UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

From 2-Way Nondeterministic Büchi Automata to Alternating Büchi Automata

Umwandlung: 2-Wege Nichtdeterministische Büchi-Automaten zu Alternierenden Büchi-Automaten

Bachelorarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Marco Andreas Kabelitz

ausgegeben und betreut von
Prof. Dr. Martin Leucker

mit Unterstützung von
Torben Scheffel

Lübeck, den 19. Februar 2015

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Lübeck, den 19. Februar 2015

Abstract

Automata theory is an integral component of runtime verification, where the transformation of members from one automata class to another is necessary on a regular basis. In this thesis, we discuss and implement three different conversion methods for finite state automata. The first conversion describes the translation of 2-way nondeterministic Büchi automata into language equivalent alternating Büchi automata. Next, we depict the complementation of alternating Büchi automata by means of weak alternating automata. Combining these methods, we investigate the complementation of 2-way nondeterministic Büchi automata. After their theoretical discussion, we present the implementation of these conversions in the context of the logic and automata library RltlConv.

Zusammenfassung

Die Automatentheorie ist ein wesentlicher Bestandteil der Laufzeitverifikation, in welcher es häufig notwendig ist, Automaten bestimmter Klassen in Automaten anderer Klassen umzuwandeln. In dieser Arbeit werden drei unterschiedliche Umwandlungsmethoden für endliche Automaten besprochen und realisiert. Die erste Umwandlung beschreibt die Transformation von 2-Wege nichtdeterministischen Büchi-Automaten zu alternierenden Büchi-Automaten, welche dieselbe Sprache akzeptieren. Danach schildern wir die Komplementierung von alternierenden Büchi-Automaten unter Einsatz von schwachen alternierenden Automaten. Die Kombination dieser Konstruktionen führt uns schließlich zur Komplementierung von 2-Wege nichtdeterministischen Büchi-Automaten. Im Anschluss an die theoretische Diskussion dieser Methoden präsentieren wir eine Implementierung im Rahmen der Logik- und Automaten-Biblithek RltlConv.

Contents

Chapter 1

Introduction

The proceeding application of software systems in all areas of work and society gives rise to an increasing demand for error-free software. Meeting this demand is hindered by the increasing complexity of these systems themselves as well as by the complexity of their development processes. Misconceptions in technical design and software errors resulting from inappropriate implementations cause great expenses to developers and clients and can lead to unwanted or even dangerous behaviors of software systems.

A famous example for such a harmful malfunctioning is the computerized radiation therapy machine Therac-25. Between 1985 and 1987, there have been six confirmed cases of massive radiation overdoses caused by concurrent programming errors (see [8]). These overdoses led to serious injuries and were even lethal in two cases. With software systems deeply embedded in such safety-critical areas as health care, power generation and public transport, we have to diminish the probability of occurrence of such software errors.

One way to reduce these risks is the application of *formal verification* methods. The idea is to specify the intended system behavior in a formal description and verify whether this specification holds for every possible execution of the system or not. This process is called *model checking*. Unfortunately, considering all potential executions of a system is quite a difficult endeavor, especially if the system's processing is not intended to terminate. So instead, we content ourselves with monitoring the system as it runs. Step by step, we validate whether its current state still satisfies the intended behavior. This process is called *runtime verification*.

A well known method for specifying a system's behaviour according to discrete time steps is the usage of temporal logics like the *linear temporal logic* (LTL). Here we can describe in a logical formula, which property the system should fulfill for a certain time step during the course of its execution. To monitor the system during runtime, we can construct a *finite state automaton* (FSA) from a given temporal logical expression. This automaton can then process the current state of the system at every time step and decide, whether this state is intended by the temporal logical formula or not. In practice, such a monitoring could be implemented as a parallel process on the same device as the original system or it could exist on a separated physical instance like an FPGA.

There are many different kinds of temporal logics with varying expressive power, so we need different methods for converting logical formulas to finite state automata. Furthermore, the automaton offering the most intuitive representation of a logical formula is not always the most adequate automaton for practical purposes. Sometimes it is necessary to convert a formula to the desired automaton via various intermediate automata transformations. We are therefore interested in methods converting certain types of FSAs into other types of FSAs without changing the original semantics.

In this thesis, we will discuss such automaton-to-automaton conversions. There are three different methods we will present:

- 2-way nondeterministic Büchi automata to alternating Büchi automata proposed by Piterman and Vardi in [11].
- Complementation of alternating Büchi automata due to Kupferman and Vardi [6].
- Complementation of 2-way nondeterministic Büchi automata by composition of the two former constructions.

In *Chapter 2*, we will provide the automata theoretical framework we build upon in this thesis. We start by giving a review of finite state automata on finite words. This will include the definitions of deterministic and nondeterministic automata as well as the discussion of 2-way and alternating automata. The main part of this chapter will introduce the reader to the notion of finite state automata on infinite words and ω-regular languages. We will therefore investigate different kinds of Büchi automata as well as weak alternating automata and parity automata.

Chapter 3 represents the main part of this thesis. We discuss the automaton conversion methods stated above by reproducing the results given in [11] and [6] and providing further explanation of the ideas behind them.

We will then describe an implementation of these conversion methods in *Chapter 4*. We introduce the reader to the logic and automata library *RltlConv* and explain the embedding of the formerly discussed constructions.

Finally, *Chapter 5* will contain our conclusion and an outlook to future tasks related to this thesis.

Chapter 2

Basics of Automata Theory

Automata theory is one of the basic fields of theoretical computer science. It provides many powerful tools like finite state automata, pushdown automata, cellular automata and Turing machines, which can be used for problem analysis in recursion and complexity theory, the design of compilers and programming languages, text processing, formal verification, computational immunology or artificial intelligence.

We only want to consider the class of *finite state automata* (FSAs) in this thesis. Although the least powerful one of the referred classes, finite state automata bring a lot of useful properties and a striking conceptual simplicity with them.

In this chapter, we will first give a review of finite state automata on finite words. We expect the reader to have basic knowledge of regular languages and the concepts of deterministic and nondeterministic finite automata. Nevertheless, we provide the most important definitions and properties to fix the notation used throughout this thesis (but we will go without informal explanations and examples). We will then look into 2-way finite state automata and alternating automata.

Afterwards, we will expand the notion of FSAs on finite words and introduce the reader to FSAs on infinite words. We will describe different classes of ω-automata (mainly Büchi automata) and discuss properties of ω-regular languages. These concepts provide the basis for the automata conversions described in the following chapters.

2.1 Finite State Automata on Finite Words

In this section, we will describe deterministic and nondeterministic finite state automata processing finite words. We will then extend these models to 2-way automata as well as to alternating automata. Both of these classes will be used in the automata conversions described in *Chapter 3*.

Definition 2.1 (Finite word)**.** *Let* Σ *be a finite and nonempty alphabet. A* finite word over Σ *is a sequence* $w = w_0 w_1 ... w_l$ *with* $l \in \mathbb{N}_0$ *and* $w_i \in \Sigma$ *for every* $i \in \{0, \dots, l\}$. *We say* w *to be of length* $l + 1$ *and denote this by* $|w| = l + 1$. *We also include the* empty word ε *in our definition, which represents the unique finite word of length* 0.

We denote the language of all finite words over Σ *by* Σ^*. *The language of all nonempty finite words over* Σ *is denoted by* Σ^+.

We refer to the concatenation of the words $w, v \in \Sigma^*$ *by* $wv \in \Sigma^*$.

Following directly from this definition, we get $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$.

By having fixed this notation, we can now introduce different classes of finite state automata on finite words. We start by defining deterministic finite automata. We remark, that we use the same notation as Piterman and Vardi do in [11].

Definition 2.2 (Deterministic finite automaton [5])**.** *A* deterministic finite automaton *(DFA) is a 5-tuple*

$$M = (\Sigma, S, s_0, \delta, F)$$

where

- Σ *is the input alphabet,*
- S *is the finite set of states,*
- $s_0 \in S$ *is the start state,*
- $\delta : S \times \Sigma \to S$ *is the transition function, assigning each pair* (s, a) *with* $s \in S$ *and* $a \in \Sigma$ *to another state* $t \in S$,
- $F \subseteq S$ *is the set of accepting states.*

We define the extended transition function $\hat{\delta} = S \times \Sigma^* \to S$ *by induction over the length of* $w \in \Sigma^*$:

$$\begin{aligned} \hat{\delta}(s, \varepsilon) &:= \quad s \\ \hat{\delta}(s, wa) &:= \delta(\hat{\delta}(s, w), a) \end{aligned}$$

We say a word $w \in \Sigma^*$ to be accepted by M if and only if $\hat{\delta}(s_0, w) \in F$. Otherwise, the word is said to be rejected. $L(M)$, the language accepted by M, is the set of all words $w \in \Sigma^*$ accepted by M, formally written as:

$$L(M) = \{w \in \Sigma^* \mid \hat{\delta}(s_0, w) \in F\}$$

A language accepted by a deterministic finite automaton is called a *regular language*. The set of regular languages is known to be closed under union, intersection, complement, concatenation, difference and the Kleene star [5].

Definition 2.3 (Nondeterministic finite automaton [5]). *A* nondeterministic finite automaton *(NFA) is a 5-tuple*

$$N = (\Sigma, S, S_0, \delta, F)$$

where Σ, S and F are corresponding to the definition of a DFA, and S_0 and δ are defined as follows:

- *$S_0 \subseteq S$ is the set of start states.*
- *$\delta : S \times \Sigma \to 2^S$ is the transition function, assigning each pair (s, a) with $s \in S$ and $a \in \Sigma$ to an $A \in 2^S$, where 2^S denotes the power set of S (so $A \subseteq S$).*

We define the extended transition function $\hat{\delta} = 2^S \times \Sigma^* \to 2^S$ by induction over the length of $w \in \Sigma^*$:

$$\begin{aligned} \hat{\delta}(A, \varepsilon) &:= A \\ \hat{\delta}(A, wa) &:= \bigcup_{s \in \hat{\delta}(A,w)} \delta(s, a) \end{aligned}$$

We say a word $w \in \Sigma^*$ to be accepted by N if and only if $\hat{\delta}(S_0, w) \cap F \neq \emptyset$. Just like with a DFA, we define the language accepted by N as the set of all words over Σ accepted by N, formally written as:

$$L(N) = \{w \in \Sigma^* \mid \hat{\delta}(S_0, w) \cap F \neq \emptyset\}$$

There are some remarks to be made about NFAs. The most important one is, that for every language accepted by an NFA, there exists a DFA accepting this language as well (shown by Rabin and Scott in [12] in 1959). Therefore NFAs aren't more powerful than DFAs, both are accepting only regular languages.

Next, we will define NFAs with ε-transitions as an extension of NFAs. These aren't any more powerful than NFAs (and therefore not more powerful than DFAs), but are often easier to construct than normal NFAs.

Definition 2.4 (Nondeterministic finite automaton with ε-transitions [5])**.** *A* nondeterministic finite automaton with ε-transitions *(ε-NFA, also called* NFA with ε-moves*) is a 6-tuple*

$$N = (\Sigma, \varepsilon, S, S_0, \delta, F)$$

where ε is a special symbol not occurring in Σ and

$$N_\varepsilon = (\Sigma \cup \{\varepsilon\}, S, S_0, \delta, F)$$

is an ordinary NFA over the alphabet $\Sigma \cup \{\varepsilon\}$.

We say a word $w \in \Sigma^*$ to be accepted by an ε-NFA N if and only if there exists a word $y \in (\Sigma \cup \{\varepsilon\})^*$ such that

- *N_ε accepts y and*
- *w is obtained from y by erasing all occurrences of the symbol ε, that is, $w = h(y)$, where*

  $$h : (\Sigma \cup \{\varepsilon\})^* \to \Sigma^*$$

  is the homomorphism defined by:

  $$\begin{aligned} h(a) :=& \quad a, \qquad a \in \Sigma \\ h(\varepsilon) :=& \quad \varepsilon \end{aligned}$$

The language accepted by the ε-NFA N is defined as:

$$L(N) = h(L(N_\varepsilon))$$

We will hereafter refer to both NFAs and ε-NFAs as NFAs, meaning, that our notion of nondeterminism will always include the possibility of ε-moves while discussing the description of an ordinary NFA.

According to the provided definition, NFAs possess a set of start states. Since Piterman and Vardi are using automata with single start states in [11], we like to recall that for every NFA N there exists another NFA N' using just a single start state with $L(N) = L(N')$.

N' can be achieved from N by adding a single new state s_0 to N, declaring s_0 as the only start state and inserting ε-transitions from s_0 to all former start states.

After reviewing these basic types of FSAs, we will now turn our attention to 2-way nondeterministic finite automata and alternating finite automata. We present extended definitions of the ones given by Piterman and Vardi in [11].

Definition 2.5 (2-way nondeterministic finite automaton [11])**.** *A* 2-way nondeterministic finite automaton *(2NFA) is a 5-tuple*

$$N = (\Sigma, S, s_0, \delta, F)$$

where Σ, S and F are corresponding to the definition of an ordinary NFA, and

- *s_0 is the single start state,*
- *$\delta : S \times \Sigma \to 2^{S\times\{-1,0,1\}}$ is the transition function, assigning each pair (s, a) with $s \in S$ and $a \in \Sigma$ to a set of pairs (t, Δ) with $t \in S$ and $\Delta \in \{-1, 0, 1\}$.*

A run of a 2NFA on a finite word $w = w_0w_1...w_l$ is a finite sequence of state-index pairs $\rho = (t_0, i_0), (t_1, i_1), \dots, (t_m, i_m)$ with:

- $t_0 = s_0$
- $i_0 = 0$
- $t_j \in S$ *for* $0 \leq j \leq m$
- $i_j \in \{0, 1, \dots, l\}$ *for* $0 \leq j < m$
- $i_m \in \{0, 1, \dots, l + 1\}$
- $(t_{j+1}, i_{j+1} - i_j) \in \delta(t_j, w_{i_j})$ *for* $0 \leq j < m$

A run is accepting if and only if $i_m = l + 1$ and $t_m \in F$. We describe the language accepted by a 2NFA N as follows:

$$L(N) = \{w \in \Sigma^* \mid \textit{There exists an accepting run of } N \textit{ over } w\}$$

The idea behind a 2NFA is, that the automaton can not only read the input word linearly from front to back (like an NFA is supposed to), but also can change its reading direction multiple times. Therefore every transition is enriched with an element of $\{-1, 0, 1\}$, indicating whether the automaton takes a step backward, no step at all, or a step forward on the input word. A run can be interpreted as the 2NFA processing a word w for m steps. At step j, it is in state t_j and reads the symbol at position i_j. To accept a word, the automaton needs to read the whole input at least once and has to terminate in an accepting state.

We already know that nondeterminism does not extend the power of a finite state automaton over finite words. In [14], Shepherdson shows the same result for 2-way finite state automata over finite words. Therefore, the class of languages accepted by 2NFAs is precisely the class of regular languages.

For illustration of the ideas just presented, we give an example of a 2NFA and an accepting run.

Example 2.1 (2NFA and accepting run). *Consider the 2-way nondeterministic finite automaton $N = (\Sigma, S, s_0, \delta, F)$ with $\Sigma = \{a, b\}$, $S = \{s_0, s_1, s_2, s_3\}$, $F = \{s_2\}$, and δ given by the transition table 2.1.*

	a	b
s_0	$\{(s_0, 1)\}$	$\{(s_0, 1), (s_1, -1)\}$
s_1	$\{(s_2, 1)\}$	$\{(s_3, 1)\}$
s_2	$\{(s_3, 1)\}$	$\{(s_2, 1)\}$
s_3	$\{(s_3, 1)\}$	$\{(s_3, 1)\}$

Table 2.1: Transition table of the 2NFA N

A graphical representation of this 2NFA is provided in Figure 2.1. N accepts the language

$$L(N) = \{w \in \{a, b\}^* \mid w = xay \text{ with } x \in \{a, b\}^* \text{ and } y \in \{b\}^+\}$$

which denotes the language of all $w \in \{a, b\}^*$ containing the symbol a at least once and ending on a nonempty sequence of b's only. To do so, the automaton "nondeterministically guesses" which one of the b's will be the first one of the b-only suffix and goes one step back to check whether the symbol previously read was an a or not.

An accepting run for the word $ababb \in L(N)$ would be

$$\rho = (s_0, 0), (s_0, 1), (s_0, 2), (s_0, 3), (s_1, 2), (s_2, 3), (s_2, 4), (s_2, 5)$$

which is accepting since $t_m = s_2 \in F$ and $i_m = 5 = l + 1$.

We remark, that we can modify the definition of a run of a 2NFA easily to get a definition of a run of an NFA: A run on a finite word $w = w_0 w_1 ... w_l$ is a finite sequence of state-index pairs $\rho = (t_0, i_0), (t_1, i_1), \dots, (t_l, i_l)$ with $t_j \in S$ and $i_j = j$ for $0 \leq j \leq l$. Again, we

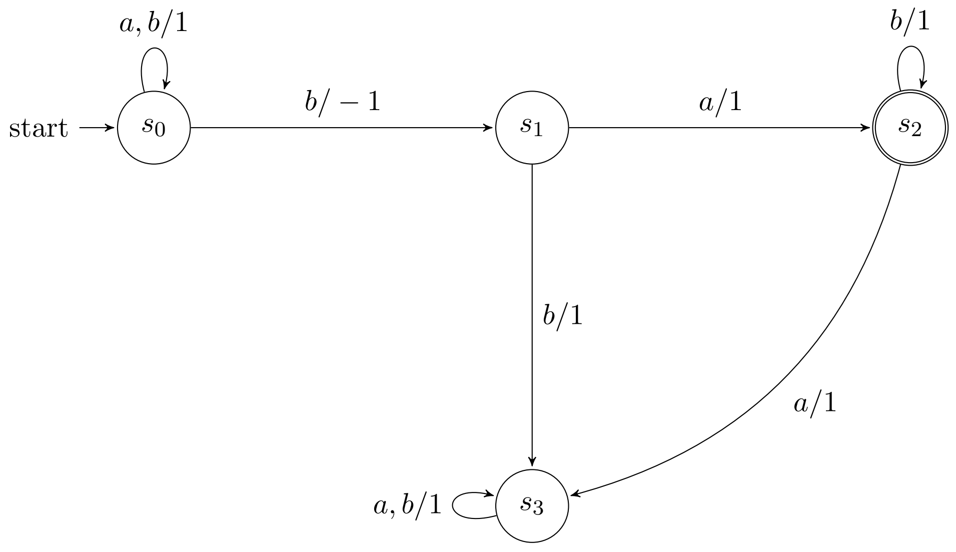

FIGURE 2.1: A graphical representation of the 2NFA N

demand $t_0 = s_0$ and $i_0 = 0$. We also have $(t_{j+1}) \in \delta(t_j, w_{i_j})$ for every j with $0 \leq j < l$. We say such an NFA run to be accepting if and only if $t_l \in F$.

Definition 2.6 (Simple run of a 2NFA). *Let $\rho = (t_0, i_0), (t_1, i_1), \ldots, (t_m, i_m)$ be a run of a 2NFA. We say ρ to be a* simple run *if and only if there exists no two state-index pairs (t_j, i_j) and (t_k, i_k) with $t_j = t_k$ and $i_j = i_k$ for $0 \leq j < k \leq m$.*

For a non-simple run, there's a "loop" in the run, visiting a certain state while reading at the same index of the word twice. A simple run on the other hand is loop free.

Theorem 2.1. *A 2NFA N accepts a word w iff N accepts w with a simple run.*

Proof.[11] Given an accepting non-simple run ρ of a 2NFA N over a word w, we will construct an accepting simple run ρ' over w from ρ. Since ρ is not simple, there exist some state-index pairs (t_j, i_j) and (t_k, i_k) with $t_j = t_k$ and $i_j = i_k$ for some $j < k$. We delete all state-index pairs (t_l, i_l) from ρ with $j < l \leq k$ and achieve the sequence $(s_0, 0), \ldots, (t_j, i_j), (t_{k+1}, i_{k+1}), \ldots, (t_m, i_m)$. Since $(t_{k+1}, i_{k+1} - i_k) \in \delta(t_k, w_{i_k})$ and $\delta(t_k, w_{i_k}) = \delta(t_j, w_{i_j})$, this sequence must still be a valid and accepting run. We repeat this process as long as there are still loops to be deleted. Since ρ is a finite sequence, there can only be finitely many loops. So in the end, we get a loop free accepting run ρ' of N over w, which complies with the definition of an accepting simple run. $\square$

So now we know, that whenever a word w is accepted by a 2NFA, w can be accepted by a simple run. We will use this property for automata constructions in *Chapter 3*.

The next class of finite state automata which will be of an important role in the course of this thesis is the class of alternating finite automata. Before we are able to discuss them properly, we need to fix some notation.

Definition 2.7 (Positive Boolean formulas over a set)**.** *Let S be a set. We denote the* set of all positive Boolean formulas over S *by $\mathcal{B}^+(S)$ and define it recursively as follows:*

$$\begin{aligned} \textbf{true} &\in \mathcal{B}^+(S) \\ \textbf{false} &\in \mathcal{B}^+(S) \\ s &\in \mathcal{B}^+(S), & s &\in S \\ \varphi_1 \wedge \varphi_2 &\in \mathcal{B}^+(S), & \varphi_1, \varphi_2 &\in \mathcal{B}^+(S) \\ \varphi_1 \vee \varphi_2 &\in \mathcal{B}^+(S), & \varphi_1, \varphi_2 &\in \mathcal{B}^+(S) \end{aligned}$$

Let $\varphi \in \mathcal{B}^+(S)$ be a positive Boolean formula and $S' \subseteq S$. S' satisfies φ if and only if φ evaluates to **true** *when assigning* **true** *to all elements in S' and* **false** *to all elements in $S \setminus S'$. If S' satisfies φ, we say S' to be a* model of φ*. If S' is a model of φ and there exists no $S'' \subset S'$ for which S'' also is a model of φ, we say S' to be* minimal model of φ *and denote this by $S' \models \varphi$. The formula $\varphi =$* **true** *is satisfied by every set, while $\varphi =$* **false** *cannot be satisfied.*

We achieve the dual *of a formula $\varphi \in \mathcal{B}^+(S)$, which we will refer to as $\widetilde{\varphi}$, by replacing all $\wedge$ by $\vee$,* **true** *by* **false** *and vice versa.*

We remark, that the necessity of including **true** and **false** explicitly into $\mathcal{B}^+(S)$ arises from the exclusion of negations. Otherwise, **true** and **false** would be expressible by $(s \vee \neg s)$ and $(s \wedge \neg s)$.

Definition 2.8 (Tree, Rooted tree, Σ-labeled tree)**.** *Let $G = (V, E)$ be a connected cycle-free graph. We call G a* tree *and define $dist : V \times V \to \mathbb{N}_0$ to be a function assigning each pair of vertices (u, v) to the length of the unique path from u to v in G.*

A rooted tree *is a directed graph $T = (V', E')$, for which the undirected underlying graph $G = (V', E)$ is a tree and there exists a root vertex $v_r \in V'$, such that for all directed edges $(u, v) \in E'$ the equation*

$$dist(v_r, v) - dist(v_r, u) = 1$$

holds. We say a vertex v to be of depth $d(v) = dist(v_r, v)$ in the rooted tree and call all vertices with an outdegree of $deg^+(v) = 0$ a leaf. For every directed edge $(u, v) \in E'$ we say u to be the predecessor of v and v to be a successor of u.

Given an alphabet Σ, we define a Σ-labeled tree *as a tuple (T, r) where $T = (V, E)$ is a rooted tree and $r : V \to \Sigma$ is a function assigning a symbol of Σ to every vertex in V.*

Following from this definition, a directed graph is a rooted tree if all edges are naturally oriented away from the root vertex v_r and all vertices v have an indegree of $deg^{-}(v) = 1$ (excepted the root itself, which has an indegree of $deg^{-}(v_r) = 0$). The root has no predecessor, the leaves have no successors and all remaining vertices have exactly one predecessor and at least one successor.

Definition 2.9 (Alternating finite automaton [11])**.** *An* alternating finite automaton *(AFA) is a 5-tuple*

$$\mathcal{A} = (\Sigma, Q, q_0, \eta, F)$$

where Σ, Q, q_0 and F are corresponding to the definition of an ordinary NFA, and the transition function $\eta : Q \times \Sigma \to \mathcal{B}^{+}(Q)$ assigns each pair (q, a) with $q \in Q$ and $a \in \Sigma$ to a positive Boolean formula over Q.

A run of an AFA on a finite word $w = w_0 w_1 ... w_l$ is a Q-labeled tree $\rho = (T, r)$ with $T = (V, E)$, $r : V \to Q$, $r(v_r) = q_0$ and

$$\begin{aligned} &\forall v \in V \text{ with } r(v) = q \text{ and } \eta(q, w_{d(v)}) = \varphi : \\ &\qquad \exists Q' \subseteq Q : Q' \models \varphi, deg^{+}(v) = |Q'| \text{ and } \forall q' \in Q' : \\ &\qquad\qquad \exists!(v, w) \in E : r(w) = q' \end{aligned}$$

We say a run of an AFA to be accepting if and only if the following condition holds for all leaves v: If the depth of a leaf is $d(v) = l + 1$, $r(v)$ has to be an element of F. Otherwise, $\eta(r(v), w_{d(v)})$ has to be equal to **true***. We define the* language accepted by an AFA $\mathcal{A}$ *as follows:*

$$L(\mathcal{A}) = \{w \in \Sigma^{*} \mid \text{There exists an accepting run of } \mathcal{A} \text{ over } w\}$$

We define the dual *of an AFA $\mathcal{A}$ as the AFA $\widetilde{\mathcal{A}} = (\Sigma, Q, q_0, \widetilde{\eta}, Q \setminus F)$.*

Please note, that we changed the naming of the state set from S to Q for this definition, which will serve a clearer distinction between alternating and non-alternating automata in the upcoming automata conversions.

Let's give an informal description of the operating principles of an AFA: There are two kinds of transitions in an AFA, called *existential* and *universal* transitions. An existential transition works just like a nondeterministic transition familiar from NFAs, where we have the choice of either going into state q_0 or into state q_1. This can be expressed by the positive Boolean formula $q_0 \vee q_1$. With a universal transition, there is no such choice, the automaton has to move to both q_0 and q_1, which can be interpreted as moving to q_0 and spawning another copy of the AFA moving to q_1. This can be expressed by the positive Boolean formula $q_0 \wedge q_1$. Existential and universal transitions may be combined arbitrarily, therefore the automaton is called *alternating.*

So when does an AFA accept a word w? If we recall the definition of runs of 2NFAs, we can take such a run as a path through the automaton, which can only be accepting, if it ends in an accepting state. But since AFAs have universal transitions, we're not able to interpret a run of an AFA as a path. We have to take account of possible branchings caused by such transitions. Therefore we defined the run of an AFA as a labeled tree, which is accepting, if and only if all leaves at depth $l + 1$ are labeled by some accepting state and all other leaves have to evaluate to **true** according to the transition function.

One remark on duals of AFAs: It is shown in [2], that the two AFAs $\mathcal{A}$ and $\widetilde{\mathcal{A}}$ accept complementary languages, so $L(\widetilde{\mathcal{A}}) = \Sigma^* \setminus L(\mathcal{A})$.

We will now provide an example of an AFA and an accepting run on this automaton.

Example 2.2 (AFA and accepting run). *Consider the following alternating finite state automaton $\mathcal{A} = (\Sigma, Q, q_0, \eta, F)$ with $\Sigma = \{a, b\}$, $Q = \{q_0, q_1, q_2, q_3\}$, $F = \{q_2\}$, and η given by the transition table 2.2.*

	a	b
q_0	$q_0 \vee (q_1 \wedge q_2)$	q_0
q_1	q_3	**true**
q_2	q_3	q_2
q_3	q_3	q_3

TABLE 2.2: Transition table of the AFA $\mathcal{A}$

A graphical representation of the automaton is provided by Figure 2.2. $\mathcal{A}$ accepts the same language as the 2NFA presented on page 8, namely

$$L(\mathcal{A}) = \{w \in \{a, b\}^* \mid w = xay \textit{ with } x \in \{a, b\}^* \textit{ and } y \in \{b\}^+\}$$

which denotes the language of all $w \in \{a, b\}^*$ containing at last one a and ending on a nonempty sequence of b's only. The automaton "nondeterministically guesses" the last a in the word and goes into two branches, one checking the suffix of the for containing only b's and the other checking if this suffix is at least of length 1.

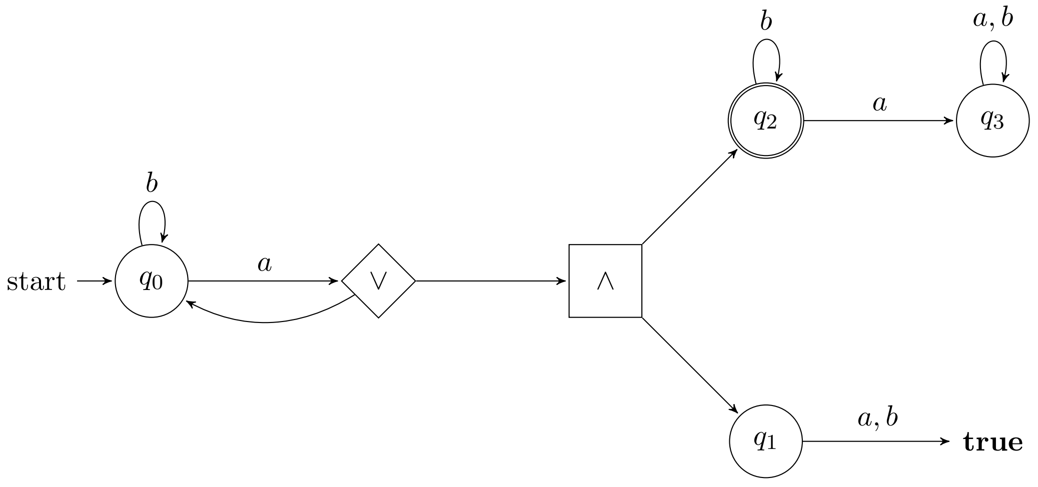

FIGURE 2.2: A graphical representation of the AFA $\mathcal{A}$

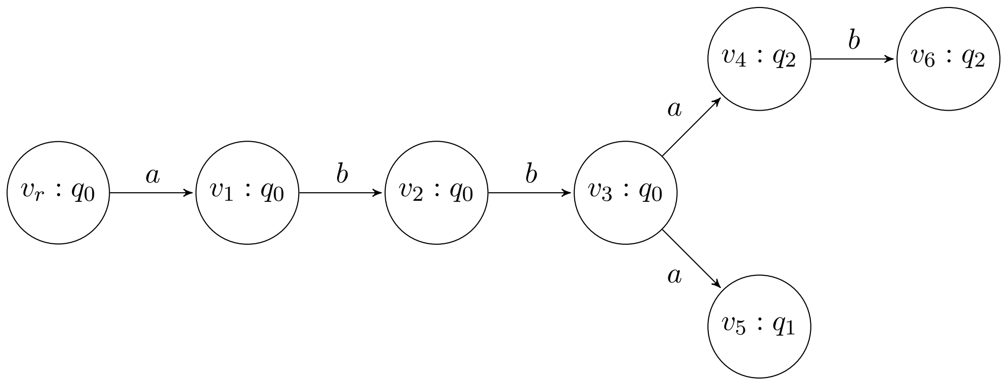

FIGURE 2.3: An accepting run of the AFA $\mathcal{A}$ over the word *ababb*

Figure 2.3 shows a run ρ for the word ababb $\in L(\mathcal{A})$, which is accepting, since v_6, the only leaf at depth $l+1$, is labeled with $q_2 \in F$ and for the other leaf v_5 we got $r(v_5) = q_1$ and $\eta(q_1, b) = \mathbf{true}$.

Again, we know AFAs to be no more powerful than NFAs (shown by Chandra in [2]). Therefore we got an equality of expressive power for all classes of automata introduced so far: DFAs, NFAs, ε-NFAs, 2NFAs and AFAs, they all accept the set of regular languages. This implies, that for every automaton M of one of these classes, there exists an automaton M' for every of the other classes with $L(M) = L(M')$. Nevertheless, two automata from different classes accepting the same regular language can differ tremendously in their number of states. It is a well known result, that the construction of a language equivalent DFA to an NFA can in a worst case scenario include an exponential blowup in the number of states. Vardi shows in [15], that the conversion of a 2-way automaton to a 1-way automaton involves an exponential blowup as well. Chandra shows in [2], that the simulation of an AFA using a DFA can even lead to a double exponential growth in the number of states. In *Chapter 3*, we will describe a method proposed by Piterman and Vardi in [11] which converts a 2NFA with n states into an language equivalent AFA with $\mathcal{O}(n^2)$ states.

2.2 Finite State Automata on Infinite Words

In this section, we will describe finite state automata processing infinite words. While FSAs on finite words are widely used in text processing and computational immunology, FSAs on infinite words find application in different areas of formal verification like *runtime verification* or *model checking*. The automata used for these purposes are quite similar to those automata classes we already discussed in the previous section. The difference lies in the acceptance behavior, since the reading process of an infinite word can't halt at the end of such an input.

Although there are other kinds of FSAs on infinite words, we will content ourselves with the discussion of different kinds of *Büchi automata* according to their definitions in [11]. These are the automata we will use in the automata conversions described in *Chapter 3*. For an introduction to *Muller*, *Rabin* and *Streett automata*, we refer the reader to [10].

Definition 2.10 (Infinite word)**.** *Let* Σ *be a finite and nonempty alphabet. An* infinite word over Σ *is an infinite sequence* $w = w_0 w_1 w_2 ...$ *with* $w_i \in \Sigma$ *for every* $i \in \mathbb{N}_0$*. We say* w *to be of length* $|w| = \omega$*.*

We denote the language of all infinite words over Σ *by* Σ^ω*.*

We write $x^\omega = xxx...$ *for the infinite concatenation of a finite word* x *to itself.*

For example, the word $w = a(bc)^\omega \in \{a, b, c\}^\omega$ is the infinite word *abcbcbcbc...* with an infinite suffix of alternating b's and c's.

Definition 2.11 (Nondeterministic Büchi automaton [11])**.** *A* nondeterministic Büchi automaton *(NBA) is a 5-tuple*

$$N = (\Sigma, S, s_0, \delta, F)$$

equally defined to a regular NFA.

A run of an NBA over an infinite word $w = w_0 w_1 ...$ *is an infinite sequence of state-index pairs* $\rho = (t_0, i_0), (t_1, i_1), \dots$ *with* $t_j \in S$ *and* $i_j \in \mathbb{N}_0$ *for* $j \in \mathbb{N}_0$*. We demand* $t_0 = s_0$*,* $i_0 = 0$ *and* $(t_{j+1}) \in \delta(t_j, w_{i_j})$ *for every* $j \in \mathbb{N}_0$*.*

Let $\inf(\rho)$ *denote the set of all states visited infinitely often in* ρ*. We say a run to be accepting if and only if* $\inf(\rho) \cap F \neq \emptyset$*. We define the language accepted by an NBA* N *as follows:*

$$L(N) = \{w \in \Sigma^\omega \mid \textit{There exists an accepting run of } N \textit{ over } w\}$$

This means that a word w is accepted by an NBA if the automaton visits some state $s \in F$ infinitely often while processing w.

Example 2.3 (NBA and accepting run)**.** *Consider the following nondeterministic Büchi automaton* $N = (\Sigma, S, s_0, \delta, F)$ *with* $\Sigma = \{a, b\}$*,* $S = \{s_0, s_1, s_2\}$*,* $F = \{s_1\}$*, and* δ *given by the transition table 2.3.*

	a	b
s_0	$\{s_0\}$	$\{s_0, s_1\}$
s_1	$\{s_3\}$	$\{s_1\}$
s_2	$\{s_3\}$	$\{s_3\}$

Table 2.3: Transition table of the NBA N

A graphical representation of the automaton is provided by Figure 2.4. N *accepts the language*

$$L(N) = \{w \in \{a, b\}^\omega \mid w = x(b)^\omega \textit{ with } x \in \{a, b\}^*\}$$

which denotes the language of all $w \in \{a, b\}^\omega$ *having an infinite suffix of* b*'s only. The automaton "nondeterministically guesses" the first* b *of this suffix.*

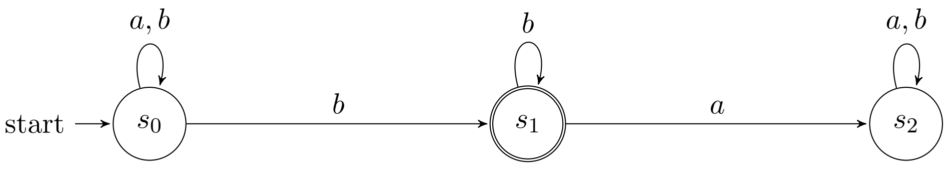

Figure 2.4: A graphical representation of the NBA N

An accepting run for the word $aba(b)^\omega \in L(N)$ *would be*

$$\rho = (s_0, 0), (s_0, 1), (s_0, 2), (s_1, 3), (s_1, 4), (s_1, 5), \ldots$$

which is accepting since $\inf(\rho) \cap F = \{s_1\} \neq \emptyset$*.*

We refer to the set of languages recognized by NBAs as *ω-regular languages.* In the previous section, we mentioned DFAs to accept the same set of languages as NFAs. For Büchi automata, this equality of expressive power of determinism and nondeterminism does not hold. In fact, there exists no *deterministic Büchi automaton* (DBA) recognizing the same language as the NBA shown in Example 2.3. Since the set of all DBAs is a strict subset of the set of all NBAs, it follows, that NBAs have to be strictly more powerful than DBAs. We won't be using any DBAs in this thesis, so we go without a formal definition of them. Additional information on this subject can be found in [10].

Definition 2.12 (2-way nondeterministic Büchi automaton [11]). *A* 2-way nondeterministic Büchi automaton *(2NBA) is a 5-tuple*

$$N = (\Sigma, S, s_0, \delta, F)$$

equally defined to a regular 2NFA.

A run of a 2NBA on an infinite word $w = w_0w_1...$ is an infinite sequence of state-index pairs $\rho = (t_0, i_0), (t_1, i_1), \ldots$ with:

- $t_0 = s_0$
- $i_0 = 0$
- $t_j \in S$ *for* $j \in \mathbb{N}_0$
- $i_j \in \mathbb{N}_0$ *for* $j \in \mathbb{N}_0$
- $(t_{j+1}, i_{j+1} - i_j) \in \delta(t_j, w_{i_j})$ *for* $j \in \mathbb{N}_0$

We say a run to be accepting if and only if $\inf(\rho) \cap F \neq \emptyset$. *We define the language accepted by a 2NBA N as follows:*

$$L(N) = \{w \in \Sigma^\omega \mid \textit{There exists an accepting run of } N \textit{ over } w\}$$

Example 2.4 (2NBA and accepting run). *Consider the following 2-way nondeterministic Büchi automaton $N = (\Sigma, S, s_0, \delta, F)$ with $\Sigma = \{a, b\}$, $S = \{s_0, s_1, s_2, s_3, s_4, s_5\}$, $F = \{s_2\}$, and δ given by the transition table 2.4.*

	a	b
s_0	$\{(s_0, 1)\}$	$\{(s_0, 1), (s_1, -1)\}$
s_1	$\{(s_5, 1)\}$	$\{(s_2, -1)\}$
s_2	$\{(s_5, 1)\}$	$\{(s_3, 1)\}$
s_3	$\{(s_4, 1)\}$	$\{(s_4, 1)\}$
s_4	$\{(s_0, 1)\}$	$\{(s_0, 1)\}$
s_5	$\{(s_5, 1)\}$	$\{(s_5, 1)\}$

Table 2.4: Transition table of the 2NBA N

A graphical representation of the automaton is provided by Figure 2.5. N accepts the language

$$L(N) = \{w \in \{a,b\}^\omega \mid |\{i \in \mathbb{N}_0 | w_i w_{i+1} w_{i+2} = bbb\}| = \infty\}$$

which denotes the language of all $w \in \{a,b\}^\omega$ containing the substring bbb infinitely often. The automaton "nondeterministically guesses" the first b of such an infix.

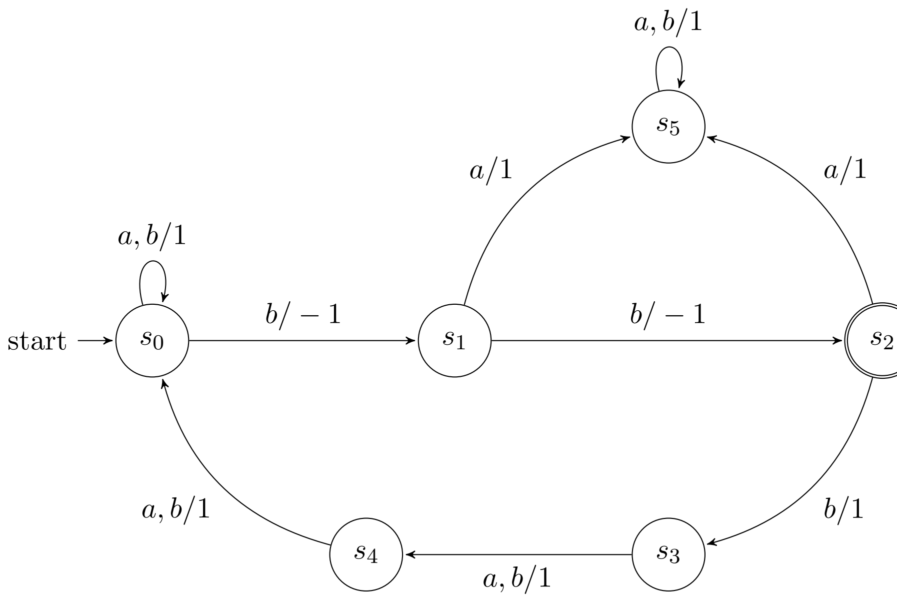

FIGURE 2.5: A graphical representation of the 2NBA N

An accepting run for the word $(abbba)^\omega \in L(N)$ would be

$$\rho = (s_0, 0), (s_0, 1), (s_0, 2), (s_0, 3), (s_1, 2), (s_2, 1), (s_3, 2), (s_4, 3), (s_0, 4), \ldots$$

and the denoted subsequence repeats itself now an infinite number of times with an additional offset of 5 *per repetition on the word positions. This run is accepting since* $\inf(\rho) \cap F = \{s_2\} \neq \emptyset$.

Definition 2.13 (Simple run of a 2NBA)**.** *Let $\rho = (t_0, i_0), (t_1, i_1), \ldots$ be a run of a 2NBA. We say ρ to be a* simple run *if and only if one of the following properties holds:*

- *For all $j \in \mathbb{N}_0, k \in \mathbb{N}$ with $j < k$, either $t_j \neq t_k$ or $i_j \neq i_k$.*
- *There exist $l, m \in \mathbb{N}$ such that for all $h \in \mathbb{N}_0, p \in \mathbb{N}$ with $h < p < l + m$, either $t_h \neq t_p$ or $i_h \neq i_p$, and for all $f \in \mathbb{N}$ with $f \geq l$, $t_f = t_{f+m}$ and $i_f = i_{f+m}$.*

The first property describes a run without any loops, while second property describes a run with exactly one loop that will never be left during the run and that contains no smaller loops within itself.

Theorem 2.2. *A 2NBA N accepts a word w iff N accepts w with a simple run.*

Proof.[11] Given an accepting non-simple run ρ of a 2NBA N over a word w, we will construct an accepting simple run ρ' over w from ρ. Since accepting states can be visited in the loops, we are not allowed to delete them completely like we did in the 2NFA case. Nevertheless, if there is a "non-accepting loop" in ρ (one which does not visit any state $t \in F$), we can delete this loop with the same justification as in the 2NFA case.

We will now divide ρ into segments. Going through the run, we start a new segment at every visit of an accepting state. Since ρ is accepting, there have to be infinitely many visits of accepting states and therefore infinitely many segments. The non-accepting loops can only be found inside these segments and every segment is of finite length, so every segment can be made loop free within a finite number of modifications. Afterwards, we can safely assume that if there exists some $j \in \mathbb{N}_0, k \in \mathbb{N}$ with $j < k$ such that $t_j = t_k$ and $i_j = i_k$, there has to be some accepting state visited between step j and step k. Out of all these j and k, we choose the minimal j and the minimal corresponding k. Now we construct the sequence $\rho' = (s_0, 0), \ldots, (t_{j-1}, i_{j-1}), ((t_j, i_j), \ldots, (t_{k-1}, i_{k-1}))^\omega$. Since we modified ρ only inside the segments and never changed the first or the last element of such a segment, ρ' must still be a valid run of N over w. Visiting the acceptance set infinitely often between t_j and t_{k-1} makes ρ' accepting. And since we deleted all loops except one, ρ' must be a simple run. □

We can interpret the ρ' constructed by this proof as going through ρ and staying inside the first occurring loop visiting an accepting state. So now we know, that whenever a word w is accepted by a 2NBA, w can be accepted by a simple run.

Definition 2.14 (Alternating Büchi automaton [11])**.** *An* alternating Büchi automaton *(ABA) is a 5-tuple*

$$\mathcal{A} = (\Sigma, Q, q_0, \eta, F)$$

equally defined to a regular AFA.

A run of an ABA on an infinite word $w = w_0 w_1 ...$ is a possibly infinite Q-labeled tree $\rho = (T, r)$ with $T = (V, E)$, $r : V \to Q$, $r(v_r) = q_0$ and

$$\begin{aligned}&\forall v \in V \text{ with } r(v) = q \text{ and } \eta(q, w_{d(v)}) = \varphi :\\ &\qquad \exists Q' \subseteq Q : Q' \models \varphi, deg^+(v) = |Q'| \text{ and } \forall q' \in Q' :\\ &\qquad\qquad \exists!(v, w) \in E : r(w) = q'\end{aligned}$$

We say a run to be an accepting run if and only if all infinite paths in the run tree starting from the root visit the accepting set F infinitely often and for all finite paths there is a leaf vertex v with $\eta(v, w_{d(v)}) =$ **true**. *We define the language accepted by an ABA $\mathcal{A}$ as follows:*

$$L(\mathcal{A}) = \{w \in \Sigma^\omega \mid \text{There exists an accepting run of } \mathcal{A} \text{ over } w\}$$

Example 2.5 (ABA and accepting run). *Consider the following alternating Büchi automaton $\mathcal{A} = (\Sigma, Q, q_0, \eta, F)$ with $\Sigma = \{a, b\}$, $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $F = \{q_1, q_3, q_5\}$, and η given by the transition table 2.5.*

	a	b
q_0	$(q_1 \wedge q_3) \vee q_5$	q_6
q_1	q_1	q_2
q_2	q_1	q_2
q_3	q_4	q_3
q_4	q_4	q_3
q_5	q_5	q_6
q_6	q_6	q_6

TABLE 2.5: Transition table of the ABA $\mathcal{A}$

A graphical representation of the automaton is provided by Figure 2.6. $\mathcal{A}$ accepts the language

$$\begin{aligned}L(A) = \{w \in \{a, b\}^\omega \mid w = (a)^\omega \text{ or}&\\ (w_0 = a \text{ and } |\{i \in \mathbb{N}_0 | w_i = a\}| = |\{i \in \mathbb{N}_0 | w_i = b\}| = \infty)\}&\end{aligned}$$

which denotes the language of all $w \in \{a, b\}^*$ which are either the word only containing a's or which start with an a and do not have an infinite suffix of either a's or b's only. The automaton "nondeterministically decides" at reading the first a (if there is one) for which of these cases it's going to check.

Figure 2.7 shows a run ρ over the word $(aab)^\omega \in L(\mathcal{A})$. The upper branch visits infinitely many vertices v with $r(v) = q_1 \in F$, while the lower branch visits infinitely many vertices w with $r(w) = q_3 \in F$. Therefore, this run is accepting.

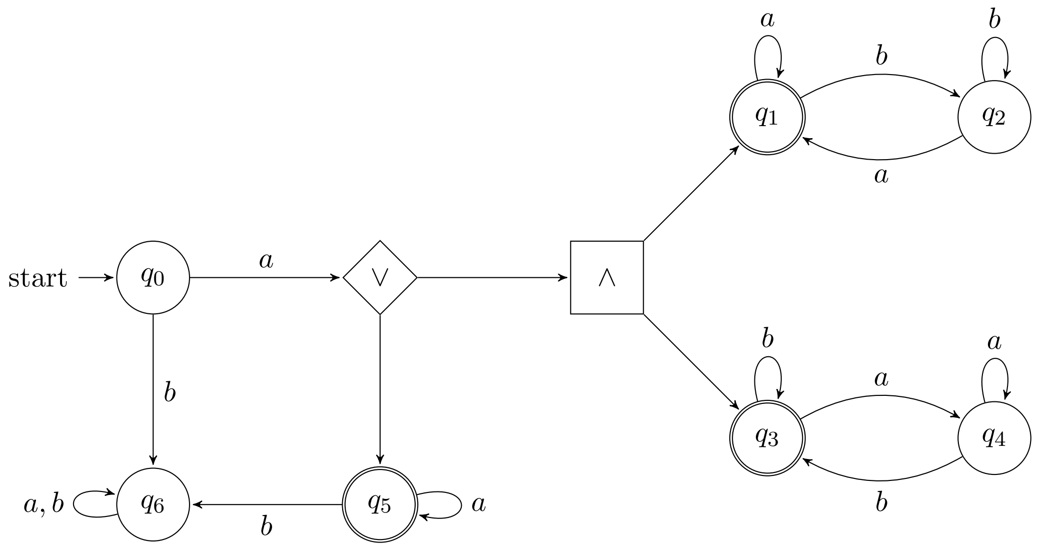

FIGURE 2.6: A graphical representation of the ABA $\mathcal{A}$

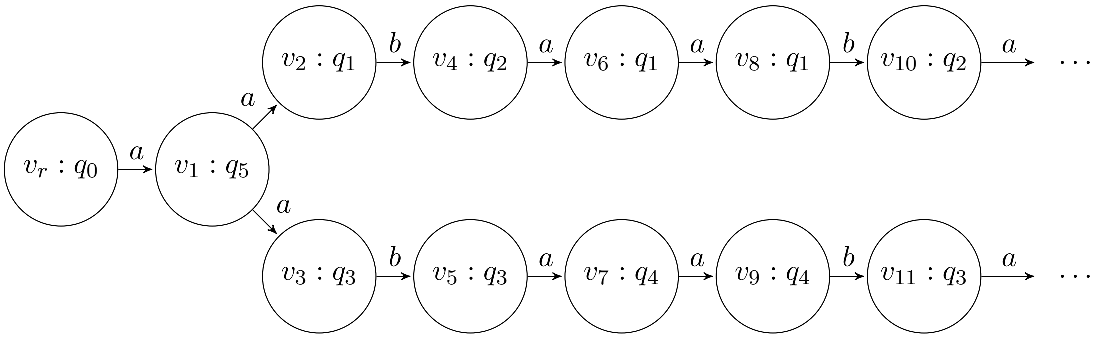

FIGURE 2.7: An accepting run of the ABA $\mathcal{A}$ over the word $(aab)^\omega$

When we introduced AFAs in the previous section, we pointed towards the possibility of creating an AFA $\widetilde{\mathcal{A}}$ from an AFA $\mathcal{A}$ with $L(\widetilde{\mathcal{A}}) = \Sigma^* \setminus L(\mathcal{A})$. This was done by dualization of $\mathcal{A}$. We are able to dualize an ABA as well, but unfortunately, a dualized ABA does not generally accept the complement language of the original ABA. But with a modification of our notion of a dualized ABA, we can indeed achieve an automaton accepting the complement language of an ABA. Though there are other complementation methods for ABAs, this is a particularly simple one.

Definition 2.15 (Alternating co-Büchi automaton [11]). *An* alternating co-Büchi automaton *(ACA) is a 5-tuple*

$$\mathcal{A} = (\Sigma, Q, q_0, \eta, F)$$

equally defined to an ABA.

A run of an ACA over an infinite word is the same as a run of an ABA. We say a run of an ACA to be accepting if and only if every infinite path in the run tree only contains a finite number of vertices v with $r(v) \in F$.

Opposite to the acceptance of ABAs, where we need to visit the accepting set infinitely often on every infinite path, in an ACA we demand to visit the accepting set only finitely often.

Using this definition, we can complement an ABA $\mathcal{A}$ by simply dualizing its transition function and interpret $\widetilde{\mathcal{A}}$ as an ACA. Then we get $L(\widetilde{\mathcal{A}}) = \Sigma^\omega \setminus L(\mathcal{A})$. In *Chapter 3*, we will discuss a method for the complementation of an ABA which delivers an ABA instead of an ACA accepting the complement language.

Definition 2.16 (Weak alternating automaton [6]). *A* weak alternating automaton *(WAA) is a 5-tuple*

$$\mathcal{A} = (\Sigma, Q, q_0, \eta, F)$$

where Σ, q_0, η and F are defined just like in an ABA and

- *$Q = \bigcup_{i=0}^{n-1} Q_i$ consists of $n \in \mathbb{N}$ pairwise disjoint sets of states Q_i, which are either a subset of F ($Q_i \subseteq F$, we call Q_i an* accepting set*) or are completely disjoint from F ($Q_i \cap F = \emptyset$, we call Q_i a* rejecting set*)*
- *There exists a partial order $\leq$ of the Q_i, such that for every $q \in Q_j$ and $q' \in Q_k$ with $0 \leq j, k < n$ for which q' occurs in $\eta(q, a)$ for some $a \in \Sigma$, we have $Q_k \leq Q_j$.*

We define a run of a WAA just the same as a run of an ABA. We say a run to be accepting, if and only if the final Q_i is an accepting set.

Because of the partial ordering, a run has to finally arrive in some Q_i which it isn't going to leave.

The last class of automata we want to talk about is the class of *alternating k-parity automata* (AkPA). Since we won't need them in the automata conversions described in this thesis, we will go without a formal definition. An alternating k-parity automaton is a generalization of an alternating Büchi automaton. While the state set of an ABA can be partitioned into two disjoint sets F and $Q \setminus F$, an AkPAs state set is partitioned into k disjoint sets Q_i, each of them being of a different *parity* i with $i \in \{1, \ldots, k\}$. Let $p : Q \to \mathbb{N}_0$ denote the function which assigns each state $q \in Q$ to the parity of the set Q_i it's included in. A *run of an alternating k-parity automaton ρ* is defined just as the run of an ABA and is said to be *accepting* if and only if the highest parity met infinitely often is even. We can formulate this acceptance condition as follows:

$$\rho \text{ is accepting} \quad \Leftrightarrow \quad \max(p(\inf(\rho))) \bmod 2 = 0$$

The only reason we are discussing AkPAs right now, is that while the automata conversion methods we will discuss in this thesis are working on ABAs, the framework in which we will implement them works on AkPAs. We want to remark that this is no problem, since we can interpret Büchi automata as 2-parity automata. Both automata have two disjoint state sets and we can describe the accepting set F of an ABA as the state set of parity 2 of an A2PA, which will "behave like an accepting set" since the only other parity is 1, which is an odd parity and therefore not accepting.

At the end of this section, we want to mention that all classes of automata over infinite words we introduced are of equal expressiveness (except the DBA, as we stated before). They all accept the ω-regular languages, which are known to be closed under union, intersection and complementation [10]. Especially the last property will be of importance for this thesis, since we are discussing complementation methods for ABAs and 2NBAs in the following chapter.

Chapter 3

Automata Conversions

Finite state automata are useful tools in many areas of modern work life and research. Depending on the circumstances providing the context for the usage of FSAs, different kinds of automata may serve more or less straightforward ways to model the situation. Unfortunately, the automaton offering the most intuitive representation is not always the most practical tool in terms of processability. We are therefore interested in methods converting certain types of FSAs into other types of FSAs without changing the original semantics.

In this chapter, we will consider two different automata conversions. In the first section, we examine the construction of alternating Büchi automata from 2-way nondeterministic Büchi automata. In [13], Sánchez and Leucker introduced regular expressions with past operators and showed how to model these using 2NFAs. We want to transfer this idea to the context of infinite words and use 2NBAs as a representation of ω-regular expressions with past operators. Such a modeling can be useful in runtime verification, but since 2NBAs are hard to process, we want to convert them to their 1-way equivalents. Miyano and Hayashi proposed a method for the conversion of ABAs to NBAs in [9], so the construction of an ABA from a 2NBA closes the gap between 2NBA and NBA. More on the motivation for this construction will be discussed in *Chapter 4*, where we will describe an implementation of this conversion in the context of the logic and automata library *RltlConv*.

The second section describes a method for achieving a weak alternating automaton accepting the complement language of a provided alternating Büchi automaton. We will do this by dualizing the ABA to receive an alternating co-Büchi automaton and construct a WAA simulating accepting runs of this ACA.

Afterwards, we discuss the composition of the 2NBA to ABA conversion and the ABA complementation method to achieve a construction capable of complementing a 2NBA.

3.1 From 2-way to Alternation

In this section, we want to describe a method for the conversion of a 2-way nondeterministic Büchi automaton to an alternating Büchi automaton. To do so, we will first take a detour and discuss the conversion of a 2NFA to an AFA. Afterwards, we extend the notion of this method to match the context of Büchi automata and infinite words.

Both conversions were proposed by Piterman and Vardi in [11]. We will reproduce their results, correct some minor formal flaws and provide extended explanations along with illustrative examples for a better understanding of the depicted ideas. We will not discuss the correctness proofs of the described automata conversions, but the interested reader can find these in [11].

3.1.1 2NFA to AFA

Given a 2NFA N with n states, we will construct an AFA $\mathcal{A}$ with $L(N) = L(\mathcal{A})$ and $\mathcal{O}(n^2)$ states. The construction method we describe has two phases. In the first phase, we will build a 2NFA N' with $L(N') = L(N)$ which does not contain any ε-moves. The second phase will be the conversion of N' to a language equivalent AFA $\mathcal{A}$.

Elimination of ε-moves

We recall that for the given 2NFA $N = (\Sigma, S, s_0, \delta, F)$ the transition function δ is defined as $\delta : S \times \Sigma \to 2^{S\times\{-1,0,1\}}$. What we want to achieve is a function $\delta' : S \times \Sigma \to 2^{S\times\{-1,1\}}$ such that the 2NFA $N' = (\Sigma, S, s_0, \delta', F)$ accepts the same language as N.

The basic idea for the construction of such a function δ' is the following: We take an $s \in S$ and an $a \in \Sigma$. Let's assume that $\delta(s, a)$ includes some tuple $(s', 0)$. This would represent an ε-move, since the 2NFA would change from state s to state s' without changing the symbol a currently read. Let's further assume, that there exists some $(t, \Delta) \in \delta(s', a)$ with $\Delta \in \{-1, 1\}$, representing either a forward or a backward move of the automaton while changing from state s' to state t. Speaking in terms of a run of N over some word w, there would be the possibility of a subsequence $(s, i), (s', i), (t, i + \Delta)$ with $w_i = a$. But since the "detour" over s' doesn't change the symbol N is reading, we can skip this ε-move by defining a new transition function which leads from (s, i) directly to $(t, i + \Delta)$. Furthermore, we can define a new transition function δ', which skips all possible sequences of ε-moves in a run of N.

To do this, we define a set C_a^s for every $s \in S$ and for every $a \in \Sigma$. C_a^s includes all forward and backward moves N can make reading a after running through a finite (and possibly empty) sequence of ε-moves starting in state s.

Formally, we define C_a^s as follows:

$$C_a^s = \left\{ (t, \Delta) \in S \times \{-1, 1\} \;\middle|\; \begin{array}{l} \exists\ t_0 t_1 ... t_k \in S^+ \text{ with } k \in \mathbb{N}_0 \text{ such that } t_0 = s, \\ (t_j, 0) \in \delta(t_{j-1}, a) \text{ for all } j \text{ with } 1 \leq j \leq k, \\ \text{and } (t, \Delta) \in \delta(t_k, a) \end{array} \right\}$$

We define $\delta'(s, a) = C_a^s$ and get the 2NFA N' without ε-moves as described above.

Anatomy of 2-way runs without ε-moves

We will now analyse the structure of a run a 2NFA $N = (\Sigma, S, s_0, \delta, F)$ without ε-moves. Recall that a run of N over a finite word w is a sequence $\rho = (s_0, 0), (s_1, i_1), \ldots, (s_m, i_m)$ with $s_j \in S$ for $0 \leq j \leq m$. Since N has no ε-moves we further get $i_{j+1} = i_j + \Delta$ for $0 \leq j < m$ and $\Delta \in \{-1, 1\}$. We say a state-index pair (s_j, i_j) in ρ to be a *forward tuple* if it results from a previous forward step and we call it a *backward tuple* if it results from a previous backward step. Formally, this means (s_j, i_j) is a forward tuple if $i_j = i_{j-1} + 1$ and a backward tuple if $i_j = i_{j-1} - 1$. We define $(s_0, 0)$ to be a forward tuple.

Since ρ is moving over the word w with every transition taken, we can visualize the run over time as something Piterman and Vardi call "zigzags". We can see an example of a zigzag in Figure 3.1. The corresponding run is a sequence starting with $(s_0, 0), (s_1, 1), (s_2, 2), (s_3, 1), (s_4, 2), (s_5, 3), \ldots$ and going on from there.

Since the AFA we want to achieve is a 1-way automaton, it will have to somehow "nondeterministically guess" how the zigzag is going to evolve in time. The main idea of the construction we're going to describe is the following: As soon as the AFA reads some w_i, it guesses all future visits of the zigzag run of w_i. One of these visits will be the last occurrence of w_i in the zigzag, so the AFA will continue processing w from the according state. Additionally, it will spawn a new process for each of the future visits, which will be verifying that the AFA does indeed reach all of these visits when processing w further.

Let's look at the example in Figure 3.1 again. There are two states in this example in which w_1 is read, namely s_1 and s_3. Since $(s_2, 1) \in \delta(s_1, w_1)$ and $(s_4, 1) \in \delta(s_3, w_1)$, we will have to make sure in the 1-way run of the AFA that s_3 resulted from s_2 and that the

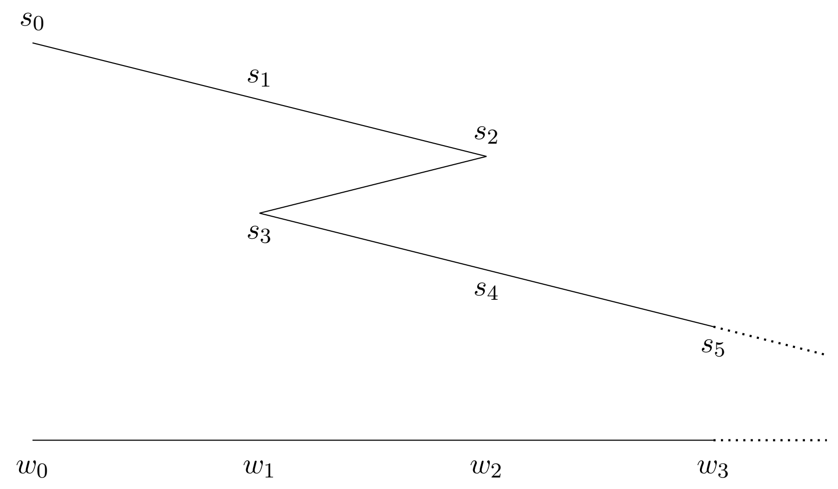

FIGURE 3.1: Example of a 2NFA zigzag run

run continuing from s_3 to s_4 and further is accepting. Therefore, when the AFA reads w_1, it spawns two processes: One is going to verify that s_1 leads to s_3 and the other one is going to check if the run going onward from s_3 is accepting.

Based on this idea, we can partition the states of the AFA into two different types. We say a state $s \in S$ to be a *singleton state*. These states are representing the part of the run which is going forward and for which we have to verify that it is going to accept. There exists exactly one singleton state s' for each index $i \in \mathbb{N}_0$ of w in a run of the AFA, which corresponds to the last forward tuple (s', i) in the zigzag run the AFA is simulating. A *pair state* $(s, t) \in S \times S$ is a pair of two states s and t for which (s, i) is a forward tuple and (t, i) is a backward tuple for some index i in the simulated 2NFA run. This represents the part of the run which ensures the state s is leading to the state t. Since the set of states of the AFA includes singleton states as well as pair states, we define $Q = S \cup (S \times S)$, therefore the blowup in the number of states used in the AFA in comparison to the number of states in the 2NFA is a quadratic one.

Using this definition, we can describe an AFA run simulating the zigzag run shown in Figure 3.1. The singleton states following from the example are s_0, s_1, s_4 and s_5. The only pair state is (s_2, s_3). The AFA run we want to use to simulate the zigzag run is presented in Figure 3.2.

Now that we have defined the set of states and introduced the concepts of singleton states and pair states, we will go and describe the transition function η of the AFA we want to construct. We will do this separately for the transitions at singleton states and the transitions at pair states.

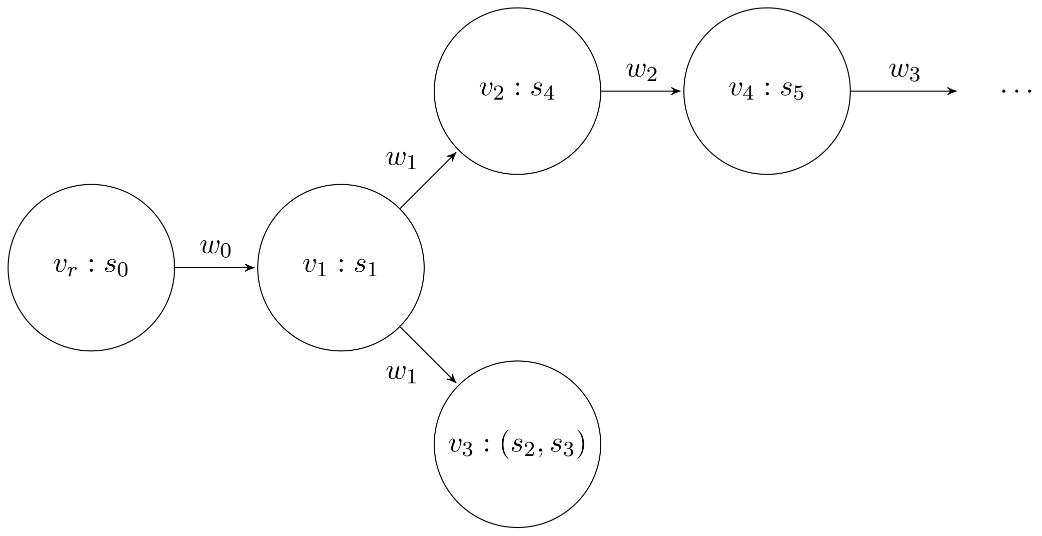

FIGURE 3.2: AFA simulation of the 2NFA zigzag run presented in Figure 3.1

Transitions at singleton states

We define the transition function η for singleton states. When the AFA is in a singleton state $t \in Q$ and reads the symbol w_i, it will guess all the states of the 2NFA N that will be reading w_i in the simulated zigzag run and which are part of a backward tuple that is followed by a forward tuple. Since we showed in the former chapter that every word accepted by a 2NFA can be accepted by a simple run, the number of states reading w_i in the zigzag run is bounded by the number of states in the N. Let's say there will be k other states reading w_i apart from t itself. We refer to these states according to the order in which they will appear during the zigzag run as $s_1, s_2, \dots, s_k$. We further denote their successors in the zigzag run by $t_1, t_2, \dots, t_k$. We refer to the successor of t itself by t_0. We can see an example for $k = 3$ in Figure 3.3. The AFA is supposed to ensure that t_j will lead to s_{j+1} for $0 \leq j < k$. t_k will be the next singleton state, from which on the automaton will read the rest of the word w and will eventually reach an accepting state.

We will now associate a set R^t_a with every pair of a state t and a symbol of the alphabet a. R^t_a includes all possible state sequences which are of length at most $2n - 1$ which do not contain equal states at two even positions or at two odd positions. The states at the even positions are part of forward tuples and will be denoted by t_j, while the states at odd positions are part of backward tuples and will be denoted by s_j. We also want the first state in the sequence t_0 to be a successor of t when reading a, so $(t_0, 1) \in \delta(t, a)$. We demand the same for all pairs s_j and t_j, so $(t_j, 1) \in \delta(s_j, a)$ must hold for $1 \leq j \leq k$.

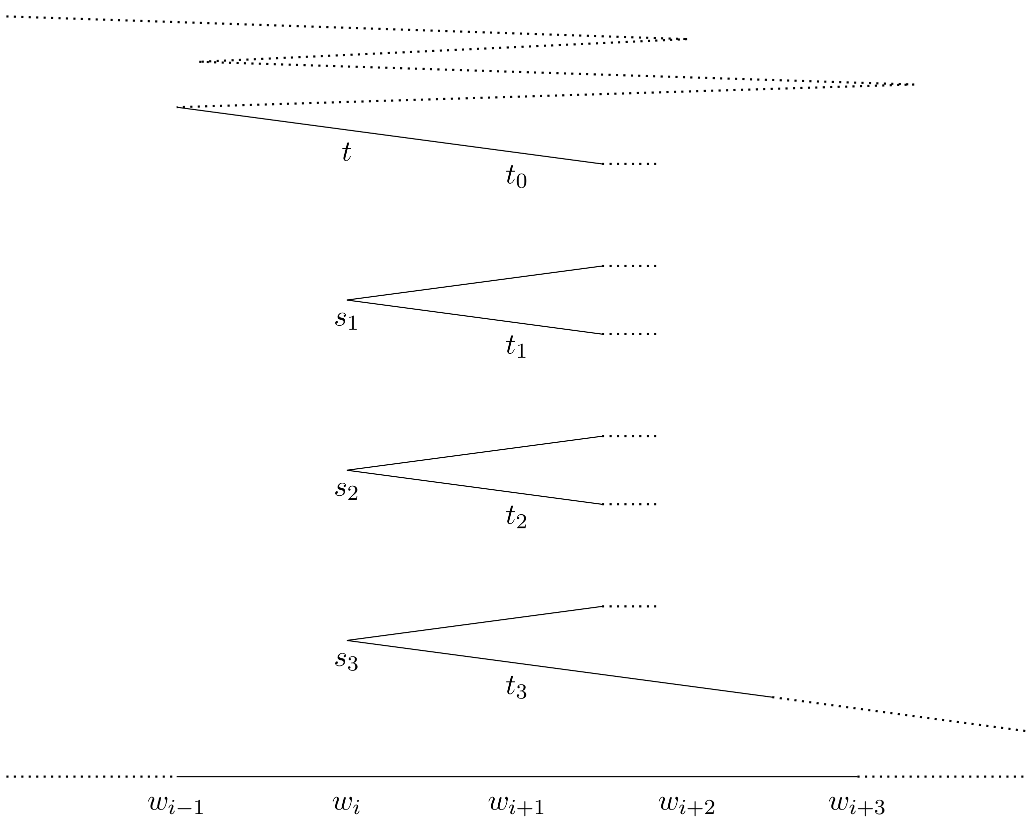

FIGURE 3.3: Example of an AFA transition at the singleton state t

Formally, we write the following:

$$R_a^t = \left\{ \langle t_0, s_1, t_1, \ldots, s_k, t_k \rangle \;\middle|\; \begin{array}{l} 0 \leq k < n \\ (t_0, 1) \in \delta(t, a) \\ \forall i < j, s_i \neq s_j \text{ and } t_i \neq t_j \\ \forall j, (t_j, 1) \in \delta(s_j, a) \end{array} \right\}$$

Now we can define the transition function η of the AFA for all singleton states t and all symbols a as follows:

$$\eta(t, a) = \bigvee_{\langle t_0, s_1, t_1, \ldots, s_k, t_k \rangle \in R_a^t} (t_0, s_1) \wedge (t_1, s_2) \wedge \ldots \wedge (t_{k-1}, s_k) \wedge t_k$$

The transition function chooses one of the sequences from R_a^t nondeterministically. The included pair states will have to ensure that all predictions about the future of the zigzag run are justified. Since the AFA has to process w in one linear sweep, a pair state (t, s) reading w_i can only use the suffix $w_i w_{i+1} ... w_l$ to verify that s follows t. But this is no problem, because the described case of singleton state transitions ensures, that a zigzag from s to t will never be visiting any indices smaller than i (otherwise, our construction would have done a different split of the zigzag run and wouldn't have spawned (t, s) in the first place).

Transitions at pair states

Now that we have defined η for singleton states, we will go on and discuss the transitions at pair states, which are quite similar. When the AFA is in a pair state $(t, s) \in Q$ and reads the symbol w_i, it will guess a segment of the zigzag run which connects t to s. As we reasoned before, it can only use the suffix $w_i w_{i+1} ... w_l$ to verify this. It does so by guessing all the states of the 2NFA N that will be reading w_i between the visits of t and s which are part of a backward tuple that is followed by a forward tuple. Again, we know that the number k of these states has to be bounded by the number of states of N. We enumerate these as $s_1, s_2, \ldots, s_k$ according to the order of their appearance in the zigzag run. We refer to the successors of these states by $t_1, t_2, \ldots t_k$, the successor of t is t_0. Different than in the case of singleton state transitions, the AFA has to also predict an additional state reading w_i which is part of a backward tuple followed by another backward tuple. The state of this following backward tuple has to be s, so the last state guessed has to be some state s_{k+1} with $(s, -1) \in \delta(s_{k+1}, w_i)$. An example for $k = 2$ can be seen in Figure 3.4.

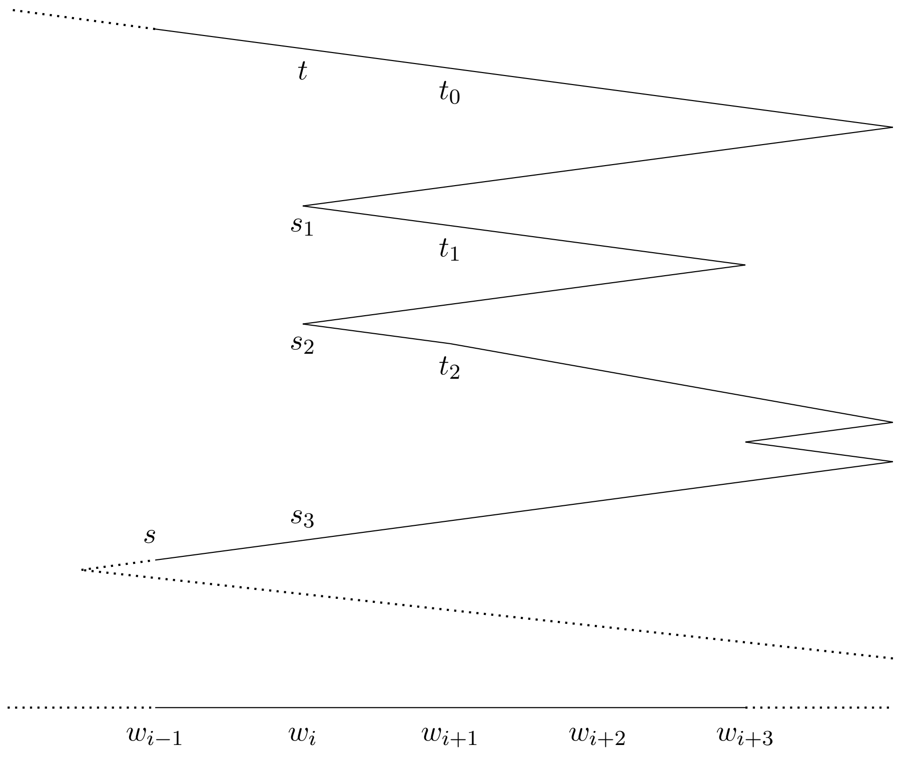

FIGURE 3.4: Example of an AFA transition at the pair state (t, s)

We construct a set $R_a^{(t,s)}$ for every pair state (t,s) and a symbol of the alphabet a. $R_a^{(t,s)}$ includes all possible state sequences which are of length at most $2n$ which do not contain equal states at two even positions or at two odd positions, just like the sets R_a^t we discussed before. We want the first state in the sequence t_0 to be a successor of t when reading a, so $(t_0,1) \in \delta(t,a)$. We demand the same for all pairs s_j and t_j, so $(t_j,1) \in \delta(s_j,a)$ must hold for $1 \le j \le k$. At last, we need $(s,-1) \in \delta(s_{k+1},a)$ to be true. Formally, we express this as follows:

$$R_a^{(t,s)} = \left\{ \langle t_0, s_1, t_1, \ldots, s_k, t_k, s_{k+1} \rangle \;\middle|\; \begin{array}{l} 0 \le k < n \\ (t_0,1) \in \delta(t,a) \\ (s,-1) \in \delta(s_{k+1},a) \\ \forall i < j, s_i \neq s_j \text{ and } t_i \neq t_j \\ \forall i, (t_i,1) \in \delta(s_i,a) \end{array} \right\}$$

We define the transition function η of the AFA for all pair states (t,s) and all symbols a as follows:

$$\eta((t,s),a) = \begin{cases} \textbf{true} & \text{If } (s,-1) \in \delta(t,a) \\ \bigvee\limits_{\langle t_0,s_1,\ldots,t_k,s_{k+1}\rangle \in R_a^{(t,s)}} (t_0,s_1) \wedge (t_1,s_2) \wedge \ldots \wedge (t_k,s_{k+1}) & \text{Otherwise} \end{cases}$$

The transition function checks if for a pair state (t,s) reading a the prediction $(s,-1) \in \delta(t,a)$ already holds. If so, the function evaluates to **true** and the predicted future has been verified for this branch of the AFA run. Otherwise, the predicted future is not yet verified, the AFA chooses one of the sequences from $R_a^{(t,s)}$ nondeterministically and continues the process on this branch.

Now that η is defined for both singleton and pair states, we will provide an example of an extended 2NFA zigzag run and the corresponding run tree of the constructed AFA.

Example 3.1 (2NFA zigzag run and corresponding AFA run tree)**.** *There are two accepting runs given in this example. Figure 3.5 shows a 2NFA zigzag run on an input word* w *of length* 9. *For reasons of simplicity the 2NFA holds one unique state* s_i *for every step* $0 \le i \le 30$ *of this run along with an additional accepting state* $s_{31} \in F$.

Figure 3.6 displays the resulting simulation of the zigzag run by an AFA constructed according to the method proposed in this subsection. Again, s_{31} *is an accepting (singleton) state, while* $\eta((s_{20},s_{21}),w_4), \eta((s_5,s_6),w_5), \eta((s_{11},s_{12}),w_7)$ *and* $\eta((s_{27},s_{28}),w_7)$ *are all defined as* **true**.

When the AFA is in state s_1 reading the symbol w_1, it guesses that there will be one more state reading w_1 in the future, namely s_{17}. It chooses s_2 to be the successor of s_1 which will have to ensure that there is some segment of the zigzag run connecting s_2 to s_{17}. Therefore a new branch in the run tree is spawned. The first vertex in this new branch is labeled by the pair state (s_2, s_{17}). The AFA further chooses s_{18} to be the successor of s_{17}. Since s_{18} is a singleton state, the corresponding branch of the AFA run tree will have end in some accepting state.

As we can see, s_{18} will indeed lead to the accepting state s_{31}. The branch started for (s_2, s_{18}) on the other hand will have to end in some leaves evaluating to **true**. *After splitting one more time while reading w_3, the branch ends in two leaves labeled by (s_5, s_6) and (s_{11}, s_{12}). Since $(s_6, -1) \in \delta(s_5, w_5)$ and $(s_{12}, -1) \in \delta(s_{11}, w_7)$ (as we can see in the zigzag run), we indeed have $\eta((s_5, s_6), w_5) = \eta((s_{11}, s_{12}), w_7) =$* **true**.

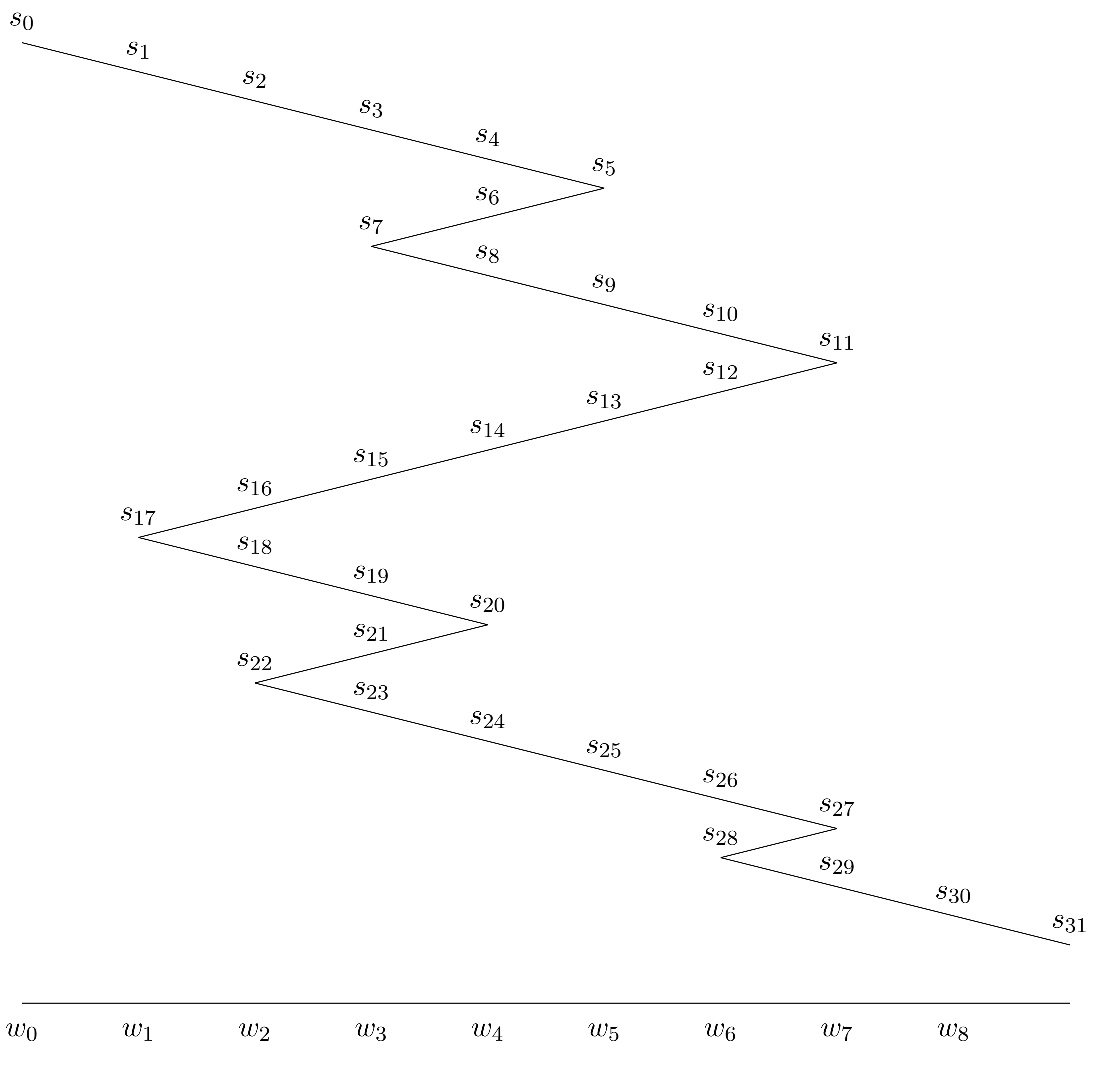

FIGURE 3.5: Extended example of a 2NFA zigzag run

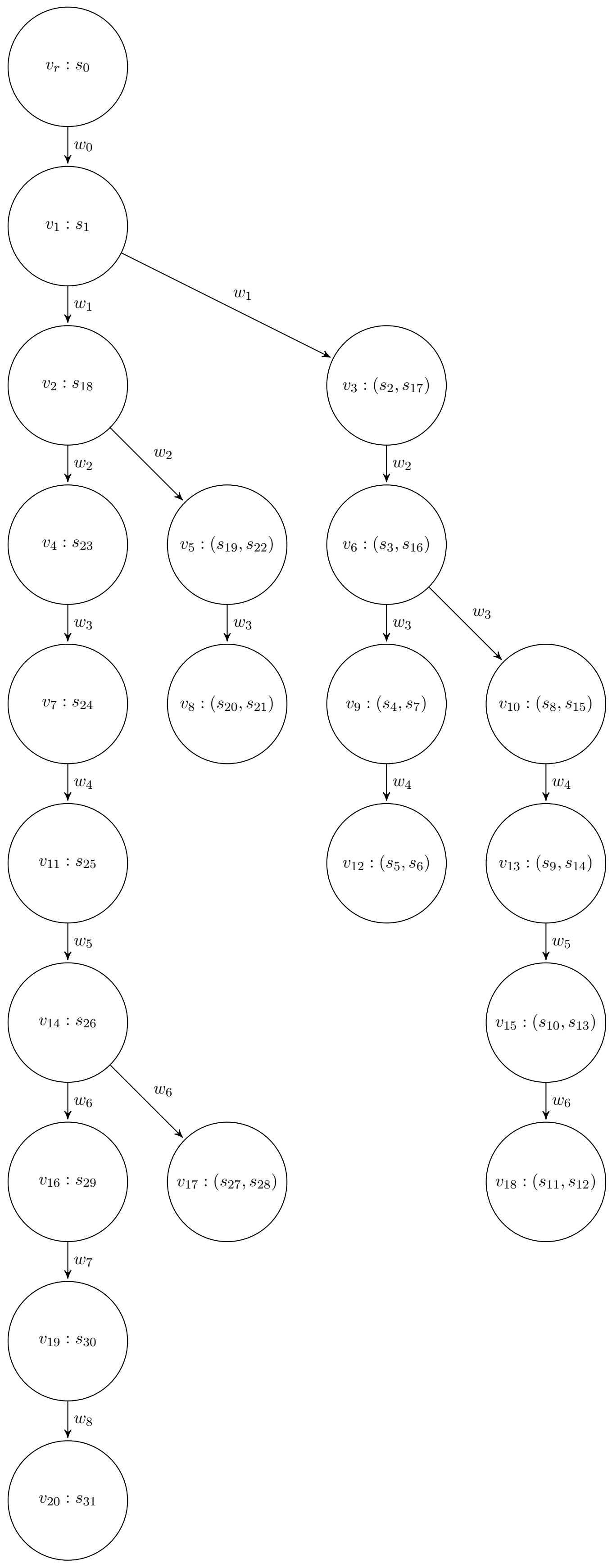

FIGURE 3.6: The corresponding AFA run tree to the zigzag run shown in Figure 3.5

With η completely defined, we have all the parts we need to define the final AFA. As already stated, the state set of the AFA is the union of all possible singleton and pair states $Q = S \cup (S \times S)$. We will neither change the start state s_0 nor will any modifications to the set of accepting states F be done. Using the new transition function η, we define the AFA as $\mathcal{A} = (\Sigma, S \cup (S \times S), s_0, \eta, F)$.

We recapitulate the achievements of this subsection: Starting from an arbitrary n-state 2NFA $N = (\Sigma, S, s_0, \delta, F)$, we first described the conversion of N to another 2NFA $N' = (\Sigma, S, s_0, \delta', F)$ which does not use any ε-moves and accepts the same language as N. From there on, we constructed an AFA $\mathcal{A} = (\Sigma, S \cup (S \times S), s_0, \eta, F)$ with $\mathcal{O}(n^2)$ states and $L(\mathcal{A}) = L(N') = L(N)$.

3.1.2 2NBA to ABA

In this subsection we will describe a method for the conversion of a 2NBA to an ABA accepting the same language. We could try to use the same method we used in the former subsection for the conversion of 2NFAs to AFAs, but this wouldn't be an adequate solution. Other than in the finite case, we have to take account of the two possibilities that an accepting run of an ABA must either contain an infinite branch visiting some accepting state infinitely often or ends in a loop visiting at least one accepting state. This can't be accomplished by the conversion described before, but with a few modifications we can achieve a proper construction to match the context of infinite words properly.

Given a 2NBA N with n states, we will construct an ABA $\mathcal{A}$ with $\mathcal{O}(n^2)$ states which accepts the same language as N, so $L(N) = L(\mathcal{A})$. Like we did in the finite case, we will remove all ε-moves from N prior to the construction of the alternating automaton.

Elimination of ε-moves

Given a 2NBA $N = (\Sigma, S, s_0, \delta, F)$ with $\delta : S \times \Sigma \to 2^{S \times \{-1,0,1\}}$ we want to construct another 2NBA $N' = (\Sigma, S', s'_0, \delta', F')$ with $\delta' : S' \times \Sigma \to 2^{S' \times \{-1,1\}}$ such that $L(N) = L(N')$ holds. In principal, we will use the same idea we already used in the context of finite words, but there are two problems to be considered. The first one is an ε-move visiting an accepting state and the second one is a loop of ε-moves visiting some $s \in F$.

We deal with these problems by expanding our state space. For every state $s \in S$, our new 2NBA N' will contain the two states $(s, \bot)$ and $(s, \top)$. In a run of N', we interpret the subsequence $\dots ((s, \bot), i), ((s', \top), i + \Delta), \dots$ for $i \in \mathbb{N}_0$ and $\Delta \in \{-1, 1\}$ as the appearance of an ε-move visiting an accepting state between (s, i) and $(s', i + \Delta)$ in the corresponding run of N. We will therefore define all states $(s, \top)$ for $s \in S$ as accepting states of N'. On the other hand, $\dots((s,\bot),i),((s',\bot),i+\Delta),\dots$ means that there hasn't been any (now eliminated) ε-move visiting F between (s,i) and $(s',i+\Delta)$.

Additionally, we introduce a single state Acc. This state represents an accepting "sink state", which will take care of all loops of ε-move which visit at least one accepting state. The state space of the new 2NBA is now described completely, so we can define it as $N' = (\Sigma, (S\times\{\bot,\top\})\cup\{Acc\}, (s_0,\bot), \delta', (F\times\{\bot\})\cup(S\times\{\top\})\cup\{Acc\})$.

We define a set NC_a^s for every $s\in S$ and for every $a\in\Sigma$. NC_a^s includes all forward and backward moves N can make when reading a after running through a finite (and possibly empty) sequence of ε-moves which started in s and did not visit any accepting states.

$$NC_a^s = \left\{ ((t,\bot),\Delta)\in((S\times\{\bot\})\times\{-1,1\}) \;\middle|\; \begin{array}{l} \exists(t_0,\dots,t_k)\in S^+ \text{ s.t. } t_0=s, \\ \forall 1\le j\le k, (t_j,0)\in\delta(t_{j-1},a), t_j\notin F \\ \text{and } (t,\Delta)\in\delta(t_k,a) \end{array} \right\}$$

Furthermore, we define another set AC_a^s with the same properties as NC_a^s except for the fact that at least one state appearing in the ε-move sequence has to be an accepting state.

$$AC_a^s = \left\{ ((t,\top),\Delta)\in((S\times\{\top\})\times\{-1,1\}) \;\middle|\; \begin{array}{l} \exists(t_0,\dots,t_k)\in S^+ \text{ s.t. } t_0=s, \\ \exists j>0 \text{ s.t. } t_j\in F, \\ \forall 1\le j\le k, (t_j,0)\in\delta(t_{j-1},a) \\ \text{and } (t,\Delta)\in\delta(t_k,a) \end{array} \right\}$$

To handle situations with loops of ε-moves visiting acceptig states, we introduce the Boolean variable $ACCEPT_a^s$ for every $s\in S$ and $a\in\Sigma$. If this variable is set to **true**, there exists such a loop starting and ending in s and taking a final forward or backward move reading a. Formally, we set $ACCEPT_a^s=\textbf{true}$ if and only if there exists a sequence of states $t_0t_1...t_k\in S^+$ that satisfies all the following conditions:

- $t_0=s$
- There exist $j,l\in\mathbb{N}_0$ such that $0\le j\le l\le k, (t_j,0)\in\delta(t_k,a)$ and $t_l\in F$.
- For all $j\in\mathbb{N}$ where $1\le j\le k$, we have $(t_j,0)\in\delta(t_{j-1},a)$.

Now we define the transition function δ' of the new 2NBA N' as follows:

$$\delta'((s,\bot),a) = \delta'((s,\top),a) = \begin{cases} \{(Acc,1)\} & \text{If } ACCEPT_a^s = \textbf{true} \\ NC_a^s \cup AC_a^s & \text{Otherwise} \end{cases}$$

$$\delta'(Acc,a) = \{(Acc,1)\}$$

We achieved an ε-move free 2NBA N' with $L(N) = L(N')$.

The ABA state set

We will now describe the construction of an ABA $\mathcal{A}$ from a provided ε-move free 2NBA $N = (\Sigma, S, s_0, \delta, F)$ such that $L(\mathcal{A}) = L(N)$ holds. The conversion is quite similar to the one we described for the context of finite words. We will stick to the idea of singleton and pair states, but we will enhance them using the symbols $\bot$ and $\top$ to express promises of visiting the acceptance set. Specifically, the state $(s,t,\top)$ with $s,t \in S$ means that the run segment leading from s to t has to visit at least one accepting state. The state $(s,\top)$ means that the zigzag segment connecting s to the previous singleton state has to visit some state from F at least once. Both states with $\bot$ instead of $\top$ do not hold these promises of visiting an accepting state.

We define the state set of $\mathcal{A}$ as $Q = (S \cup (S \times S)) \times \{\bot,\top\}$, which includes all combinations of singleton states and pair states paired with $\bot$ and $\top$. The initial state is $q_0 = (s_0, \bot)$.

Different than in the 2NFA to AFA conversion, we will allow the ABA to have transitions from a singleton state to a set of pair states (in the finite context, there always had to be some other singleton state). This will be necessary to handle the case of the ABA run ending in an infinite loop. One of the pair states visited in the loop will include the promise to visit an accepting state of N.

We define the acceptance set of $\mathcal{A}$ as $F' = (S \times \{\top\}) \cup (F \times \{\bot\})$. The transition function η will be described in the following two segments. Afterwards, the ABA will be defined as $\mathcal{A} = (\Sigma, Q, q_0, \eta, F') = (\Sigma, (S \cup (S \times S)) \times \{\bot,\top\}, (s_0,\bot), \eta, (S \times \{\top\}) \cup (F \times \{\bot\}))$.

Transitions at singleton states

We construct the same set R_a^t for every $t \in S$ and $a \in \Sigma$ as we did in the finite context.

$$R_a^t = \left\{ \langle t_0, s_1, t_1, \ldots, s_k, t_k \rangle \;\middle|\; \begin{array}{l} 0 \leq k < n \\ (t_0, 1) \in \delta(t, a) \\ \forall i < j, s_i \neq s_j \text{ and } t_i \neq t_j \\ \forall j, (t_j, 1) \in \delta(s_j, a) \end{array} \right\}$$

Again, we will interpret a sequence $\langle t_0, s_1, t_1, s_2, \ldots, t_{k-1}, s_k, t_k \rangle$ as a sequence of pair states $(t_0, s_1), (t_1, s_2), \ldots, (t_{k-1}, s_k)$ and as a singleton state t_k. All these pair states are segments of the zigzag run connecting t_0 to t_k. In a state $q \in Q$ of the ABA, t_k will be annotated either with $\bot$ or with $\top$. In the latter case, there has to be some pair state (t_j, s_{j+1}) with $0 \leq j < k$ in the sequence which is also annotated by $\top$, promising that there is some visit to an accepting state while going from t_j to s_{j+1}. In fact, there could be more than one pair state in the sequence promising to visit an accepting state, but since we talk about a finite sequence and therefore finitely many visits of F, we can safely say there's just one pair state annotated by $\top$ and all the other pair states are annotated by $\bot$.

For every $k \in \mathbb{N}_0$ with $0 \leq k < n$ we consider all possible sequences of $\bot$ and $\top$ of length $k+1$. We construct the set α_k^R to include all those sequences in which, if the last element is $\top$, than there has to be exactly one other element to be $\top$. Otherwise, all elements are $\bot$.

$$\alpha_k^R = \left\{ \langle \alpha_0, \ldots, \alpha_k \rangle \in \{\bot, \top\}^{k+1} \;\middle|\; \begin{array}{l} \text{If } \alpha_k = \top \text{ then } \exists! i \text{ s.t. } 0 \leq i < k \text{ and } \alpha_i = \top \\ \text{If } \alpha_k = \bot \text{ then } \forall\ 0 \leq i < k, \alpha_i = \bot \end{array} \right\}$$

Combining the R_a^t and the α_k^R we are able to deal with the case of singleton states connecting to another singleton state. What is still left to do is to take care of the case of the 2NFA run ending in an infinite loop. To guess such a loop, the ABA will have to predict two sequences of pair states. The first one describes the zigzag run leading from the singleton state to the first state in the loop. The second sequence describes the loop itself. We can see an example of such a situation in Figure 3.7. When in singleton state t, the ABA estimates the pair states $(t_0, s_1), (t_1, s_2)$ to lead to the start of the loop and the pair states $(t_2, s_3), (t_3, s_2)$ to describe the loop itself. This corresponds to the sequences $\langle t_0, s_1, t_1, s_2 \rangle$ and $\langle t_2, s_3, t_3, s_2 \rangle$. Both do not contain any state twice at odd or twice at even positions. Furthermore, both sequences end on the same state s_2.

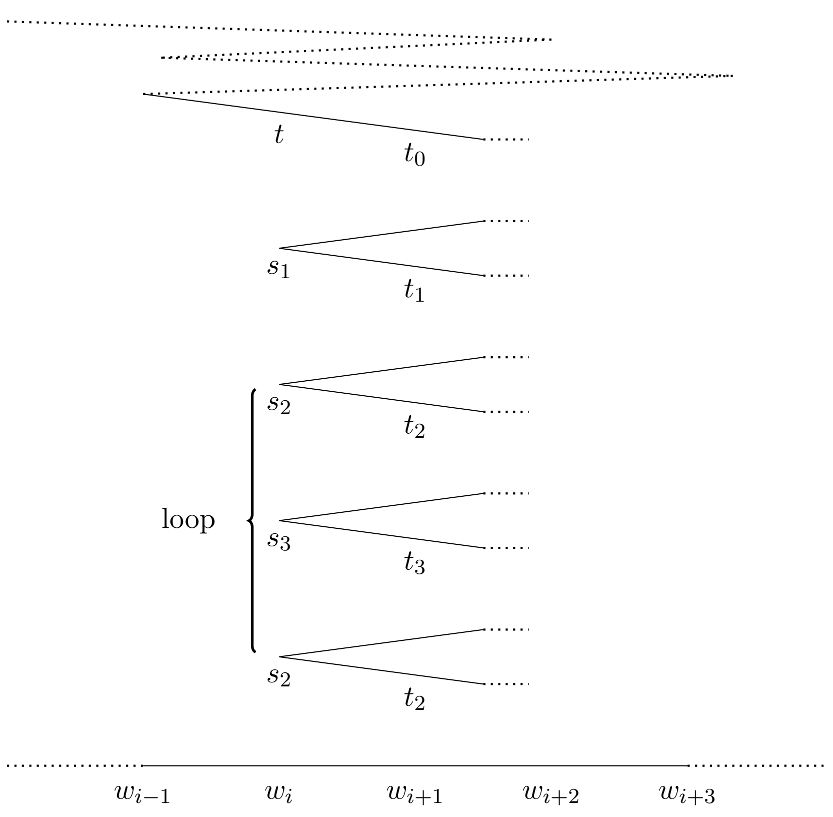

FIGURE 3.7: Example of an ABA transition at the singleton state t guessing an infinite loop starting in s_2 while reading w_i

Formally, we require the first state of the first sequence to be a valid successor of t when reading a symbol $a \in \Sigma$, so $(t_0^1, 1) \in \delta(t, a)$. The first state of the second sequence has to be a successor of the last state of the first sequence, so $(t_0^2, 1) \in \delta(s_{k+1}^1, a)$. Each sequence itself must define a valid sequence of pair states, so for every s_i^p with $p \in \{1, 2\}$, t_i^p has to be a successor, so we get $(t_i^p, 1) \in \delta(s_i^p, a)$. At last, we demand the last states of both sequences to be equal, so $s_{k+1}^1 = s_{k+1}^2$ has to hold. We denote the set holding all the sequences satisfying these conditions for some $t \in S$ and $a \in \Sigma$ by L_a^t.

$$L_a^t = \left\{ \left\langle \begin{array}{c} \langle t_0^1, s_1^1, t_1^1, \ldots, s_k^1, t_k^1, s_{k+1}^1 \rangle, \\ \langle t_0^2, s_1^2, t_1^2, \ldots, s_l^2, t_l^2, s_{l+1}^2 \rangle \end{array} \right\rangle \;\middle|\; \begin{array}{l} 0 \le k < n, 0 \le l < n \\ (t_0^1, 1) \in \delta(t, a), (t_0^2, 1) \in \delta(s_{k+1}^1, a) \\ \forall i < j, s_i^1 \ne s_j^1 \text{ and } t_i^1 \ne t_j^1 \\ \forall i < j, s_i^2 \ne s_j^2 \text{ and } t_i^2 \ne t_j^2 \\ \forall i, \forall p, (t_i^p, 1) \in \delta(s_i^p, a) \\ s_{k+1}^1 = s_{l+1}^2 \end{array} \right\}$$

One of the states in the second sequence always has to be an accepting state, otherwise there would be no visit to F in the infinite loop and the ABA run could not be accepting. Like we argued for the α_k^R, one visit to F is sufficient, so we define our sequences of $\perp$

and $\top$ as follows:

$$\alpha_l^L = \{\langle \alpha_0, \ldots, \alpha_l \rangle \in \{\bot, \top\}^{l+1} \mid \exists ! i \text{ s.t. } \alpha_i = \top\}$$

Now we got all the definitions we need to describe the transition function η for singleton states. When the ABA reaches a singleton state $(t, \bot)$ or $(t, \top)$ and reads a symbol $a \in \Sigma$, it will nondeterministically guess whether it will reach another singleton state or if it will enter a loop. After doing so, it will guess a sequence of states (or two sequences of states, depending on the former guess) and a sequence of $\bot$ and $\top$. Formally, the transition function for singleton states is defined as follows:

$$\eta((t,\bot),a) = \eta((t,\top),a) = \begin{array}{c} \left(\bigvee_{R_\alpha^t, \alpha_k^R} (t_0, s_1, \alpha_0) \wedge \ldots \wedge (t_{k-1}, s_k, \alpha_{k-1}) \wedge (t_k, \alpha_k) \right) \\ \bigvee \\ \left(\bigvee_{L_\alpha^t, \alpha_l^L} \left(\begin{array}{c} (t_0^1, s_1^1, \bot) \wedge \ldots \wedge (t_k^1, s_{k+1}^1, \bot) \wedge \\ (t_0^2, s_1^2, \alpha_0) \wedge \ldots \wedge (t_l^2, s_{l+1}^2, \alpha_l) \end{array} \right) \right) \end{array}$$

Transitions at pair states

The transition function for pair states has only one difference to the one in the finite context which is the annotation of the states with $\bot$ and $\top$. The set $R_a^{(t,s)}$ is defined identically to the eponymous sets in the 2NFA to AFA conversion.

$$R_a^{(t,s)} = \left\{ \langle t_0, s_1, t_1, \ldots, s_k, t_k, s_{k+1} \rangle \;\middle|\; \begin{array}{l} 0 \le k < n \\ (t_0, 1) \in \delta(t, a) \\ (s, -1) \in \delta(s_{k+1}, a) \\ \forall i < j, s_i \ne s_j \text{ and } t_i \ne t_j \\ \forall i, (t_i, 1) \in \delta(s_i, a) \end{array} \right\}$$

If a pair state $(t, s) \in (S \times S)$ is annotated with $\top$, we have to verify that there was indeed a visit to F on the segment of the zigzag run connecting t to s. Of course, if $t \in F$ or $s \in F$, the promise is already kept. Based on these conditions, we define a set of sequences of $\bot$ and $\top$.

$$\alpha_{s,t,k}^R = \left\{ \langle \alpha_0, \ldots, \alpha_k \rangle \in \{\bot, \top\}^{k+1} \;\middle|\; \begin{array}{l} \text{If } s \notin F \text{ and } t \notin F \text{ then } \exists ! i \text{ s.t. } \alpha_i = \top \\ \text{Otherwise } \forall\ 0 \le i \le k, \alpha_i = \bot \end{array} \right\}$$

The transition function of the ABA will now have to choose a sequence of states and sequence of $\perp$ and $\top$. We define η for pair states as follows:

$$\eta((t,s,\perp),a) = \begin{cases} \textbf{true} & \text{If } (s,-1) \in \delta(t,a) \\ \bigvee\limits_{R_a^{(t,s)}} (t_0,s_1,\perp) \wedge (t_1,s_2) \wedge \ldots \wedge (t_k,s_{k+1},\perp) & \text{Otherwise} \end{cases}$$

$$\eta((t,s,\top),a) = \begin{cases} \textbf{true} & \begin{array}{l}\text{If } (s,-1) \in \delta(t,a) \text{ and} \\ (s \in F \text{ or } t \in F)\end{array} \\ \bigvee\limits_{R_a^{(t,s)},\alpha^R_{s,t,k}} (t_0,s_1,\alpha_0) \wedge \ldots \wedge (t_k,s_{k+1},\alpha_k) & \text{Otherwise} \end{cases}$$

The transition function η is now completely defined (and therefore the ABA is, too). Since the details of this definition are quite technical, we want to illustrate the main ideas by providing a detailed example of a 2NBA zigzag run ending in an infinite loop. We will then discuss the corresponding ABA run tree.

Example 3.2 (2NBA zigzag run with loop and corresponding ABA run tree)**.** *There are two accepting runs given in this example. Figure 3.8 shows a 2NBA zigzag run on an input word w of infinite length. The states s_2, s_8 and s_{10} are accepting states, so $\{s_2, s_8, s_{10}\} \subseteq F$. The 2NBA runs into an infinite loop when reading w_1 in s_5. So the run subsequence*

$$(s_5,1),(s_6,2),(s_7,3),(s_8,4),(s_9,3),(s_{10},2),(s_{11},1),(s_{12},2)$$

repeats itself infinitely often. Since s_8 and s_{10} are visited an infinite number of times, the 2NBA run itself is indeed accepting.

In Figure 3.9 wee see the simulation of the zigzag run by the ABA. As we can see, the run tree is finite. This results from the zigzag run ending in an infinite loop. When reading w_1 in singleton state $(s_1,\perp)$, the automaton guesses this loop to start in s_5 while reading the same symbol. It further predicts that s_{11} will also read w_1 inside the loop. Now the automaton has to guess the successors of these three states processing w_1. It chooses the successor of s_5 to be s_6, the successor of s_{11} to be s_{12} and the successor of the current state s_1 to be s_2. So there have to be three processes: A pair state (s_2,s_5,α) connecting the current state to the start of the loop and the two pair states (s_6,s_{11},β) and (s_{12},s_5,γ) which are defining the loop. Let's take a closer look at these states to determine how to choose α, β and γ from $\{\perp,\top\}$ to ensure the acceptance of the run.

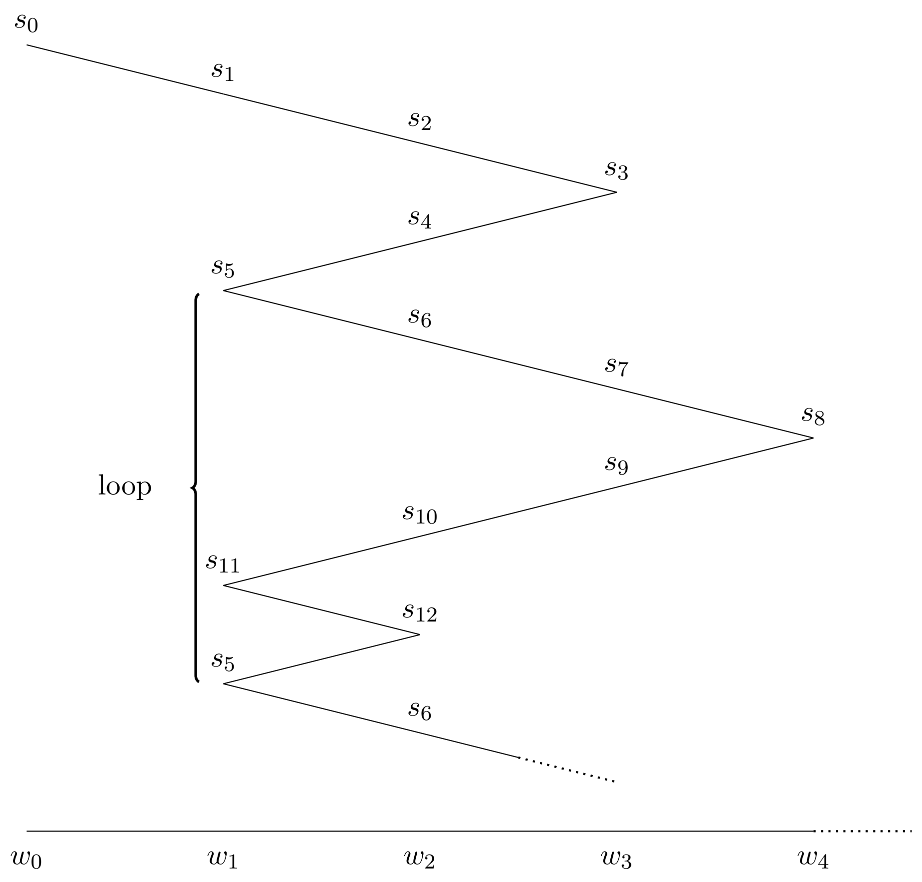

FIGURE 3.8: Example of a 2NBA zigzag run ending in an infinite loop

The pair state (s_2, s_5, α) describes the segment leading from the last singleton state $(s_1, \bot)$ to the start of the loop. Since it is of finite length and is not repeated in the zigzag run, there can only be a finite number of visits to the acceptance set. Therefore the pair state can't make a difference in the acceptance of the total run and we don't have to bother with the question of possible accepting states in this segment. It follows that, although there is a visit to the accepting state s_2, α is set to $\bot$. The following pair state $(s_3, s_4, \bot)$ doesn't have to include an accepting state as well. Since $(s_4, -1) \in \delta(s_3, w_3)$, we get the transition $\eta((s_3, s_4, \bot), w_3) =$ **true** *and are done with this branch.*

The loop itself is described by (s_6, s_{11}, β) and (s_{12}, s_5, γ), which means that there has to be some visit to an accepting state in the induced segments. In fact, there are two such states in the segment (namely s_7 and s_{10}), but guessing only one of them is sufficient for the acceptance of the run tree. So the ABA sets a promise in (s_6, s_{11}, β), which means $\beta = \top$ and therefore $\gamma = \bot$. Since $(s_5, -1) \in \delta(s_{12}, w_2)$, we get the transition $\eta((s_{12}, s_5, \bot), w_2) =$ **true** *and don't have to look further at this branch.*

Now we investigate $(s_6, s_{11}, \top)$. Since none the two states s_6 and s_{11} is an accepting state, the promise has to be passed on to a following state. The ABA guesses $(s_7, s_{10}, \top)$

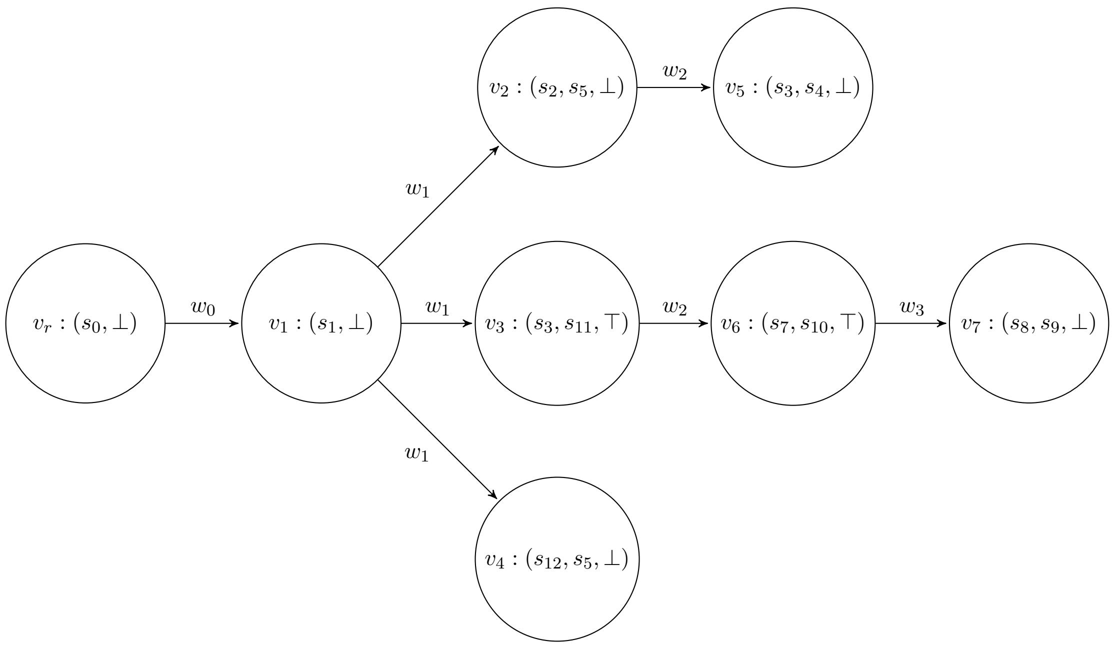

FIGURE 3.9: The corresponding ABA run tree to the zigzag run shown in Figure 3.8

to be the successor. Because of $s_{10} \in F$, the promise is satisfied reaching this state. But since s_{10} is no valid successor to s_7 in the 2NBA, there has to be another pair state following on $(s_7, s_{10}, \top)$. The ABA takes another guess and chooses $(s_8, s_9, \bot)$. Although s_8 is an accepting state itself, the promise for this branch has already been fulfilled in the former pair state, so the acceptance of s_8 is not considered in the ABA run. We see that $(s_9, -1) \in \delta(s_8, w_4)$, so the last branch we had to consider ends with the transition $\eta((s_8, s_9, \bot), w_4) = \textbf{true}$.

All leaves of the finite run tree ended in some pair state which is (in combination with the corresponding symbol w_i) defined as **true** *by the transition function η. Therefore the ABA run is indeed accepting.*

We summarize the accomplishments of this subsection: Starting from an arbitrary n-state 2NBA $N = (\Sigma, S, s_0, \delta, F)$, we first described the conversion of N to another 2NBA $N' = (\Sigma, S', (s_0, \bot), \delta', F')$ which does not use any ε-moves and accepts the same language as N. From there on, we constructed an ABA $\mathcal{A} = (\Sigma, Q, q_0, \eta, F') = (\Sigma, (S' \cup (S' \times S')) \times \{\bot, \top\}, ((s_0, \bot), \bot), \eta, (S' \times \{\top\}) \cup (F' \times \{\bot\}))$ with $\mathcal{O}(n^2)$ states and $L(\mathcal{A}) = L(N') = L(N)$.

3.2 Automata Complementation

In this section, we will discuss a complementation method for alternating Büchi automata proposed by Kupferman and Vardi in [6]. Our ambition is to construct a weak alternating automaton accepting the complement language of the original ABA. We will henceforth refer to this WAA as the *Complement WAA*. This conversion will include a quadratic blowup in the number of states.

We already described the conversion of a 2NBA to an ABA in the former section of this chapter, which included a quadratic blowup in the number of states as well. Therefore, the composition of the 2NBA to ABA and the ABA to Complement WAA conversions provides a complementation method for 2NBAs including a biquadratic blowup. In [11], Piterman and Vardi reasoned about a way to omit the quadratic blowup in the ABA to Complement WAA part when the ABA itself was constructed according to their 2NBA to ABA conversion. So the overall complementation of the 2NBA will be achieved with a quadratic blowup in the number of states.

3.2.1 ABA to Complement WAA

Given an ABA $\mathcal{A} = (\Sigma, Q, q_0, \eta, F)$, our goal is the construction of a Complement WAA $\mathcal{A}' = (\Sigma, Q', q_0', \eta', F')$. To achieve this, we first dualize the transition function of A to get an alternating co-Büchi automaton $\widetilde{\mathcal{A}}$. From there, we want to build $\mathcal{A}'$ capable of simulating accepting runs of $\widetilde{\mathcal{A}}$. Overall, we want the construction to ensure that $\Sigma^\omega \setminus L(\mathcal{A}) = L(\widetilde{\mathcal{A}}) = L(\mathcal{A}')$ holds.

Before we can describe the conversion of $\widetilde{\mathcal{A}}$ to $\mathcal{A}'$, we have to discuss some properties of the runs of ACAs.

Memoryless runs

In *Chapter 2*, we described the run of an ACA as a labeled tree (T, r) with $T = (V, E)$, in which each vertex is labeled by a state of the automaton. Our definition includes the possibility of a set of vertices $V_q^l \subseteq V$ of same depth l in the tree to be labeled by the same state q from the ACA state set Q. If we merge all the vertices from V_q^l into one single vertex v_q^l of depth l and labeled by q, we end up with a graph in which all vertices of same depth are labeled differently. We denote this graph by $G_{r'} = (V', E')$ with vertex set $V' = \{v_q^l \mid \exists v \in V \text{ with } d(v) = l \text{ and } r(v) = q\}$ and edge set $E' = \{(u_q^l, v_{q'}^{l'}) \mid \exists (u, v) \in E \text{ with } u \in U_q^l, v \in V_{q'}^{l'}\}$. The function $r' : V' \to Q$ labels each vertex v_q^l by the corresponding state q. Combined, we say the tuple $(G_{r'}, r')$ to be a *memoryless run* of the ACA. The acceptance conditions stays the same, so a memoryless run is accepting if and only if for every (possibly infinite) path in the run there are finitely many vertices labeled by accepting states.

In [4], Emerson and Jutla show that whenever an ACA accepts a word w, it also has to accept w by means of a memoryless run. It will therefore be sufficient to discuss memoryless runs exclusively for the rest of this section. We remark, that the graph $G_{r'}$ of a memoryless run is no tree, but a *directed acyclic graph* (DAG), since vertices are now allowed to have multiple predecessors. We will therefore talk about "run DAGs" instead of "run trees".

Anatomy of memoryless runs

Let (G_r, r) with $G_r = (V, E)$ be an accepting memoryless run of the ACA $\widetilde{\mathcal{A}}$ (we also say G_r to be an accepting run DAG). For simplicity, we will refer to the vertex $v_q^l \in V$ by the tuple (q, l). A vertex (q', l') is reachable from another vertex (q, l) if and only if there is a finite sequence successive vertices that starts with (q, l) and ends with (q', l'). We further say a vertex (q, l) to be an *α-vertex* iff q is an accepting state. We denote the number of states in the ACA by $|Q| = n$.

We make two observations. First, for any given depth l in G_r (which we will also call a *level* l in G_r), there can be at most n vertices. This arises from the fact that more than n vertices would imply that there have to be two vertices labeled by the same state, which is not possible by construction of G_r. The second observation is, that there are only finitely many α-vertices on each path in G_r, since (G_r, r) is an accepting run.

Consider a DAG $G \subseteq G_r$. We call a vertex (q, l) *endangered* in G if and only if there are only finitely many vertices reachable from (q, l). This means, that (q, l) can't be a member of any infinite path. We further say a vertex to be *safe* in G iff every vertex that is reachable from (q, l) is not an α-vertex. This implies in particular that no α-vertex can be a safe vertex.

We will now define an infinite hierarchy of DAGs $G_0 \supseteq G_1 \supseteq G_2 \supseteq \ldots$ inductively:

- $G_0 = G_r$
- $G_{2i+1} = G_{2i} \setminus \{(q, l) \mid (q, l) \text{ is endangered in } G_{2i}\}$
- $G_{2i+2} = G_{2i+1} \setminus \{(q, l) \mid (q, l) \text{ is safe in } G_{2i+1}\}$

Note that, if a graph G_{2i} is finite, all the following graphs G_{2i+j} with $j \in \mathbb{N}$ must be empty. We provide an example for such a DAG hierarchy.

Example 3.3 (DAG hierarchy)**.** *Consider the DAG G_0 shown on the far left side in Figure 3.10. There are three "columns" of vertices in this graph. The column in the middle contains filled circles which are representing α-vertices, while the unfilled circles in the outer columns aren't labeled by accepting states. The "rows" on the other hand are corresponding to the symbols of the input word w, meaning all states of the lth row are reading the symbol w_l.*

G_0 contains an infinite number of infinite paths. Each of these paths visit at most two (and therefore finitely many) α-vertices. The DAG is therefore an accepting run DAG. Except for the unique infinite path going through the vertices in the left column, all paths are finally staying in the right column.

As we can see, G_1 equals G_0, since there are no endangered vertices in initial graph. In G_2, we removed all safe vertices from G_1. This results in the vanishing of the complete column on the right, since for all these vertices there aren't any reachable α-vertices. G_2 now has an infinite number of α-vertices with no more reachable vertices, which are therefore endangered. Removing these α-vertices, we achieve G_3. This graph consists solely of a single infinite path, which contains no α-vertices except for its very first vertex. So these states are safe and no part of G_4. Since G_4 is finite, all vertices in this graph have to be endangered. Therefore, all graphs G_i with $i \geq 5$ are empty.

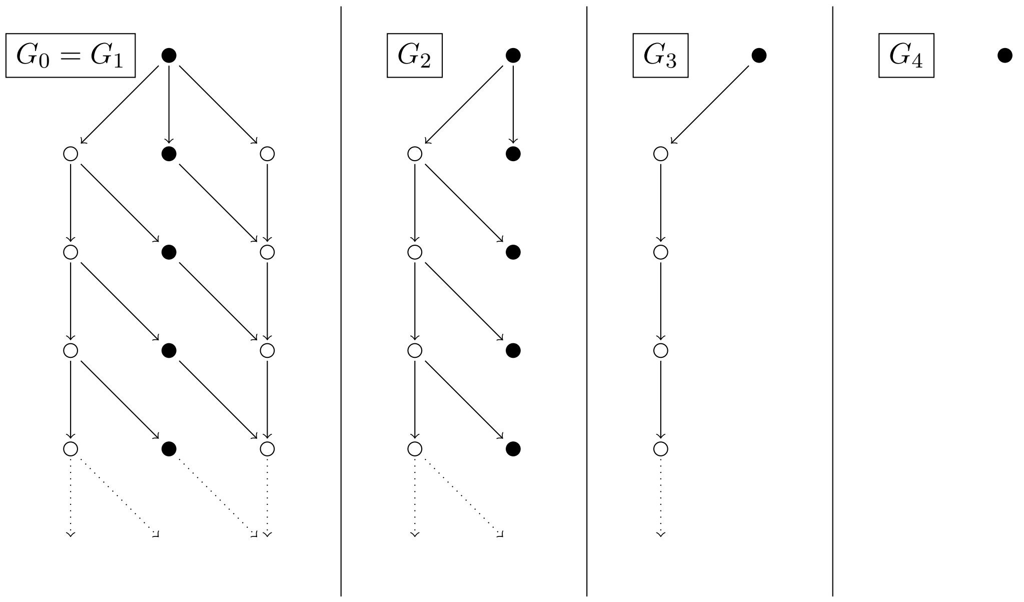

FIGURE 3.10: Example of a DAG hierarchy. The filled circles represent the α-vertices. The graphs G_i with $i \geq 5$ are empty.

Kupferman and Vardi show in [6], that the graph G_{2n} has to be finite for every run DAG G_r and therefore G_{2n+1} and all following graphs have to be empty. We will not reproduce their proof, but give an informal summary of their argumentation.

Consider an infinite accepting run DAG $G_r = G_0$. As we saw in Example 3.3, G_0 does not have to contain any endangered vertices, so we assume $G_1 = G_0$ without loss of generality. Now there have to be some safe vertices in G_1, because otherwise all vertices would be reaching another α-vertex and G_r wouldn't be accepting. There also can't be finitely many safe vertices in G_1, since in this case these vertices would have been endangered in G_0. So there is an infinite number of safe vertices in G_1. It may well be that these vertices are all merging into a single infinite path (also seen in Example 3.3). Now there has to be some level $l \in \mathbb{N}_0$ in G_2, such that for every level l' greater than or equal to l there are at most $n-1$ vertices (we recall, that the DAG can at most have n vertices on every level). Assuming G_2 isn't finite, Piterman and Vardi repeat this procedure, always having some level in G_{2i} for $i \in \mathbb{N}_0$ with at most $n-i$ vertices. Therefore, in G_{2n}, there has to be some level without any vertices, which means that G_{2n} has to be finite and all following graphs in the hierarchy have to be empty.

Since G_{2n+1} always has to be empty, we conclude that every vertex (q,l) of the run DAG has to have a unique index i with $0 \leq i \leq n$ in such a way that it is either endangered in G_{2i} or safe in G_{2i+1}. We use this to define the *rank* of a vertex (q,l) as follows:

$$rank(q,l) = \begin{cases} 2i & \text{If } (q,l) \text{ is endangered in } G_{2i} \\ 2i+1 & \text{If } (q,l) \text{ is safe in } G_{2i+1} \end{cases}$$

We denote the set $\{0,1,\ldots,k\}$ by $[k]$ and refer to its subset of all odd members by $[k]^{odd}$. So it follows, that for every vertex (q,l) of a run DAG, $rank(q,l) \in [2n]$ must hold. Per definition, no α-vertex (q',l') can be a safe vertex of any graph and therefore $rank(q',l') \notin [2n]^{odd}$.

We remark, that if a vertex (q',l') is reachable from another vertex (q,l), the non-strict inequality $rank(q',l') \leq rank(q,l)$ must hold. Since there are only finitely many ranks ($2n$ at most), it follows that for every infinite path in the DAG, there has to be some vertex (q,l) of an odd rank for which all reachable states (q',l') on the path are of the same rank, so $rank(q,l) = rank(q',l')$.

As we have seen, if an ACA has an accepting memoryless run (G_r, r) on an infinite word w, the run DAG G_r is highly structured by the concept of ranks we just presented. The fact, that vertices in G_r can only reach vertices of a lower or equal rank, will be used to define the states of the WAA we're about to construct in a partial ordering. Another important fact is, that the ordering of the vertices by rank includes the separation of sets including α-vertices (which are of an even rank) and sets containing no α-vertices at all (which are of an odd rank). This will be used to construct the acceptance set of the WAA.

Construction of the WAA

Given the alternating co-Büchi automaton $\widetilde{\mathcal{A}} = (\Sigma, Q, q_0, \widetilde{\eta}, F)$, we construct the language equivalent weak alternating automaton $\mathcal{A}' = (\Sigma, Q', q_0', \widetilde{\eta}', F')$ as follows:

- $Q' = Q \times [2n]$

  The state set of $\mathcal{A}'$ is the combination of all original ACA states with all possible ranks. Being in state (q, i) while reading the lth letter of the input word means that $\mathcal{A}'$ guesses the rank of (q, l) in an accepting run DAG to be i. Since we combine n states with $2n$ ranks, we get a quadratic blowup in the number of states.

- $q_0' = (q_0, 2n)$

  We assume the highest possible rank for the start state q_0 of $\widetilde{\mathcal{A}}$, so that we don't limit the number of accepting ACA runs that can be simulated by $\mathcal{A}'$ by choosing the starting rank too low.

- $\widetilde{\eta}' : Q' \times \Sigma \to \mathcal{B}^+(Q')$

  To describe the transition function of $\mathcal{A}'$, we first define the function

  $$\mathit{release} : \mathcal{B}^+(Q) \times [2n] \to \mathcal{B}^+(Q')$$

  as follows: *release* maps a positive Boolean formula φ over the states of $\widetilde{\mathcal{A}}$ and a rank $i \in [2n]$ to a positive Boolean formula φ' over the states of $\mathcal{A}'$. This happens by replacing every atom q in φ by the logical disjunction of all states (q, i') from $\mathcal{A}'$ with $0 \leq i' \leq i$. An example provided in [6] looks as follows:

  $$\mathit{release}(q_3 \wedge q_5, 2) = ((q_3, 2) \vee (q_3, 1) \vee (q_3, 0)) \wedge ((q_5, 2) \vee (q_5, 1) \vee (q_5, 0))$$

  Now we can the define the actual transition function of the WAA for some state $(q, l) \in Q'$ and some symbol $a \in \Sigma$:

  $$\widetilde{\eta}'((q, l), a) = \begin{cases} \mathit{release}(\widetilde{\eta}(q, a), l) & \text{If } q \notin F \text{ or } l \text{ is even} \\ \mathbf{false} & \text{If } q \in F \text{ and } l \text{ is odd} \end{cases}$$

  This means, that if the automaton guessed an α-vertex to be of an odd rank, it already guessed wrong and is not simulating an accepting ACA run. Otherwise, the corresponding transition function of $\widetilde{\mathcal{A}}$ is applied to q and a. This simulates the behavior of $\widetilde{\mathcal{A}}$ by providing a positive Boolean formula $\varphi \in \mathcal{B}^+(Q)$. From there, $\mathit{release}(\varphi, l)$ is used to guess the ranks of the states reading the next symbol of the input word.

- $F' = Q \times [2n]^{odd}$

  We define all states of $\mathcal{A}'$ being of an odd rank to be accepting. This means, that an accepting run of $\mathcal{A}'$ has to contain infinitely many states of an odd rank on every infinite path.

A formal proof of the correctness of this construction can be found in [6]. Informally, we can see that $\mathcal{A}'$ indeed is a WAA, since there is the partial ordering of the states provided by their ranks. Further, the transition function only leads from a certain state to following states of the same or some lower rank. So the infinite run finally has to "get trapped" at some rank and it can only be accepting if this rank is odd. This means, that at some point, the simulated run DAG of the ACA $\widetilde{\mathcal{A}}$ stops visiting vertices of even ranks. Since all α-vertices are ranked by an even number, this corresponds to $\widetilde{\mathcal{A}}$ only visiting finitely many α-vertices in its accepting run DAG.

We showed how to complement an ABA by means of a Complement WAA. The blowup in the number of states in this automata conversion is quadratic. Next, we will use this knowledge in combination with the construction presented in 3.1.2 to describe a complementation method for 2NBAs.

3.2.2 2NBA to Complement WAA

Consider a 2NBA $N = (\Sigma, S, s_0, \delta, F)$. We could complement N by using the 2NBA to ABA conversion discussed in 3.1.2 and use the ABA complementation method described in 3.2.1 on the resulting ABA. Each of these two conversions involves a quadratic blowup in the number of states, so the overall blowup of this composition would be a biquadratic one. In [11], Piterman and Vardi show a way to avoid the second quadratic factor. We will reproduce their argumentation in the following paragraphs.

Starting with N, we use the 2NBA to ABA conversion to achieve a language equivalent ABA $\mathcal{A}$. Dualizing $\mathcal{A}$'s transition function, we achieve the ACA $\widetilde{\mathcal{A}} = (\Sigma, Q, q_0, \widetilde{\eta}, \mathcal{F})$. So far, we did nothing new and got a quadratic blowup in the number of states of N. The critical point comes in the conversion of $\widetilde{\mathcal{A}}$ to the WAA $\mathcal{A}'$: We know from the previous constructions, that the state set of $\widetilde{\mathcal{A}}$ consists of singleton states and pair states, so $Q = (S \cup (S \times S)) \cup \{\bot, \top\}$. We further know, that pair states aren't members of the acceptance set F and that only pair states are reachable from a pair state. This means, that in an accepting run DAG G_r of $\widetilde{\mathcal{A}}$, there are no α-vertices reachable from any vertex labeled by a pair state. Therefore, all these vertices have do be endangered in $G_r = G_0$, so 0 is the only possible rank for these vertices. Since they can only have one rank and never reach singleton states, we can pass on ranking the pair states at all.

We will now define the Complement WAA $\mathcal{A}' = (\Sigma, Q', q_0', \widetilde{\eta}', F')$ as follows:

- $Q' = (S \times \{\bot, \top\} \times [2n]) \cup (S \times S \times \{\bot, \top\})$

  With $|S| = n$, the quadratic blowup from the 2NBA to ABA conversion arises from the construction of the $\mathcal{O}(n^2)$ pair states. But since we don't rank the pair states, we only have to consider the blowup resulting from the combination of the $\mathcal{O}(n)$ singleton states with $2n$ ranks. This results in $\mathcal{O}(n^2)$ ranked states, which are added to the $\mathcal{O}(n^2)$ pair states. So we end up with $\mathcal{O}(n^2)$ states in total (and therefore omit the biquadratic blowup of the "primitive" method).

- $q_0' = (s_0, \bot, 2n)$

  Again, we start with the former start state with a rank of $2n$.

- $\widetilde{\eta}' : Q' \times \Sigma \to \mathcal{B}^+(Q')$

  The function *release* is defined in the same way as introduced in 3.2.1. To describe the transition function, we have to distinguish between the transition of ranked singleton state $(s, \alpha, l) \in (S \times \{\bot, \top\} \times [2n])$ and the transition of a pair state $(t, s, \alpha) \in (S \times S \times \{\bot, \top\})$. Given a symbol $a \in \Sigma$, we define $\widetilde{\eta}'$ as follows:

  $$\widetilde{\eta}'((s, \alpha, l), a) = \begin{cases} release(\widetilde{\eta}((s, \alpha), a), l) & \text{If } (s, \alpha) \notin F' \text{ or } l \text{ is even} \\ \textbf{false} & \text{If } (s, \alpha) \in F' \text{ and } l \text{ is odd} \end{cases}$$

  $$\widetilde{\eta}'((t, s, \alpha), a) = \widetilde{\eta}((t, s, \alpha), a)$$

  If the current state is a pair state, we do exactly the same the ACA $\widetilde{\mathcal{A}}$ would do. Otherwise, we proceed as described in 3.2.1.

- $F' = (S \times \{\bot, \top\} \times [2n]^{odd}) \cup (S \times S \times \{\bot, \top\})$

  We say the states of an odd rank to be accepting for the same reasons as before. Since the pair states can't be the labels of α-vertices in an accepting run DAG of $\widetilde{\mathcal{A}}$ either, we define them as accepting, too.

We have now constructed a Complement WAA $\mathcal{A}'$ to the 2NBA N with a quadratic blowup in the number of states. At this point, we have finished the discussion of the automata conversions we aimed to present in this thesis. The next chapter will concern the implementation of these methods and provides examples of automata resulting from them.

Chapter 4

Implementation

Automata conversions aren't discussed solely for the achievement of theoretical insight, they can also be of relevance for practical purposes. In *runtime verification*, we formulate the intended behavior of a system S as a temporal logical expression. This expression is transformed into a finite state automaton, which evaluates at every discrete time step whether an implementation of S is still satisfying the intended behavior or not. Pragmatically, we call the FSA monitoring S a *monitor*.

Usually, the FSA we transformed a temporal logical formula immediately into, tends not to be an adequate automaton for practical monitoring intentions. We rather have to convert it to another, more suitable kind of FSA, often using multiple intermediate conversions. In the end, we get a "chain of conversions" leading from the temporal logical expression we started with to the final monitor.

In this chapter, we will describe an implementation of the automata conversions presented in this thesis. We will start with an introduction to *RltlConv*, a logic and automata library capable of constructing monitors from formulas of different temporal logics. We will then discuss the embedding of our conversion methods into RltlConv. Afterwards, we investigate some of the algorithmic details of our implementation.

4.1 RltlConv

RltlConv is a logic and automata library developed at the Institute for Software Engineering and Programming Languages of the University of Lübeck. It features the interpretation of temporal logical or (ω-)regular expressions in form of finite state automata and provides various automata conversion methods for the achievement of runtime monitors.

So far, RltlConv supports monitor construction for formulas from the set of LTL, RLTL (*regular linear temporal logic*, introduced by Leucker and Sánchez in [7]) and (ω-)regular expressions. The library is still under construction and it's range of functions grows steadily over time, like it did in the course of this thesis.

A typical use case looks as follows: We want to achieve a monitor by passing a system's intended behavior to the library in form of an LTL expression. This will be interpreted as an ABA, which will then be converted into a language equivalent NBA. From this NBA, there will be a conversion into an NFA, which also means a change of context from languages of infinite words to languages of finite words. This step is necessary for the construction of practically usable monitors, but it's discussion is beyond the scope of this thesis (for more information on this topic, see the construction proposed by Bauer, Leucker and Schallhart in [1]). The NFA will then be determinized and afterwards converted into a deterministic Moore machine, which can be used to monitor a system's behavior during runtime. Overall, the chain of conversions for this example looks as follows:

$$\text{LTL} \rightarrow \text{ABA} \rightarrow \text{NBA} \rightarrow \text{NFA} \rightarrow \text{DFA} \rightarrow \text{DMoore}$$

4.1.1 Motivation for Presented Automata Conversions

In *Chapter 3*, we described the conversion of 2NBAs to ABAs. The motivation to implement this method in RltlConv is the following: There is a direct transformation from regular expressions with past operators (as defined by Sánchez and Leucker in [13]) to 2NFAs. If we extend this notion to the context of infinite words, we are able to achieve a 2NBA from an ω-regular expression with past operators. As the previously described use case suggests, all intermediate steps leading from an ABA to a deterministic Moore machine (and therefore a practical monitor) are already implemented in RltlConv. The implementation of the 2NBA to ABA conversion gets us one step closer to the construction of monitors based on ω-regular expressions using past operators. The chain of conversions looks as follows:

$$\omega\text{-RegEx with past} \rightarrow \text{2NBA} \rightarrow \text{ABA} \rightarrow \text{NBA} \rightarrow \text{NFA} \rightarrow \text{DFA} \rightarrow \text{DMoore}$$

We note that, up to the present day, the first step of this chain (the construction of 2NBAs) is not supported in RltlConv and therefore the monitor construction based on ω-regular expressions using past operators is not yet possible.

Since the description of a system's intended behavior using ω-regular expressions with past operators does not appear to be an intuitive process, we'd rather use temporal logics with past to do so. As an example, Sánchez and Leucker introduced the *regular linear temporal logic with past* (pRLTL for short) in [13]. pRLTL formulas can be transformed into 2-way alternating 3-parity automata (2A3PAs), which will then be converted into NBAs using a conversion method proposed by Dax and Klaedtke in [3]. Since this conversion involves the complementation of 2NBAs, we have given a motivation to implement the 2NBA to Complement WAA conversion presented in the former chapter of this thesis.

4.1.2 Embedding of New Conversions in RltlConv

RltlConv is mostly written in the object-oriented, functional programming language Scala. All the code produced in the course of this thesis can be found in the two files `Apa.scala` and `Nba.scala`, which are both located inside the project package `de.uni_luebeck.isp.rltlconv.automata`. The only other files referenced by our code, which are `Common.scala` and `PosBool.scala`, can be found in the same package. Figure 4.1 shows all the classes modified or referenced during the implementation of the new automata conversions presented in this thesis.

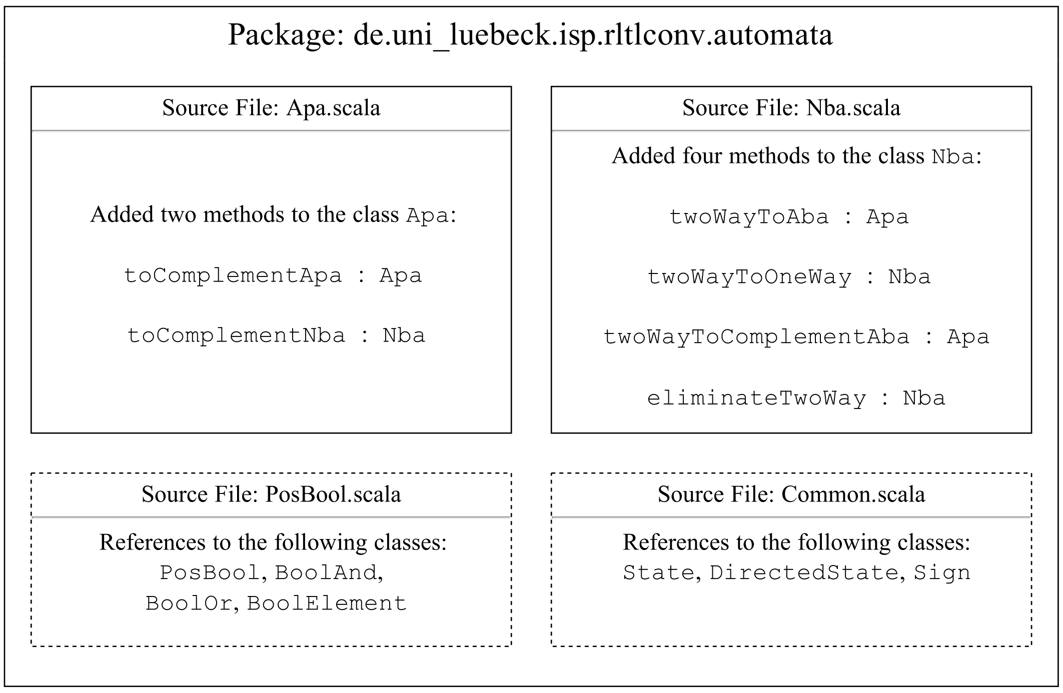
Package: de.uni_luebeck.isp.rltlconv.automata
Source File: Apa.scala
Added two methods to the class Apa:
toComplementApa : Apa
toComplementNba : Nba
Source File: Nba.scala
Added four methods to the class Nba:
twoWayToAba : Apa
twoWayToOneWay : Nba
twoWayToComplementAba : Apa
eliminateTwoWay : Nba
Source File: PosBool.scala
References to the following classes:
PosBool, BoolAnd, BoolOr, BoolElement
Source File: Common.scala
References to the following classes:
State, DirectedState, Sign

FIGURE 4.1: Overview of modified and referenced classes inside the `automata` package. While we modified the classes `Apa` and `Nba` by implementing additional methods, we only referenced multiple classes from the file `PosBool.scala` and `Common.scala` without changing any of the original RltlConv code.

The source file `Apa.scala` includes the class `Apa`, which represents alternating parity automata in RltlConv. The APAs described in this class may be of an arbitrary parity $k \in \mathbb{N}$ and can be 1-way as well as 2-way automata. As we already stated in *Chapter 2*, there are no alternating Büchi automata in RltlConv. Instead, we interpret an ABA as a A2PA in which the ABA's accepting set F equals the A2PA's set Q_2 (and therefore the ABA's $Q \setminus F$ equals the A2PA's Q_1). This way, we can simulate ABAs using the class `Apa`.

We implemented two methods in `Apa`. The first one is `toComplementAba`, which is called on an instance $\mathcal{A}$ of `Apa` and returns another instance $\mathcal{A}'$ of `Apa` such that $L(\mathcal{A}') = \Sigma^\omega \setminus L(\mathcal{A})$. The construction of $\mathcal{A}'$ follows the ABA complementation method discussed in 3.2.1 (where we achieved a Complement WAA, which matches the return value of $\mathcal{A}'$ being an ABA, since ABAs are a generalization of WAAs). The complementation only works for ABAs, so `toComplementAba` will only be executed if all parities in $\mathcal{A}$ are elements of $\{1, 2\}$.

The second method added to the `Apa` class is `toComplementNba`. This method simply executes `toComplementAba` and calls the already existing `toNba` on the resulting Complement WAA. We therefore end up with an NBA accepting the complement language of the original ABA.

In `Nba.scala`, we find the class `Nba`, which is the RltlConv representation of both 1-way and 2-way NBAs. We implemented four new methods in `Nba`. First, `twoWayToAba` converts a 2NBA into an equivalent ABA using the construction presented in 3.1.2. In fact, the conversion works on a 1-way NBA as well.

The second method is `twoWayToOneWay`. This is the composition of `twoWayToAba` and the already implemented `toNba`, so it converts a 2NBA into an equivalent 1-way NBA.

The third method, `twoWayToComplementAba`, constructs the Complement WAA of a 2NBA. We could achieve such a conversion by the composition of `twoWayToAba` and `Apa.toComplementAba`, but this would involve an biquadratic blowup in the number of states of the original 2NBA. Instead, we avoid the quadratic factor of the ABA complementation step by implementing the 2NBA to Complement WAA conversion discussed in 3.2.2.

Finally, in `eliminateTwoWay`, we check whether the calling NBA is a 1-way or a 2-way automaton. In the latter case, it is converted into a 1-way NBA by calling `twoWayToAba` first and executing `toNba` on the resulting ABA.

4.2 Explanation of Implemented Methods

We will now explain the functionality of our implementation in more detail. The three automata conversion described in this thesis are represented by `Nba.twoWayToAba`, `Apa.toComplementAba` and `Nba.twoWayToComplementAba`, so we will discuss these methods one by one.

4.2.1 2NBA to ABA

The 2NBA to ABA conversion, as we discussed it earlier in 3.1.2, is implemented in `Nba.twoWayToAba`. All functions mentioned in this subsection are solely defined in the scope of this method. The first part of the conversion, which was described as the elimination of ε-moves, can be skipped, since the 2NBAs in RltlConv are already ε-move free.

We will now go through the different phases of the implemented ABA construction. The title of each phase can also be found as a headline in the Scala code, so our explanations can be easily attributed to the corresponding part of the implementation. As usually, we refer to the 2NBA by N and to the ABA by $\mathcal{A}$.

- *build states*:
We start by doing a depth first search in the 2NBA, which shall provide us with the set of all states in N which are reachable from one of its start states. We take the Cartesian product of the reachable states to get state pairs. Now we enhance the single states as well as the state pairs once with $\perp$ and once with $\top$, to achieve the actual singleton and pair states of the ABA.

- *build start states*:
In RltlConv, 2NBAs can have a set of start states instead of just a single start state. Furthermore, the start states of APAs (and therefore ABAs) are represented by a positive Boolean formula over the set of states. So we build the initial states of the ABA as a logical disjunction over all the singleton states, which are both enriched by $\perp$ and for which the corresponding 2NBA state is a start state.

- *build colors*:
We refer to parities as “colors” in RltlConv. We assign the color 2 to every accepting singleton state in $\mathcal{A}$, while all the other states get the color 1.

- *build singleton state sequences*:
In this phase, we construct the sequence sets R_a^t for all states t of N and for all symbols a of the alphabet. We do this by recursively adding states to an initially empty list as long

as the resulting list is still a valid sequence in the sense of the R^t_a conditions presented in 3.1.2. If we have no more valid possibilities of adding another state, we trace back. After the construction of the R^t_a sets, we build the corresponding L^t_a sets. We recall, that a set L^t_a contains pairs of sequences, for which each single sequence must be a member of the corresponding R^t_a. So instead of starting an exhaustive search of the set of states again, we check all pairs of sequences in R^t_a for the conditions of L^t_a.

- *build pair state sequences*:
We build the $R^{(t,s)}_a$ sequence sets. We do this by simply taking all the R^t_a sequences and check for every state s of N whether a sequence extended by s is a valid $R^{(t,s)}_a$ sequence (and therefore s an appropriate s_{k+1} candidate).

- *build singleton state alpha sequences*:
The singleton state alpha sequences with respect to some state t from N and some symbol a from the alphabet are R^t_a and L^t_a sequences in which every state is enriched by $\top$ or $\bot$. This enrichment happens according to the corresponding α^R_k and α^L_l sequences and is done recursively. We remark, that while R^t_a and L^t_a contain sequences of states from N, the singleton state alpha sequences consist of states of $\mathcal{A}$.

- *build pair state alpha sequences*:
We build the pair state alpha sequences in a similar way we build the singleton state alpha sequences before. All sequences in $R^{(t,s)}_a$ are recursively enriched by $\top$ and $\bot$ according to the conditions given by $\alpha^R_{s,t,k}$.

- *build singleton state transitions*:
For every 2NBA state t and every symbol from the alphabet a, we construct the transition $\eta((t,\bot),a) = \eta((t,\top),a)$ as a positive Boolean formula over the state set of $\mathcal{A}$. We do this the following way: For every singleton state alpha sequence with respect to t and a, we take the logical conjunction of all sequence members and therefore receive a representing positive Boolean formula for every sequence. Now we build the logical disjunction over all these formulas and get the proper transition as described in 3.1.2.

- *build pair state transitions*:
The pair state transitions are build in a similar way to the singleton state transitions. Since $\eta((t,s,\bot),a)$ $\eta((t,s,\top),a)$ are not defined equally, we have to distinguish these cases according to 3.1.2.

- *build ABA*:
We assemble all the parts we built in the previous phases and construct the ABA $\mathcal{A}$. We also call the RLTLConv minimization function `toReducedApa` on $\mathcal{A}$.

4.2.2 ABA to Complement WAA

Now we investigate the implementation of the ABA complementation method presented in 3.2.1. We refer to the original ABA by $\mathcal{A}$, to the ACA by $\widetilde{\mathcal{A}}$ and to the resulting Complement WAA by $\mathcal{A}'$.

- *build coABA*:
We start with the construction of the ACA $\widetilde{\mathcal{A}}$, which happens by dualization of $\mathcal{A}$'s transition function. The dualization itself is implemented as a recursive function, which takes a positive Boolean formula over $\mathcal{A}$'s state set and breaks it down into single logical conjunctions and disjunctions. There we can substitute every $\wedge$ by $\vee$, **true** by **false**, and vice versa. We remark that, though RltlConv does not support co-Büchi acceptance conditions, there is no problem in describing the structure of an ACA as an ABA and therefore model it as an object of class `Apa`.

- *build states*:
For every state in $\widetilde{\mathcal{A}}$, there are $2n$ ranked versions of this state in the Complement WAA $\mathcal{A}'$ (where n is the number of states in $\widetilde{\mathcal{A}}$). This means, that there is one state in $\mathcal{A}'$ for every element of the Cartesian product of the states in $\widetilde{\mathcal{A}}$ and $\{0, 1, \ldots, 2n\}$.

- *build start states*:
The start states of $\mathcal{A}'$ are all the states of rank $2n$ for which the corresponding state in $\widetilde{\mathcal{A}}$ is a start state. Since the starting conditions for ABAs are described as a positive Boolean formula over the state set in RltlConv, we have to recursively break down this formula to get its atoms and therefore the proper start states.

- *build colors*:
We set all states in $\mathcal{A}'$ of odd rank to be accepting, while all states of an even rank are not accepting.

- *build transitions*:
The function *release* maps a positive Boolean formula φ over the states of $\widetilde{\mathcal{A}}$ and a rank $i \in \{0, 1, \ldots, 2n\}$ to a positive Boolean formula φ' over the states of $\mathcal{A}'$. This happens by replacing every atom q in φ by the logical disjunction of all states (q, i') from $\mathcal{A}'$ with $0 \leq i' \leq i$. From there on, the construction of the transitions of $\mathcal{A}'$ is straightforward according to 3.2.1.

- *build cABA*:
We create the Complement WAA $\mathcal{A}'$ (which is of class `Apa`) using the elements we constructed previously. We use `toReducedApa` to minimize $\mathcal{A}'$.

4.2.3 2NBA to Complement WAA

The implementation of this construction is quite similar to the one presented in the former subsection. Since we start with a 2NBA this time, we first have to convert it into an ABA (using the previously described `Nba.twoWayToAba`) to enable the construction of an ACA.

The main difference to the method described in 4.2.2 is that only the singleton states are ranked, which has to be considered in the coloring of the states and in the construction of the transitions. In exchange for this extra effort, the additional quadratic blowup in the number of states of the ACA can be avoided.

4.3 I/O Examples

We will now present some examples describing the usage of the implemented automata conversions in RltlConv. As we stated before, the construction of 2NBAs from ω-regular expressions is not yet supported, so we will have to pass the automata descriptions directly to the library.

The following file `2NbaExample.txt` encodes the same automaton we presented in the Example 2.1 on page 8. Although we introduced this automaton in *Chapter 2* as an example for 2NFAs, we will now interpret the same structure as a 2NBA. We also remark, that the non-accepting “sink” state s_3 as well as all transitions involving this state are not defined in `2NbaExample.txt`. RltlConv is able to handle undefined transitions and since our conversions invoke blowups in the number of states, we try to reduce this number as far as possible without changing the language accepted by the automaton.

Listing 4.1: Content of `2NbaExample.txt`

```
2NBA {
  ALPHABET = ["(a)", "(b)"]
  STATES = [s0, s1, s2: ACCEPTING]
  START = [s0]
  DELTA(s0, "(a)") = [s0: FORWARD]
  DELTA(s0, "(b)") = [s0: FORWARD, s1: BACK]
  DELTA(s1, "(a)") = [s2: FORWARD]
  DELTA(s2, "(b)") = [s2: FORWARD]
}
```

As we can see, the description includes the automaton type, the used alphabet, the state set (where accepting states $s \in F$ are indicated by `s: ACCEPTING`), the set of start states and a listing of all defined transitions. Every forward transition $(t, 1) \in \delta(s, a)$ is represented by `DELTA(s, "(a)")= [t: FORWARD]`, while every backward transition $(t, -1) \in \delta(s, a)$ is represented by `DELTA(s, "(a)")= [t: BACK]`.

Now that we prepared a 2NBA, we want to use RltlConv to convert it into automata of other kinds. We start RltlConv on a Unix system using the shell script `rltlconv.sh` (there is also a batch file `rltlconv.bat` for Windows). To pass `2NbaExample.txt` to RltlConv and make it recognize this as an automaton, we have to locate the input file and the shell script in the same folder and execute the following in our command line interface:

Listing 4.2: Interpretation of `2NbaExample.txt` as an automaton

```
$ sh rltlconv.sh @2NbaExample.txt --automaton
```

The output to this call is exactly the same as the content of `2NbaExample.txt`, which is not surprising, since we didn't invoke any conversion method yet. To test our implemented method `twoWayToAba`, we add `--APA` to the previous call.

Listing 4.3: Conversion of the 2NBA example into an ABA

```
$ sh rltlconv.sh @2NbaExample.txt --automaton --APA
APA {
  ALPHABET = ["(a)", "(b)"]
  STATES = [s0_bottom:1, s0_s1_bottom:1, s2_bottom:2, q0:2]
  START = s0_bottom
  DELTA(s0_bottom, "(a)") = s0_bottom OR (s0_s1_bottom AND
     s2_bottom)
  DELTA(q0, ?) = q0
  DELTA(s0_s1_bottom, "(b)") = q0
  DELTA(s2_bottom, "(b)") = s2_bottom
  DELTA(s0_bottom, "(b)") = s0_bottom
}
```

For every state in the resulting automaton, we can see the parity of the subset this state is contained in. For example, `s0_bottom:1` denotes that the singleton state $(s_0, \perp)$ is of parity 1, which implies $(s_0, \perp) \notin F$. $(s_2, \perp)$ on the other hand is of parity 2 and therefore accepting. Since there are only these two parities, the APA constructed by RltlConv is indeed an ABA.

In `DELTA(s0_bottom, "(a)")` we can see that the transition function in this constructed automaton maps to positive Boolean formulas over the state set. In this specific case, we have an AND-term nested in an OR-expression.

Another point worth mentioning is the existence of the state `q0:2`. This state arises from the APA minimization provided by RltlConv, which leads all APA transitions evaluating to **true** into an infinite loop over an accepting sink state (which is equivalent to the run branch as ending in **true**). The question mark in the transition `DELTA(q0, ?)= q0` tells us that this transition is independent of the current symbol of the input word.

Encoded automata are hard to read, so we want to get a graphical representation of the ABA we just constructed. RltlConv provides a method for automated graphical illustration based on the open-source graph visualization software Graphviz[1]. All we have to do, is to add the flag `--pdf` to our call and redirect the result into an arbitrarily named PDF file.

Listing 4.4: Creating graphical output of the resulting ABA

```
$ sh rltlconv.sh @2NbaExample.txt --automaton --APA --pdf >
    ABA.pdf
```

The graphical representation resulting from this call can be seen in Figure 4.2.

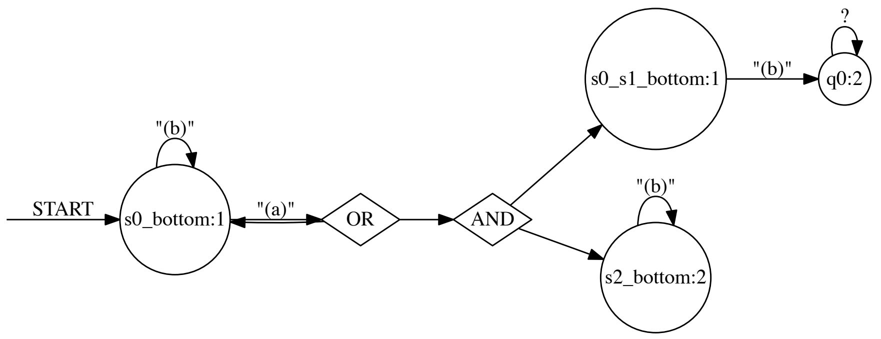

FIGURE 4.2: Graphical representation of the constructed ABA. `ABA.pdf`, automatically rendered by RltlConv.

This automaton is not minimal in its number of states. A language equivalent ABA could be achieved by deleting the states $(s_0, s_1, \bot)$ and q_0, which would replace the transition $\delta((s_0, \bot), a) = (s_0, \bot) \vee ((s_0, s_1, \bot) \wedge (s_2, \bot))$ by the transition $\delta((s_0, \bot), a) = (s_0, \bot) \vee (s_2, \bot)$. Since there would be no more alternation involved, this minimized

[1]http://www.graphviz.org/

structure could even be interpreted as an NBA. However, the minimization of ABAs in general is no trivial task and extends the scope of this thesis.

We will now construct the Complement WAA of our 2NBA example. To call the proper RltlConv method, we use the flag `--COMPABA`. Since the encoded version of the resulting automaton is considerably larger than in the 2NBA to ABA example, we take a pass on displaying the command line output. Instead, we go directly for a graphical representation which can be seen in Figure 4.3.

Listing 4.5: Conversion of the 2NBA example into the Complement WAA

```
$ sh rltlconv.sh @2NbaExample.txt --automaton --COMPABA
   --pdf > COMPABA.pdf
```

Not only can we observe a quadratic blowup in the number states, we also find a noticeable more complex pattern in the transitions of the new automaton. Again, the automaton seen in Figure 4.3 already underwent the minimization process provided by RltlConv. Still, this construction represents a vast improvement to the 2NBA complementation using an ABA as an intermediate step. Of course, we can simulate this process using our newly implemented methods as well. We use the following command:

Listing 4.6: Conversion of the 2NBA example into the Complement WAA with an intermediate step of an ABA

```
$ sh rltlconv.sh @2NbaExample.txt --automaton --APA
   --COMPABA --pdf > BIGCOMPABA.pdf
```

The graphical representation of this automaton can be seen in Figure 4.4. Clearly, the language equivalent Complement WAA shown in Figure 4.3 is significantly more succinct. Nevertheless, we think considering this larger Complement WAA to be worthwhile, since it shows the layered structure of the Complement WAAs in quite an obvious way. These layers arise from the ranking of the states and from the *release* function used in the definition of the transition function.

Now that we have provided examples for every automata conversion method we implemented, we approach the end of this thesis. The last chapter will summarize the work we have done and point out questions as well as open implementation tasks arisen in the course of our writing.

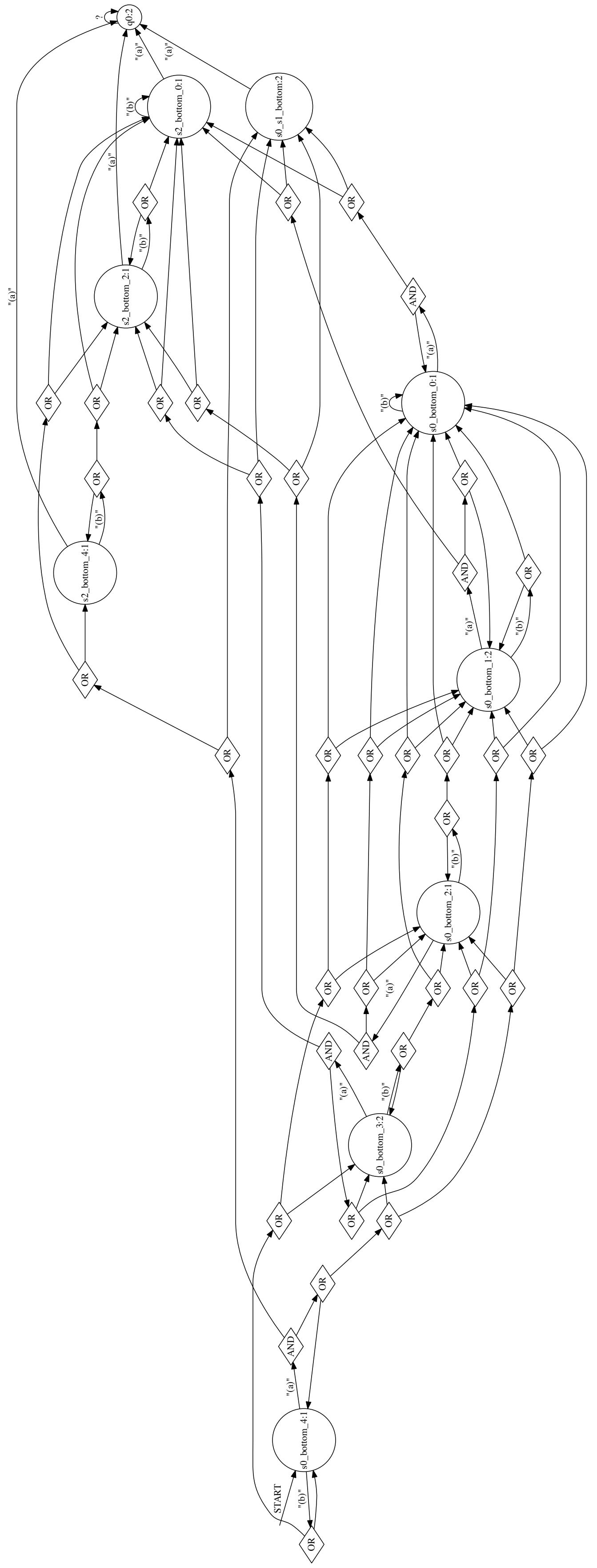

FIGURE 4.3: Graphical representation of the constructed Complement WAA rendered by RltlConv. We used the improved method described in 3.2.2, which implies a quadratic blowup in the number of states of the 2NBA.

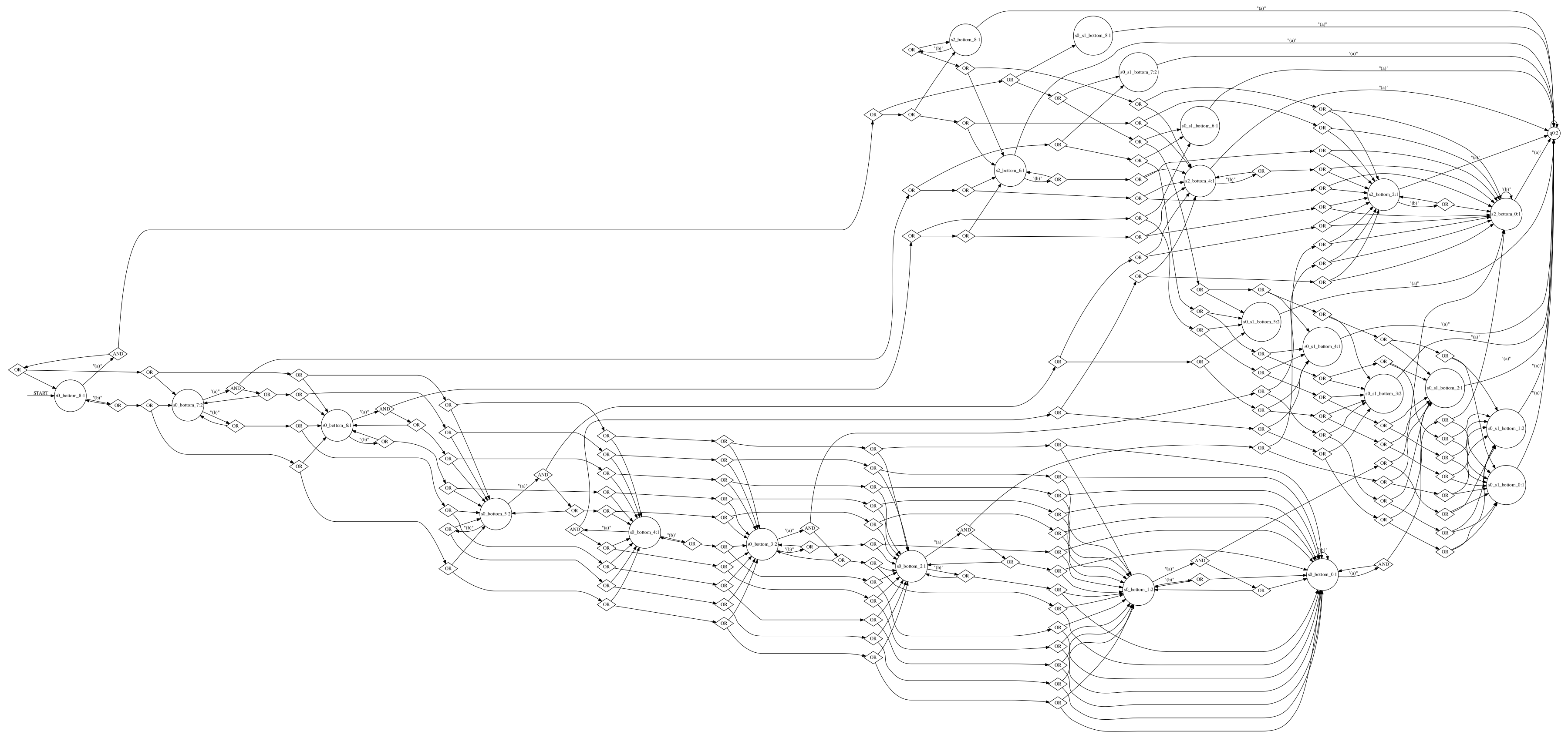

FIGURE 4.4: Graphical representation of the constructed Complement WAA rendered by RltlConv. We did not use the improved method described in 3.2.2, but used the intermediate step of an ABA instead. This implies a biquadratic blowup in the number of states of the original 2NBA. We can observe the layered structure of the Complement WAAs resulting from Kupferman's and Vardi's construction very clearly in this example.

Chapter 5

Conclusion and Outlook

In this thesis, we presented and implemented three different automata conversion methods. The first one described the construction of a language equivalent alternating Büchi automaton from a 2-way nondeterministic Büchi automaton. The other two methods described the complementation of ABAs as well as 2NBAs by means of weak alternating automata.

Although these conversions perform well for input automata of "reasonable size" and don't have huge blowups in the number of states, the complexity of the transitions in the resulting automata increases alarmingly. We will therefore need to investigate these conversions further and try to decrease the number of states as well as the complexity of the transitions.

One example, where we could start such an attempt of minimization, is the upper bound to the ranks presented in 3.2.1. In [6], Kupferman and Vardi show all vertices in an accepting run DAG to be of some rank $i \in [2n]$. They never mention this bound to be tight. In other words, it is unclear whether there exist accepting run DAGs for which G_{2n} is finite while G_{2n-1} isn't. For all examples we created in the course of this thesis, $n+1$ ranks were sufficient. If this bound should turn out to be tight, the number of states in the resulting Complement WAA could be halved.

Apart from these matters of efficiency, there are multiple features that have yet to be implemented in RltlConv. As we already mentioned before, there is no transformation from ω-regular expressions with past operators to 2NBAs. There is also no conversion of 2ABAs or 2A3PAs to NBAs, which would be helpful in the monitor construction based on pRLTL expressions.

Bibliography

[1] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime Verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14, 2011.

[2] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, January 1981.

[3] Christian Dax and Felix Klaedtke. Alternation Elimination by Complementation (Extended Abstract). In *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference, LPAR 2008, Doha, Qatar, November 22-27, 2008. Proceedings*, volume 5330 of *Lecture Notes in Computer Science*, pages 214–229. Springer, 2008.

[4] E. Allen Emerson and Charanjit S. Jutla. Tree Automata, Mu-Calculus and Determinacy (Extended Abstract). In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*, pages 368–377. IEEE Computer Society, 1991.

[5] Dexter Kozen. Automata and Computability. *Undergraduate Texts in Computer Science*. Springer, 1997.

[6] Orna Kupferman and Moshe Y. Vardi. Weak Alternating Automata Are Not That Weak. *ACM Trans. Comput. Log.*, 2(3):408–429, 2001.

[7] Martin Leucker and César Sánchez. Regular Linear-Time Temporal Logic. In *TIME 2010 - 17th International Symposium on Temporal Representation and Reasoning, Paris, France, 6-8 September 2010*, pages 3–5. IEEE Computer Society, 2010.

[8] Nancy G. Leveson and Clark Savage Turner. An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7):18–41, 1993.

[9] Satoru Miyano and Takeshi Hayashi. Alternating Finite Automata on Omega-Words. *Theor. Comput. Sci.*, 32:321–330, 1984.

[10] Madhavan Mukund. Finite-state automata on infinite inputs. In *Modern Applications of Automata Theory*, pages 45–78. 2012.

[11] Nir Piterman and Moshe Y. Vardi. From Bidirectionality to Alternation. *Theor. Comput. Sci.*, 295:295–321, 2003. (Preprint submitted to Elsevier Science, 21. November 2001).

[12] M. O. Rabin and D. Scott. Finite Automata and Their Decision Problems. *IBM J. Res. Dev.*, 3(2):114–125, April 1959.

[13] César Sánchez and Martin Leucker. Regular Linear Temporal Logic with Past. In *Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings*, volume 5944 of *Lecture Notes in Computer Science*, pages 295–311. Springer, 2010.

[14] J.C. Shepherdson. The Reduction of Two-Way Automata to One-Way Automata. *IBM Journal of Research and Development*, 3(2):198–200, April 1959.

[15] Moshe Y. Vardi. A Note on the Reduction of Two-Way Automata to One-Way Automata. *Inf. Process. Lett.*, 30(5):261–264, 1989.

List of Figures

List of Tables

Listings

Abbreviations

2ABA	**2**-way **A**lternating **B**üchi **A**utomaton
2A*k*PA	**2**-way **A**lternating ***k*-P**arity **A**utomaton
2NBA	**2**-way **N**ondeterministic **B**üchi **A**utomaton
2NFA	**2**-way **N**ondeterministic **F**inite **A**utomaton
ABA	**A**lternating **B**üchi **A**utomaton
ACA	**A**lternating **C**o-Büchi **A**utomaton
AFA	**A**lternating **F**inite **A**utomaton
A*k*PA	**A**lternating ***k*-P**arity **A**utomaton
DAG	**D**irected **A**cyclic **G**raph
DBA	**D**eterministic **B**üchi **A**utomton
DFA	**D**eterministic **F**inite **A**utomton
DMoore	**D**eterministic **Moore** Machine
FSA	**F**inite **S**tate **A**utomaton
LTL	**L**inear **T**emporal **L**ogic
NBA	**N**ondeterministic **B**üchi **A**utomaton
NFA	**N**ondeterministic **F**inite **A**utomaton
pRLTL	**R**egular **L**inear **T**emporal **L**ogic with **p**ast
RegEx	**Reg**ular **Ex**pression
RLTL	**R**egular **L**inear **T**emporal **L**ogic
WAA	**W**eak **A**lternating **A**utomaton