(s)(k) &= \text{if } val(e_1)(k) \text{ then } val(e_2 \circ s[1,c])(k) \\
&\quad \text{else } val(e_3 \circ s[1,c])(k) && \text{Definition of } val \\
&= \text{if } val(e_1)(k) \\
&\quad \text{then } val(e_2)(k) \circ val(s)(k+1) \\
&\quad \text{else } val(e_2)(k) \circ val(s)(k+1) && \text{Definition of } val \text{ for } k+1 > N-1 \\
&= val(ite(e_1,e_2,e_3))(k) \circ val(s)(k+1) && \text{Definition of } val \\
&= val(ite(e_1,e_2,e_3))(k) \\
&\quad \circ (\overset{N-1}{\underset{i=k+1}{\uparrow}} \circ_c \, val(ite(e_1,e_2,e_3))(i)) && \text{Induction} \\
&= \overset{N-1}{\underset{i=k}{\uparrow}} \circ_c \, val(ite(e_1,e_2,e_3))(i) && \text{Definition of } \overset{n}{\underset{i=k}{\uparrow}} \circ_c \text{ for } k \leq n
\end{aligned}$$

Now we prove by induction over $k \in \{0,...,N-1\}$ that $val(s')(k) = \overset{k}{\underset{i=0}{\downarrow}} \circ_c \, val(ite(e_1,e_2,e_3))(i)$:

- *Base case:* Let $k = 0$. Then

$$\begin{aligned}
val(s')(k) &= \text{if } val(e_1)(k) \text{ then } val(s'[-1,c] \circ e_2)(k) \\
&\quad \text{else } val(s'[-1,c] \circ e_3)(k) && \text{Definition of } val \\
&= \text{if } val(e_1)(k) \text{ then } c \circ val(e_2)(k) \\
&\quad \text{else } c \circ val(e_3)(k) && \text{Definition of } val \text{ for } k-1 < 0 \\
&= c \circ val(ite(e_1,e_2,e_3))(k) && \text{Definition of } val \\
&= (\overset{k-1}{\underset{i=0}{\downarrow}} \circ_c \, val(ite(e_1,e_2,e_3))(i)) \\
&\quad \circ val(ite(e_1,e_2,e_3))(k) && \text{Definition of } \overset{n}{\underset{i=k}{\downarrow}} \circ_c \text{ for } k > n \\
&= \overset{k}{\underset{i=0}{\downarrow}} \circ_c \, val(ite(e_1,e_2,e_3))(i) && \text{Definition of } \overset{n}{\underset{i=k}{\downarrow}} \circ_c \text{ for } k \leq n
\end{aligned}$$

- *Induction Step:* Let $0 \leq k < N-1$. Then

$$\begin{aligned}
val(s')(k) &= \text{if } val(e_1)(k) \text{ then } val(s'[-1,c] \circ e_2)(k) \\
&\quad \text{else } val(s'[-1,c] \circ e_3)(k) && \text{Definition of } val \\
&= \text{if } val(e_1)(k) \\
&\quad \text{then } val(s')(k-1) \circ val(e_2)(k) \\
&\quad \text{else } val(s')(k-1) \circ val(e_3)(k) && \text{Definition of } val \text{ for } k-1 < 0 \\
&= val(s')(k-1) \circ val(ite(e_1,e_2,e_3))(k) && \text{Definition of } val \\
&= (\overset{k-1}{\underset{i=0}{\downarrow}} \circ_c \, val(ite(e_1,e_2,e_3))(i)) \\
&\quad \circ val(ite(e_1,e_2,e_3))(k) && \text{Definition of } \overset{n}{\underset{i=k}{\downarrow}} \circ_c \text{ for } k > n \\
&= \overset{k}{\underset{i=0}{\downarrow}} \circ_c \, val(ite(e_1,e_2,e_3))(i) && \text{Definition of } \overset{n}{\underset{i=k}{\downarrow}} \circ_c \text{ for } k \leq n
\end{aligned}$$

Thus follows:

$$\begin{aligned} val(s)(0) &= \overset{N-1}{\underset{i=0}{\uparrow}} \circ_c \, val(ite(e_1, e_2, e_3))(i) \\ &= \overset{N-1}{\underset{i=0}{\downarrow}} \circ_c \, val(ite(e_1, e_2, e_3))(i) && \text{Theorem 4.1} \\ &= val(s')(N-1) \end{aligned}$$

□

In the proof it was assumed that $s = ite(e_1, e_2 \circ s[1, c], e_3 \circ s[1, c])$. The proof for the alternative case, where $s = ite(e_1, s[1, c] \circ e_2, s[1, c] \circ e_3)$, is completely analogous but would need a new definition of $\overset{n}{\underset{i=k}{\uparrow}} \circ_c e(i)$ and $\overset{n}{\underset{i=k}{\downarrow}} \circ_c e(i)$. Therefore it is omitted here. If $\circ$ is commutative however, then it follows directly from the here shown proof. There is also the case where both forms are mixed, i.e. in $s = ite(e_1, e_2 \circ s[1, c], s[1, c] \circ e_3)$ or $s = ite(e_1, s[1, c] \circ e_2, e_3 \circ s[1, c])$. If $\circ$ is commutative then those streams can simply be rewritten into one of the above forms. For non-commutative operators these streams become very complex and rarely appear in practice, thus we will not consider them here.

4.4.1 Nested If-Expression

Nested if-expression can be seen as a single if-expression with a series of 'else if'-cases (as long as the conditions do not contain if-expressions). We can now simply translate this bigger case distinction in the same way as we did it before (so the consequence of each case i must have the form $e_i \circ s[1, c]$, respectively $s[1, c] \circ e_i$) and then translate the result back into a series of nested if-expressions.

4.4.2 Self References in the Condition

If we want to encode a statistic that collects the maximum over a positive input stream t without using the 'max'-operator, then we have to use if-expressions. When try to use forward-recursion, it would look like this:

$$max = ite(t > max[1, 0], t, max[1, 0])$$

In this stream we have a self-reference in the condition of the if-expression. Those streams are problematic and in general not translatable by our methods, because the value of the condition at each time step may be changed by the translation. If, however, the if-expression semantically encodes an associative operator (like here the max-operator), then there is always an equivalent efficiently monitorable stream This problem may be circumvented by using operators instead of corresponding if-expressions, whenever this is possible.

4.4.3 Cases with no Self References

Now imagine an if-expression that has only a self reference in the consequence or alternative, but not in the other part. As an example consider the LOLA stream $s = ite(e_1, e_2, e_3 \circ s[1, c])$, where e_1, e_2, e_3 do not depend on s. If we now recursively compute the value of s at some time point, then the recursion always terminates as soon as the condition becomes true. As a consequence we cannot simply rewrite the stream to $s' = ite(e_1, e_2, s[-1, c] \circ e_3)$, since $val(s)(0)$ and $val(s')(N-1)$ are not necessarily equal. For example, it could happen that the condition is true at times 0 and $N-1$. Then obviously $val(s)(0) = val(e_2)(0)$ and $val(s')(N-1) = val(e_2)(N-1)$, thus we cannot translate such if-expressions with our methods.

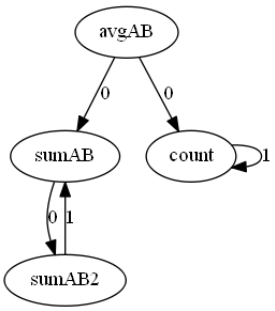

Figure 4.1: Dependency Graph

4.4.4 If-Expressions as Subexpressions

Consider a stream where an if-expression with self references only appears as a subexpression, for example the stream

$$s = e' \circ ite(e_1, e_2 \circ s[1, c], e_3 \circ s[1, c])$$

We can now follow two different strategies to bring this expression into a translatable form. The first one is to move the self reference upwards until we can translate the stream with the methods from sections 4.2 and 4.3. In this example we could use the distributivity of the consequence and alternative of if and the associativity of $\circ$, to arrive at

$$s = (e' \circ ite(e_1, e_2, e_3)) \circ s[1, c],$$

which we can easily translate into an efficiently monitorable stream. The alternative is to move the if-expression upwards, in order to be able to apply the methods from this section. In the example we can use the distributivity of the consequence and alternative of if in order to move e' inwards, and we would arrive at

$$s = ite(e_1, (e' \circ e_2) \circ s[1, c], (e' \circ e_3) \circ s[1, c]),$$

which can also be easily translated into an efficiently monitorable stream.

4.5 Application Example

Finally let us look at an application of the previously described methods on a little more complex example. Assume we have two integer input streams, called A and B, and the following set of output streams:

$$\begin{aligned} avgAB &= sumAB/count \\ sumAB &= A + sumAB2 \\ sumAB2 &= B + sumA[1, 0] \\ count &= ite(A > B, 1 + count[1, 0], 0 + count[1, 0]) \end{aligned}$$

Also assume that we have somehow given the task to print the value of $avgAB$ at the first execution step. The dependency graph of these streams can be seen in figure 4.1. Since $avgAB$ has no self loop and references both $sumAB$ and $count$ at the same offset, we can simply translate those two streams to translate $avgAB$.
Let us now look at $sumAB$. Here we have a single cycle of references containing $sumAB$ and $sumAB2$ as can be seen in the dependency graph. The sum of the offsets is $1 + 0 = 1$ and thus positive. We can now by substitution rewrite $sumAB$ to

$$sumAB = A + B + sumAB[1, 0].$$

Since + is associative and commutative, this can simply be translated to

$$sumAB' = sumAB'[-1, 0] + A + B$$

The stream *count* contains an if-expression, which is already in a translatable form. Thus we can just translate it to

$$count' = ite(A > B, count[-1, 0] + 1, count[-1, 0] + 0)$$

We now have an equivalent and efficiently monitorable set of LOLA streams:

$$\begin{aligned} avgAB' &= sumAB'/count' \\ sumAB' &= sumAB'[-1, 0] + A + B \\ count' &= ite(A > B, count[-1, 0] + 1, count[-1, 0] + 0) \end{aligned}$$

However, now we need to give the task to print the value of $avgAB$ at the last execution step, in order to have the same value printed.

5 Experiments

In this chapter we will shortly describe the implementation of our compiler and present some experimental results, which should demonstrate the usefulness of our algorithm.

5.1 Implementation

We implemented the automata-theoretic algorithm from chapter 3 as a compiler in Java. The compiler takes a LOLA specification (with the syntax of the Stanford LOLA implementation [1]) and returns an optimized LOLA specification with the same trigger behavior. This optimized specification contains no positive cycles in its dependency graph, if the input specification is completely of boolean type, and has less positive cycles, if the input specification uses also non-boolean types.
The core task of the optimization algorithm is the transformation of a 2AA into a DFA. This can only be done by using two subset constructions, which causes a double-exponential state blowup. The size of the generated optimized Specification is linear in the size of the DFA, therefore it makes sense to minimize the DFA first. In our implementation we used Hopcroft's algorithm [11] to accomplish this, which iteratively computes a partition over the state set corresponding to the Myhill-Nerode-Theorem. Thus we are able to keep the generated specification relatively small. However, this does not reduce the size of the intermediary representation, which is still facing a double-exponential blowup and could therefore cause memory problems. In order to reduce the state set size of the intermediary NFA, which is created from a 2AA, we used the efficient construction from [8]. It is basically the same construction as shown in chapter 3, only that breadth first search is used to find relevant states. Therefore no irrelevant states are generated, which heavily reduces the state set size. We used a similar technique to efficiently implement the translation from NFA to DFA.

5.2 Results

We used the Stanford LOLA implementation [1] to run tests for three different LOLA specifications

- S_1: Checks, whether each occurance of a is eventually followed by an occurance of b. This corresponds to the LTL formula $\square(a \rightarrow \lozenge b)$.

$$s = ite(a, evb, true)$$
$$evb = b \vee evb[1, false]$$
$$\textbf{trigger } \neg s$$

- S_2: Checks, whether each occurance of a is followed by b being true for the rest of the execution. This corresponds to the LTL formula $\square(a \rightarrow \square b)$.

$$s = ite(a, alb, true)$$
$$alb = b \wedge alb[1, true]$$
$$\textbf{trigger } \neg s$$

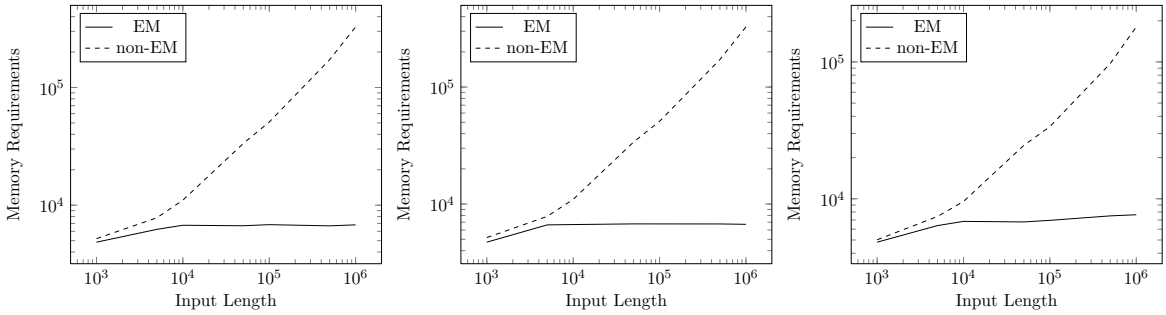

Figure 5.1: Worst case memory requirements for S_1, S_2 and S_3 (from left to right)

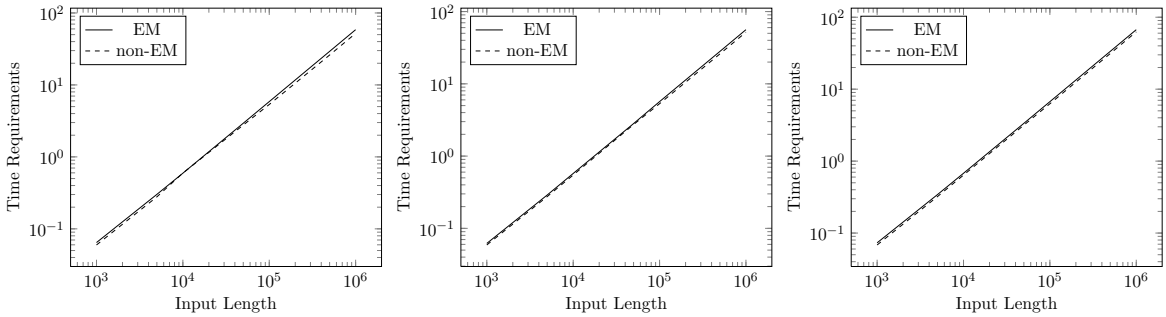

Figure 5.2: Worst case time requirements for S_1, S_2 and S_3 (from left to right)

- S_3: Checks, whether each occurance of a is eventually followed by an occurance of c or b is true since the start of the execution. This corresponds to the (past-)LTL formula $\square(a \rightarrow (\boxminus b \vee \Diamond c))$.

$$
\begin{aligned}
s &= ite(a, alb \vee evc, true) \\
alb &= b \wedge alb[-1, true] \\
evc &= c \vee evc[1, false] \\
&\textbf{trigger } \neg s
\end{aligned}
$$

We translated these specifications with the implemented compiler and ran the LOLA online monitoring algorithm with both the optimized and unoptimized specifications on on-the-fly generated worst case traces. For example, when using S_1 we generated the trace so that no b occurs. Note that the here generated worst-case traces can be interpreted as partial traces, whose length is equal to the amount of execution steps it takes for b to appear after an appearance of a. As trace lengths we used 1000,5000,10000,50000,100000,500000 and 1000000 steps. We ran these tests on an Intel Xeon Processor using Linux with 32GB RAM. The results for the memory requirements are depicted in figure 5.1. It can be seen that the optimized (EM) specifications perform much better than the original (non-EM) specifications for large traces. The time requirements (figure 1.2) are generally a little higher for the optimized specification, but the difference does not grow and is relatively small. The reason for this difference is the size of the generated specification (which is linear in the DFA size). More complex expressions need more time to be evaluated and thus cause an extra amount of time for the monitoring algorithm that is linear in the trace size. However, since we minimize the DFA, the additional size of the created specification should almost always be negligible in practice. The memory requirements behave as expected, except for the smaller trace sizes. The reason for this is probably that the traces with a size smaller than 10^4 are simply too small to create meaningful results.
We ran an additional test for specification S_1, where we tried to create randomized traces for the specification. The random generation of such a trace is a little problematic, because the

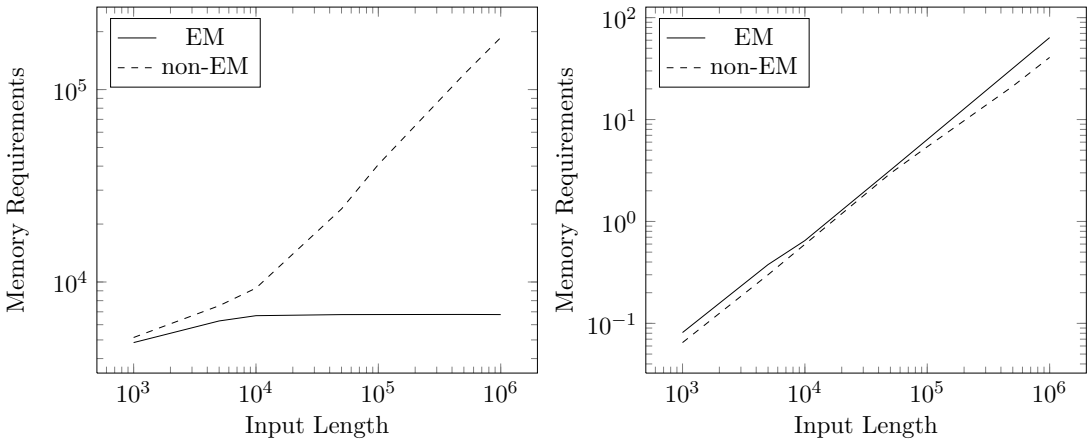

Figure 5.3: Average memory and time requirements for S_1 with random traces

distribution of events has to match the specification type. For S_1 we did it so that a is true at every execution step and the response time of b is initially set to a random value between 0 and the trace length. As soon as a number of execution steps has passed that is equal to the response time, b will be activated and a new random response time will be generated. We used a sample size of 200 and the average results are illustrated in figure 5.3. Again the difference in the time consumption is relatively small, even though the difference seems to rise for large traces. A reason for this could be irregularities in the random trace generation, which is inevitably part of the time measurement. The memory requirements behave the same as they do in the non-random experiment and as it is expected.

6 Conclusion

In this thesis we presented methods to automatically optimize forward-recursive LOLA specifications. We first focused on strictly boolean specifications and showed how to create equivalent LOLA specifications with the same trigger behavior and no forward-recursive stream definitions. We also implemented this algorithm as a compiler and presented experimental results, which showed that the optimized specification performed a lot better memory-wise with a very small time trade-off. The development of these methods produced an important side result: We showed that efficiently monitorable LOLA specifications have at least regular expressiveness and how to synthesize such specifications directly from automata-theoretic representations of regular properties.
Further, we examined the optimization of forward-recursive statistics in LOLA and presented simple rewriting rules for streams, which match a certain pattern, by using associativity. Unfortunately however, such rules are not applicable to more complex streams.
In general our methods showed good results for the verification of regular properties in worst-case scenarios and could be relevant for industrial usage. However, the size of the synthesized specification can be unsatisfying depending on the encoded property. Therefore it might make more sense to create a modified monitoring algorithm, that operates directly on automata instead of the synthesized specification. Such automata could previously be constructed from LOLA specifications as we have shown before. It might also be possible to further optimize the synthesis of LOLA specifications from DFAs by running sophisticated simplification algorithms on the synthesized LOLA expressions..

6.1 Open Problems

We have shown working methods to optimize boolean LOLA specifications, but we did not investigate how to completely remove forward-recursion from arbitrary typed specifications. These can be used to express context-free properties (e.g. 'every request has a matching grant') by using integer streams as stacks and can thus not be translated into equivalent DFAs. For a subset of these specifications it might be possible to translate them into equivalent pushdown automata, however this would become very difficult if integer streams are used in other ways than serving as a stack. The synthesis of efficiently monitorable LOLA specifications from context-free formalisms could be a subject of future work.
For statistics we only described rewriting rules that can be applied if LOLA streams match certain patterns, but we did not present a more general optimization procedure. The optimization of statistics can be reduced to the mathematical problem of solving the corresponding recurrence equations and bringing them into a closed form (i.e. removing the recursion). It is then easy to synthesize an efficiently monitorable LOLA specification that incrementally computes this closed form. However, the automatic determination of such a closed form is generally not very easy and often needs sophisticated mathematical methods.

Bibliography

[1] 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 *Temporal Representation and Reasoning, 2005. TIME 2005. 12th International Symposium on*, pages 166–174. IEEE, 2005.

[2] Roberto Bagnara, Alessandro Zaccagnini, and Tatiana Zolo. The automatic solution of recurrence relations. i. *Linear recurrences of finite order with constant coefficients. Quaderno*, 334, 2003.

[3] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *Verification, Model Checking, and Abstract Interpretation*, pages 44–57. Springer, 2004.

[4] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming*, 19(2):87–152, 1992.

[5] Laura Bozzelli and César Sánchez. Foundations of boolean stream runtime verification. In *Runtime Verification*, pages 64–79. Springer, 2014.

[6] Bernd Finkbeiner, Sriram Sankaranarayanan, and Henny Sipma. Collecting statistics over runtime executions. *Electronic Notes in Theoretical Computer Science*, 70(4):36–54, 2002.

[7] Bernd Finkbeiner and Henny Sipma. Checking finite traces using alternating automata. *Formal Methods in System Design*, 24(2):101–127, 2004.

[8] Paul Gastin and Denis Oddoux. Ltl with past and two-way very-weak alternating automata. In *Mathematical Foundations of Computer Science 2003*, pages 439–448. Springer, 2003.

[9] Thierry Gautier, Paul Le Guernic, and Loic Besnard. Signal: A declarative language for synchronous programming of real-time systems. In *Functional programming languages and computer architecture*, pages 257–277. Springer, 1987.

[10] Klaus Havelund and Grigore Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 342–356. Springer, 2002.

[11] John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, DTIC Document, 1971.

[12] Dexter C Kozen. *Automata and computability*. Springer Science & Business Media, 2012.

[13] Zohar Manna and Henny B Sipma. Alternating the temporal picture for safety. In *Automata, Languages and Programming*, pages 429–450. Springer, 2000.

[14] Nicolas Markey. Temporal logic with past is exponentially more succinct. *EATCS Bulletin*, 79:122–128, 2003.

[15] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[16] Moshe Vardi. Alternating automata and program verification. *Computer Science Today*, pages 471–485, 1995.

[17] Moshe Y Vardi. Reasoning about the past with two-way automata. In *Automata, Languages and Programming*, pages 628–641. Springer, 1998.