Masters thesis

Complementation of Büchi automata:
A survey and implementation

by
Anders Lindahl
Mattias Svensson

LITH-IDA-EX-04/008-SE

2004-02-06
Masters thesis

Complementation of Büchi automata:
A survey and implementation

by

Anders Lindahl
Mattias Svensson

LiTH-IDA-EX-04/008-SE

Supervisor and Examiner:
Ulf Nilsson, IDA
Abstract

This thesis is a survey of the field of languages over infinite sequences. There is active research going on in this field, during the last year several new results where published.

We investigate the language containment problem for infinite sequences, with focus on complementation of Büchi automata. Our main focus is on the approach with alternating automata by Kupferman & Vardi. The language containment problem has been proved to be in EXPSPACE. We identify some cases when we can avoid the exponential blow-up by taking advantage of properties of the input automaton.

Some of the algorithms we explain are also implemented in a Sicstus Prolog library.

Keywords: Infinite sequences, Alternating automata, Complementation, Omega-regular, Büchi automata.
# Contents

1 **Introduction** .......................... 1
  1.1 What is this? .......................... 1
  1.2 Why is it useful? ...................... 1
  1.3 Is it hard? ............................ 2
  1.4 Where does it come from? .......... 2
    1.4.1 A little about Temporal Logic .. 3
  1.5 How to read this thesis .......... 3

2 **Preliminaries** ........................ 5
  2.1 Languages ............................ 5
  2.2 Various automata .................... 6
    2.2.1 Büchi and co-Büchi automata .... 6
    2.2.2 Muller automata ................. 8
    2.2.3 Alternating automata .......... 12
    2.2.4 Weak alternating automata .... 15
  2.3 Other preliminaries ............... 15

3 **The simpler operations** .......... 17
  3.1 Union .................................. 17
  3.2 Intersection .......................... 19
  3.3 Emptiness .............................. 21
  3.4 Complement of deterministic Büchi . 22

4 **Complementation** .................... 25
  4.1 From Büchi to co-Büchi .......... 25
  4.2 Rank ................................. 27
    4.2.1 Minimal rank .................... 33
  4.3 From co-Büchi to WAA ............... 34
  4.4 From WAA back to Büchi .......... 36
  4.5 The composed algorithm .......... 38
  4.6 Lower bound of complementation ... 40
  4.7 Minimization .......................... 42
    4.7.1 Choosing the right algorithm ... 42
    4.7.2 Reducing the size of δ .......... 45
    4.7.3 Simulations of nondeterministic Büchi automata . 45
    4.7.4 Removal of SCC .................. 50
5 Safra's construction  53
  5.1 The subset construction .................. 53
  5.2 The marked subset construction .......... 54
      5.2.1 Levels .................................. 55
  5.3 Safra trees .................................. 55
      5.3.1 Soundness and completeness .......... 57
  5.4 Complexity ................................. 58
      5.4.1 How to reach optimal bound ............ 58

6 Implementation  61
  6.1 Design choices ............................ 61
      6.1.1 Native Prolog or C/Java? .............. 61
      6.1.2 Data structure for automata .......... 61
      6.1.3 The predicates ......................... 62
      6.1.4 nbw_print_dot/1 and nbw_print_dot/2 ... 64
  6.2 Verifying our implementation .............. 64
  6.3 Performance ............................... 64

7 Conclusion  65
  7.1 Summary .................................... 65
  7.2 The implementation ....................... 65
      7.2.1 Alternative approaches ............... 65
      7.2.2 Improvements .......................... 66
  7.3 Future work ................................ 66
      7.3.1 Minimization ......................... 66
      7.3.2 Transform the specification .......... 67

A The Sicstus Prolog library  69
Chapter 1

Introduction

1.1 What is this?

This is a thesis about language containment and complementation of Büchi automata. An automaton is a theoretical construction for describing some language, and by complementation we want to create the automaton that describes all words not in the language of the original automaton. When you read “language” in this thesis, you should not think of languages in the natural sense (like “Swedish” or “English”). Languages in this thesis are generally more predictable (some might say boring) well-defined sequences of characters like “all strings with an even number of $b$’s” or “every string that begins with $abba$”.

Languages can have different properties depending on what kind of automaton is needed to describe them. In this thesis we work with so called $\omega$-regular (“omega-regular”) languages. Their characteristic property is that they are made up of infinite sequences of characters.

1.2 Why is it useful?

$\omega$-regular languages can be used for describing behavior of reactive computer systems that are designed to run for “infinite” time, for example operating systems or cellphones. The description can be used to verify that the system indeed meets its specification (which we of course also describe with an $\omega$-regular language).

Assume that we have a system described by the $\omega$-regular language $A$ and its specification described by the language $B$. We want to know if:

$$A \subseteq B$$

That is, we want to know if every behaviour of $A$ is “in” $B$ and hereby consistent with the specification.

The relation above is equivalent to the following expression:

$$A \cap \overline{B} = \emptyset$$

We see that we can check language containment if we can compute intersection, complement and emptiness. Intersection and emptiness are rather easy.
Complement is another issue.

1.3 Is it hard?

Yes, indeed. The problem of computing the complement of any given Büchi automaton is proved to be an EXPSPACE problem. This means the space required for the complement automaton grows exponentially with the size of the original automaton. Let’s count a little: We assume that the complement of an automaton with $n$ states has $2^n$ states (obviously exponential).

<table>
<thead>
<tr>
<th>$n$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>170</td>
<td>$2^{170} \approx 10^{50}$</td>
</tr>
</tbody>
</table>

A 170 state automaton would require about $10^{50}$ states to describe its complement. $10^{50}$ is, by the way, approximately the number of atoms in the planet Earth. Doubtfully they are all available for a complementation calculation at the same time.

And, if that was not enough, all optimal complementation algorithms actually has a higher growth rate than $2^n$ in the worst case.

1.4 Where does it come from?

Automata theory for infinite words was introduced in 1960 by Büchi. He was studying decision problems for $SIS$, monadic second-order theory of one successor. Büchi showed that a $SIS$ formula corresponds to an $\omega$-regular language.

Many people have since come up with variations on Büchi’s automata definition. In this thesis we will make use of Muller automata [Muller, 1963], Rabin automata [Rabin, 1969], Streett automata [Streett, 1982] and alternating automata [Chandra et al., 1981] in addition to Büchi’s construction.

In Büchi’s original work he had an algorithm for computing the complement of an automaton, involving a $2^{2^{cn}}$ size blow-up in the worst case. This was improved to $2^{2^{cn}}$ by Sistla, Vardi & Wolper [Sistla et al., 1987]. The next year, Michel proved that $2^{O(n \log n)}$ was a lower bound for the worst case and it was not long before algorithms meeting this complexity was presented by Safra [Safra, 1988] and later by Klarlund [Klarlund, 1991].

In this thesis we focus on a complementation construction originally by Kupferman & Vardi [Kupferman and Vardi, 2000], involving translation to alternating automata.

The interested reader might want to take a look at Wolfgang Thomas chapter in *Handbook of Theoretical Computer Science* [Thomas, 1990], which contains the fundamentals of this field.
1.4.1 A little about Temporal Logic

To describe the behaviour of a reactive system, we have to be able to reason about time. A multi-tasking operating system might need properties like “it must never happen that two processes are in their critical regions at the same time” or “a process must enter its critical region sometime”. To formalize expressions like these, we use Propositional Linear Time Logic (PLTL). This logic adds temporal operators to the usual propositional connectives:

<table>
<thead>
<tr>
<th>Formula</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Fp$</td>
<td>sometimes $p$</td>
</tr>
<tr>
<td>$Gp$</td>
<td>always $p$</td>
</tr>
<tr>
<td>$Xp$</td>
<td>next time $p$</td>
</tr>
<tr>
<td>$pUq$</td>
<td>$p$ until $q$</td>
</tr>
</tbody>
</table>

There are similar operations for expressing history.

The interesting thing here is that a formula in PLTL can be transformed to a Büchi automaton with $2^{|p|}$ states (with $|p|$ being the length of the formula). If the automaton describes a non-empty language, then the formula is satisfiable.

Two important properties for process handling are mutual exclusion (two processes cannot be in critical regions at the same time) and liveness (a process will enter its critical region sometime). If we consider two processes with critical regions $CR_0$ and $CR_1$, the two properties can be expressed in PLTL as follows:

- Mutual exclusion: $G \neg (CR_0 \land CR_1)$
- Liveness: $F CR_0$

The corresponding Büchi automata are shown in Figure 1.1 and Figure 1.2. That they are both non-empty will be clear after reading Chapter 2.

We are not going to dig any deeper into PLTL or its transformation to Büchi automata. Through the rest of this thesis we will assume that we have meaningful automata, and focus on computing the constructions needed for checking language inclusion.

1.5 How to read this thesis

In the next chapter, we will present the theoretical background needed for the rest of the thesis. The reader who is already familiar with Büchi automata in various forms can safely skip this chapter or check back to it for reference later.
Then, chapter 3 goes through the simpler operations. These are useful to read and understand to get used to working with Büchi automata.

Chapter 4 is the main chapter, and devoted to the general complement of Büchi automata. The method described (and implemented) is from Kupferman & Vardi [Kupferman and Vardi, 2000], and it was chosen because it was the state of the art when we started to gather material. This chapter will also touch upon the complexity of complementation and some minimization techniques.

In Chapter 5 we describe Safra’s construction to translate a nondeterministic automaton into a deterministic one.

Chapter 3, 4 and 5 is rather independent and can be read in any order.

The last chapters are about the Prolog library (available in Appendix A) we have implemented and our conclusions.
Chapter 2

Preliminaries

In this chapter we will present the background needed for the rest of this thesis. A large part of it is definitions of various types of automata. We assume that the reader is familiar with DFA (deterministic finite automata) and NFA (non-deterministic finite automata) for regular languages of finite words. To learn about these subjects we recommend the book by Hopcroft, Motwani & Ullman [Hopcroft et al., 2001].

2.1 Languages

In formal languages theory for finite words, a language is defined as a set of words. A word is a finite sequence of letters taken from an alphabet $\Sigma$. An alphabet is a finite set of symbols.

In this thesis we are interested in languages of infinite sequences, which we will call $\omega$-sequences. The alphabet is defined as for finite words, and an $\omega$-sequence is an infinite sequence of letters from the alphabet. A set of $\omega$-sequences form a language, which we call an $\omega$-language.

We will restrict the thesis to only cover $\omega$-regular languages. This means that the language can be recognized by an automata without any additional memory, like push-down automata or Turing machines.

We will denote an $\omega$-sequence with $\alpha$, and a letter with $a$. By $\alpha_{m..n}$ we denote a finite sequence, starting with letter $\alpha_m$ and ending with letter $\alpha_n$. Letter $\alpha_0$ is the first letter of a sequence.

Definition 2.1 Let $\Sigma$ be an alphabet. An $\omega$-sequence $\alpha$ is an infinite sequence of letters belonging to $\Sigma$. We denote this $\alpha \in \Sigma^\omega$. An $\omega$-language is a set of $\omega$-sequences.

Definition 2.2 Given a regular language $\mathcal{L} \subseteq \Sigma^*$, the limit language $\text{lim}(\mathcal{L})$ is the set of $\omega$-sequences:

$$\{\alpha \in \Sigma^\omega \mid \exists n \in \mathbb{N} : \alpha_{0..n} \in \mathcal{L}\}$$

By $\exists^\omega x$ we mean "there exists an infinite number of $x$". Intuitively, $\text{lim}(\mathcal{L})$ is a language of $\omega$-sequences with an infinite number of prefixes from $\mathcal{L}$. We will use this for describing properties of automata over infinite sequences in terms of regular languages.
2.2 Various automata

There are several types of automata for describing \( \omega \)-regular languages. They all have a finite alphabet \( \Sigma \), a finite set of states \( Q \), a start state \( q_0 \), some function for moving between states \( \delta \) and some states or sets of states for the acceptance condition \( F \). The difference between the various automata lies in the interpretation and structure of \( F \) and \( \delta \).

2.2.1 Büchi and co-Büchi automata

**Definition 2.3** A Büchi automaton \( B \) is a five-tuple \( \langle \Sigma, Q, q_0, \delta, F \rangle \) where \( \Sigma \) is a finite alphabet, \( Q \) is a finite set of states, \( q_0 \in Q \) is the start state, \( \delta : Q \times \Sigma \rightarrow 2^Q \) is a transition function and \( F \subseteq Q \) is a set of accepting states.

We often want to reason about the states an automaton visits when reading some input. The following two definitions give us tools for this:

**Definition 2.4** A run \( \sigma_\alpha \) (or just \( \sigma \)) is the infinite sequence of states an automaton visits while reading the infinite input \( \alpha \).

Another way to write an infinite sequence of states is \( Q^\omega \).

**Definition 2.5** \( \text{inf}(\sigma) \) is the set of states the automaton visits infinitely often in a run \( \sigma \).

\( \text{inf} \) will be used as a tool for expressing acceptance conditions for automata.

**Definition 2.6** Let \( B = \langle \Sigma, Q, q_0, \delta, F \rangle \) be a Büchi automaton. An \( \omega \)-sequence \( \alpha \) is accepted by \( B \) iff there is a run \( \sigma_\alpha \) on \( \alpha \) such that \( F \cap \text{inf}(\sigma_\alpha) \neq \emptyset \).

**Definition 2.7** A co-Büchi automaton is defined as a Büchi automaton. The acceptance condition however is the opposite; an \( \omega \)-sequence \( \alpha \) is accepted by a co-Büchi automaton \( B \) iff there is a run \( \sigma \) on \( \alpha \) such that \( F \cap \text{inf}(\sigma) = \emptyset \).

**Definition 2.8** Consider an automaton \( \langle \Sigma, Q, q_0, \delta, F \rangle \). If \( \forall q \in Q, \forall a \in \Sigma : \delta(Q, a) \) is a singleton, the automaton is deterministic. Otherwise, it is non-deterministic.

The Büchi definition is almost the same as the definition of DFA or NFA in languages over finite strings. The difference is the acceptance condition; while a DFA/NFA accepts by ending up in one of the accept states, the Büchi automaton accepts by passing an accept state an infinite number of times while reading the infinite string. For the co-Büchi automaton, the acceptance condition is reversed. It will accept any string that passes the accept states only a finite number of times.

We will compare the various algorithms by the size of the resulting automaton. Hence, we need a definition of size:

**Definition 2.9** The size of an automaton \( B = \langle \Sigma, Q, q_0, \delta, F \rangle \) is \(|Q| \) (the number of states in \( Q \)) unless otherwise stated. It is usually denoted \( n \).
In some cases the size of $F$ or $\delta$ is also interesting as a measure of the space complexity of an automaton.

An important property of Büchi automata is that a non-deterministic Büchi automaton has strictly more expressive power than a deterministic one. This is shown by describing what languages a deterministic Büchi can recognize, and then demonstrating a language that cannot be deterministic.

**Theorem 2.1** A language $L \subseteq \Sigma^\omega$ is recognisable by a deterministic Büchi automaton iff $L$ is of the form $\lim(\mathcal{W})$ for some regular language $\mathcal{W} \subseteq \Sigma^*$. 

**Proof:** Let the DFA $A = (\Sigma, Q, q_0, \delta, F)$ recognize the language $\mathcal{W}$. We define the Büchi automaton $B = (\Sigma, Q, q_0, \delta, F)$ with the exact same definition as $A$. Let $B$ accept a input $a \in \Sigma^\omega$. This means that there is an accepting run $\sigma_\alpha$ of $B$ on input $a$. Thus $\sigma_\alpha$ must visit $F$ infinitely many times.

Let $a_0 \cdots a_k \cdots a_n$ be a prefix of $a$. For all $k$ there is some $n \geq k$ such that $a_0 \cdots a_k \cdots a_n$ reaches $F$. Since $B$ is defined as $A$, this path must also exist in $A$. The path is accepted by $A$, and hence $a_0 \cdots a_k \cdots a_n \in \mathcal{W}$. By the definition of limit language, $a \in \lim(\mathcal{W})$.

$\square$

**Lemma 2.1** The language $(a + b)^*b^\omega$ is not recognisable by any deterministic Büchi automaton.

**Proof:** The language $(a + b)^*b^\omega$ recognize every sequence with a finite number of $a$’s. Assume that there is a deterministic Büchi automaton that recognize $(a + b)^*b^\omega$. Then by Theorem 2.1 there must exist a $\mathcal{W} \subseteq \Sigma^*$ such that $(a + b)^*b^\omega = \lim(\mathcal{W})$. The sequence $b^\omega$ belongs to $\lim(\mathcal{W})$ where $\mathcal{W}$ is $b^n$ for some $n_1 \geq 0$. Fix this $n_1$ and the sequence $b^nab^\omega$ belong to $\lim(\mathcal{W})$ where $\mathcal{W}$ is $b^nab^n$ for some $n_2 \geq 0$. The word $b^nab^n a \cdots ab^n$ belongs to $\mathcal{W}$ for all $i \geq 0$ by similar reasoning. Hence this means that the sequence $b^nab^n a \cdots ab^n a \cdots$ belongs to $\lim(\mathcal{W})$. This is a contradiction since this sequence has an infinite number of $a$’s.

$\square$

The deterministic Büchi automaton in Figure 2.1 recognizes the language $L = (b^*a)^\omega$, that is all sequences of $a$’s and $b$’s with an infinite number of $a$’s.

![Figure 2.1: A deterministic Büchi automaton that recognize the language $(b^*a)^\omega$.](image-url)
Figure 2.2: A nondeterministic Büchi automaton that recognize the language $(a + b)^*b^\omega$.

The complement language is $\overline{L} = (a + b)^*b^\omega$, which accepts all strings with only a finite number of $a$'s. $\overline{L}$ is accepted by the non-deterministic automaton in Figure 2.2. By Lemma 2.1 $\overline{L}$ cannot be expressed by a deterministic Büchi, and hence the following two corollary:

**Corollary 2.1** Deterministic Büchi automata are not closed under complement.

**Corollary 2.2** Nondeterministic Büchi automata are strictly more expressive than deterministic Büchi automata.

### 2.2.2 Muller automata

**Definition 2.10** A Muller automaton $M$ is a five-tuple $(\Sigma, Q, q_0, \delta, F)$ where $\Sigma$ is a finite alphabet, $Q$ is a finite set of states, $q_0 \in Q$ is the start state, $\delta : Q \times \Sigma \rightarrow 2^Q$ is a transition function and $F = \{F_1, F_2, \ldots, F_k\}$ where $F_i \subseteq Q$ for $i \in \{1, 2, \ldots, k\}$ is an accepting set of states. An $\omega$-sequence $\alpha$ is accepted by a Muller automaton $M$ iff there is a run $\sigma_\alpha$ on $\alpha$ such that $\inf(\sigma_\alpha) \in F$ i.e. $\inf(\sigma_\alpha) = F_i$ for some $i \in \{1, 2, \ldots, k\}$.

Instead of just having to visit an accept state infinitely often as in a Büchi automaton, the Muller automaton accepts a string if it can reach one of the accepting sets $F_i \in F$ in a finite number of steps and then stay there visiting all states of $F_i$ infinitely often. An example Muller automaton can be seen in Figure 2.3.

**Theorem 2.2** [McNaughton, 1966] An $\omega$-language is $\omega$-regular iff it is Muller recognisable.

By theorem 2.2 each $\omega$-regular language can be described with a Muller automaton. We will now show that this also holds for nondeterministic Büchi:

**Theorem 2.3** Nondeterministic Büchi and Muller automata are equal in terms of expressive power.

We prove this by showing that a Büchi automaton can simulate a Muller automaton and vice versa.
Figure 2.3: An example Muller automaton accepting the language $a((ab^+ba^+)+(ba^+ab^+))^\omega$ with one acceptance set (illustrated with the dashed circle).

**Simulate Büchi with Muller:** This is the easier of the two simulations. Consider a Büchi automaton

$$B = (\Sigma, Q, \phi, \delta, F).$$

It accepts by passing some state $q \in F$ an infinite number of times. If the automaton interpreted as a Muller automaton is to accept the same string, its set of sets of acceptance states must all contain $q$. Hence, in the corresponding Muller automaton

$$M = (\Sigma, Q, q_0, \delta, F'),$$

$F'$ is defined:

$$F' = \{Q' \mid Q' \in 2^Q \text{ and } Q' \cap F \neq \emptyset\}$$

**Proposition 2.1** Let $B = (\Sigma, Q, q_0, \delta, F)$ be a Büchi automaton and let $M = (\Sigma, Q, q_0, \delta, F')$ be the Muller automaton constructed as above. If $B$ accepts the input $\alpha$ then $M$ also accepts $\alpha$.

**Proof:** If an input $\alpha$ is accepted by $B$, this corresponds to a unique run $\sigma_\alpha$ and $\inf(\sigma_\alpha)$ must contain at least one state $q \in F$. Since $\inf(\sigma_\alpha) \in 2^Q$ and $q \in \inf(\sigma_\alpha)$, $\inf(\sigma_\alpha) \in F'$ must hold and $M$ will accept $\alpha$.

**Proposition 2.2** Let $B = (\Sigma, Q, q_0, \delta, F)$ be a Büchi automaton and let $M = (\Sigma, Q, q_0, \delta, F')$ be the Muller automaton constructed as above. If $B$ rejects the input $\alpha$ then $M$ also rejects $\alpha$. 


Proof: If an input $\alpha$ is rejected by $B$, this means that it passes through all of the acceptance states $F$ only a finite number of times. Since every subset $F_j \in F'$ by definition contain some state $q \in F$, no $F_j$ can have all of its states visited an infinite number of times and $M$ will reject $\alpha$.

Simulate Muller with Büchi: The simulation of a Muller automaton with Büchi is less straight-forward. We must express the accepting sets in the Muller automaton by constructing new sets for the Büchi automaton. When the automaton reaches such a set, we must make sure that it stays within the set. Consider a Muller automaton

$$M = \langle \Sigma, Q, q_0, \delta, F \rangle$$

and an accepting run (infinite sequence of states) on said automaton

$$q_0 q_1 q_2 \cdots q_n q_{n+1} \cdots$$

After a finite time $n$, the automaton will enter one of the accepting states $F_j \in F$. For the Büchi, we construct a new state and an additional transition that it might nondeterministically choose:

$$\langle q_2, \emptyset, j \rangle \in \delta(q_1, a) \text{ if } q_2 \in \delta(q_1, a) \land q_2 \in F_j$$

A transition from $q_1$ to $q_2$ on some input $a$ might end up in the constructed state $\langle q_2, \emptyset, j \rangle$ where $j$ is the index of this acceptance set in $M$.

We now explain these constructed sets: Whenever we are in a state where we have the option to move into a state within a set $F_j \in F$, we also add the possibility to move into another set of states on the form $\langle p, P, j \rangle$. The “original” state is represented by $p$, $P$ is used for collecting states from $F_j$ we have visited so far and $j$ is the index of the acceptance set $F_j$ in $M$.

While moving from state to state inside $F_j$, we add the states we have visited to $P$. Hopefully we end up in a state $\langle p, F_j \setminus q, j \rangle$ with a possible transition from $p$ to $q$. This means we have found a path from $q$ through all states in $F_j$ and back to $q$ again, the acceptance condition of $M$. We add a transition from $\langle p, F_j \setminus q, j \rangle$ to $\langle q, \emptyset, j \rangle$ and a possible cycle for Büchi acceptance is constructed.

Formally, we construct a Büchi automaton

$$B = \langle \Sigma, Q', q_0, \delta', F' \rangle$$

where:

$$Q' = Q \cup \{Q \times 2^Q \times \{1, \ldots, |F|\}\}$$

$$F' = \{F \times \emptyset \times \{1, \ldots, |F|\}\}$$
2.2. VARIOUS AUTOMATA

The transition function $\delta'$ is defined as:

$$
q \in \delta'(p, a) \quad \text{if} \quad q \in \delta(p, a)
$$

$$
\langle q, \emptyset, j \rangle \in \delta'(p, a) \quad \text{if} \quad q \in F_j \text{ and } q \in \delta(p, a)
$$

$$
\langle q, P \cup \{q\}, j \rangle \in \delta'((p, P, j), a) \quad \text{if} \quad q \in \delta(p, a) \text{ and } P \cup \{q\} \subseteq F_j
$$

$$
\langle q, \emptyset, j \rangle \in \delta'((p, P, j), a) \quad \text{if} \quad q \in \delta(p, a) \text{ and } P \cup \{q\} = F_j
$$

**Proposition 2.3** Let $M = \langle \Sigma, Q, q_0, \delta, F \rangle$ be a Muller automaton and let $B = \langle \Sigma', Q', q_0', \delta', F' \rangle$ be the Büchi automaton constructed as above. If $M$ accepts the input $\alpha$ then $B$ also accepts $\alpha$.

**Proof:** If $M$ accepts an input $\alpha$ corresponding to a run $\sigma_\alpha$, $F_j = \inf(\sigma_\alpha)$ must hold for some $F_j$. There will be an accept state $\langle q, \emptyset, j \rangle$ in $B$ that corresponds to some state in $F_j$ and a possible transition to it (by definition of $F'$ and $\delta'$). The constructed states will form a cycle containing $\langle q, \emptyset, j \rangle$ (by definition of $\delta'$); $\langle q, \emptyset, j \rangle \in \inf(\sigma_\alpha)$ and $B$ accepts $\alpha$.

\[ \square \]

**Proposition 2.4** Let $M = \langle \Sigma, Q, q_0, \delta, F \rangle$ be a Muller automaton and let $B = \langle \Sigma', Q', q_0', \delta', F' \rangle$ be the Büchi automaton constructed as above. If $M$ rejects the input $\alpha$ then $B$ also rejects $\alpha$.

**Proof:** Assume an input $\alpha$ not accepted by $M$, corresponding to a run $\sigma_\alpha$. Then it must hold that $\forall F_j \subseteq F$, $F_j \neq \inf(\sigma_\alpha)$. If we move into a constructed set of states corresponding to some $F_j$, the only way to accept is to visit all of the states in $F_j$ an infinite number of times (by definition of $\delta'$). Hence, we can not move into the constructed states and since there are no other accept states, $B$ can not accept $\alpha$.

\[ \square \]

An example of this construction can be seen in Figure 2.4.

**Proof:** (of Theorem 2.3) Proposition 2.1 and Proposition 2.2 stated that Muller automaton can simulate a nondeterministic Büchi automaton. By Proposition 2.3 and Proposition 2.4 a Büchi automaton can be simulated by a Muller automaton. Hence, Muller automata and a nondeterministic Büchi automata have the same expressive power.

\[ \square \]

The simulation of Büchi automata with Muller does not remove nondeterminism. Determinism will however be preserved, the Muller automaton simulating a deterministic Büchi by the construction above will be deterministic. When going in the other direction, a Büchi automaton simulating a Muller will always be nondeterministic by the definition of the construction.

There are techniques to transform nondeterministic Büchi to deterministic Muller or Rabin automata, more on this in Chapter 5.
2.2.3 Alternating automata

We will now introduce alternating automata, a less restricted computation mode. This does not lead to an increase in expressive power regarding \( \omega \)-regular languages, since we have already seen that nondeterministic Büchi automata recognize every \( \omega \)-regular language. The advantage is that an alternating automaton is easy to complement.

The complex structure of alternating automata make operations that are simple for nondeterministic automata hard. For example, the emptiness problem is a PSPACE-complete problem for alternating automata while for nondeterministic automata it is checked in polynomial time. It is also more difficult for a programmer to keep track of whether the alternating automaton accepts an input. This means that we want to use the good complementation property of alternating automata and then transform it back to a nondeterministic automata. This is however also a hard problem.

**Definition 2.11** \( B^+(X) \) are the positive (negation-free) boolean formulas defined on the elements of \( X \) with the operators \( \land \) and \( \lor \) and the constants true and false.

**Definition 2.12** An Alternating Büchi Automaton \( A \) is a five-tuple

\[ \langle \Sigma, Q, q_0, \delta, F \rangle \]

\( \Sigma, Q \) and \( q_0 \) are defined as in ordinary Büchi automata. \( F \) is a condition that defines a subset of \( Q^\omega \), and the transition function is:

\[ \delta : Q \times \Sigma \rightarrow B^+(Q) \]
2.2. VARIOUS AUTOMATA

The δ function of an alternating automaton produces a formula over states of the automaton. A literal in the formula is true when the automaton is in that state, and false otherwise.

This definition is a bit more relaxed than the one found in Miyano & Hayashi report from 1984 [Miyano and Hayashi, 1984]. In their definition, each state is either existential or universal; each state is associated with either ∨ or ∧. The definition we use is from Kupferman & Vardi [Kupferman and Vardi, 2000].

In fact, ordinary deterministic and non-deterministic automata can be regarded as special cases of alternating automata. In the deterministic case, we add the constraint that δ(q, a) must be an atom, and in the non-deterministic case that δ(q, a) must be a possible empty disjunction.

What happens when δ(q, a) contains a conjunction, for example δ(q₁, a) = (q₁ ∧ q₂)? The informal meaning of this is "When standing in state q₁ and reading a, the automaton should move to state q₁ and q₂". Think of it as the automaton being in several states at the same time, or forking several copies of itself which all have to accept the input.

A positive boolean formula has a unique representation in Disjunctive Normal Form, DNF. We will sometimes use a set representation of the boolean formula.

**Definition 2.13** A set representation of a boolean formula on DNF is a set of sets of literals. The literal sets are interpreted as a conjunction, and the sets are interpreted as a disjunction. We represent false with ∅ and true with the set of all literals.

**Example 2.1** The formula (q₁ ∧ q₂) is represented by \{q₁, q₂\}. The formula (q₁ ∧ q₂) ∨ (q₃ ∧ q₄) is represented by \{q₁, q₂\}, \{q₃, q₄\}. The formula (q₁ ∧ q₃ ∧ q₄) ∨ (q₂ ∧ q₅ ∧ q₁) is represented by \{q₁, q₂, q₃\}, \{q₂, q₅, q₁\}.

From a boolean formula we can form a lattice, with the singleton sets being its generators. We will use the lattice structure in Chapter 4 when computing the dual to a transition function.

**Example 2.2** A lattice formed by the generators q₁, q₂, q₃ and q₄ is shown in Figure 2.5. The formula (q₁ ∧ q₂) ∨ q₃ ∨ q₄ is illustrated with circles. To see this, the generators \{q₁\} and \{q₂\} produce \{q₁, q₂\}, the generators \{q₃\} and \{q₄\} produce \{q₃\} respectively \{q₄\}. The whole formula is a disjunction between the produced sets, in set representation \{\{q₁, q₂\}, \{q₃\}\}. The formula from the previous example with set representation \{\{q₁, q₃, q₄\}, \{q₂, q₅, q₁\}\} is illustrated with polygons.

The concept of a run is a little more complex with alternating automata than the automata earlier described.

- With a deterministic automaton, there is only one sequence of states corresponding to a run on an input α.
- With a nondeterministic (or existential) automaton, there are several possible runs on the input α. At least one of these must be accepting to accept the input.
- With a universal automaton, there are several possible runs that all must be accepting.
With an alternating automaton, we allow both existential and universal transitions.

A run of an ordinary automaton corresponds to a computation tree of an alternating automaton. It has the following definition:

**Definition 2.14** We define $T(A, \alpha)$ to be the computation tree of alternating automata $A$ on the input $\alpha$, where $\alpha \in \Sigma^\omega$. The root vertex has label $q_0$ and if a vertex $v_1$ representing $m$ states in level $n$ has label $\bigwedge_{i=1}^{m} q_i$ and there is an edge for $a \in \Sigma$ to a vertex $v_2$ in level $n + 1$ then $v_2$ is labeled $\bigwedge_{i=1}^{m} \delta(q_i, a)$. We write the label for $v_2$ in DNF as $\bigvee_{i=1}^{r} C_i$ where each $C_i$ is a conjunction of generators of the label and $r$ is the number of disjunctions.

Figure 2.6 shows an example of a computation tree.

We define acceptance of an alternating automaton as follows:

**Definition 2.15** A run $\sigma$ of an alternating B"{u}chi automaton $A$ is an infinite sequence of states. A run is accepting iff it visit some state $q \in F$ infinitely many times. A branch in the computation tree $T(A, \alpha)$ is accepting iff every run in that branch is accepting. $A$ is accepting iff there is an accepting branch.
2.3. OTHER PRELIMINARIES

Figure 2.6: Computation tree for an alternating automaton.

Acceptance of an alternating co-Büchi automaton is similar, a run is accepting iff it visits all accepting states only a finite number of times.

### 2.2.4 Weak alternating automata

Weak alternating Büchi automata was introduced by Muller, Saoudi & Schupp [Muller et al., 1986]. They also showed that weak alternating Büchi automata are as expressive as (non-weak) alternating Büchi automata.

**Definition 2.16** An alternating automaton \(A = (\Sigma, Q, \phi, \delta, F)\) is weak if there is a partition of \(Q\) into disjoint sets, \(\{Q_1 \ldots Q_n\}\), such that either \(Q_i \subseteq F\) or \(Q_i \cap F = \emptyset\). There must also be a partial order on the collection of \(Q_i\)'s such that every transition from a state in \(Q_i\) must go to a state in a \(Q_j\) where \(i \geq j\).

In Chapter 4 we will present an algorithm for determining if an automaton is weak, and also make use of weakness in a more efficient complementation algorithm.

### 2.3 Other preliminaries

When reasoning about infinite trees in chapter 4, the following classical lemma by König will be used:

**Lemma 2.2** (König’s lemma) Any tree containing infinitely many nodes with each node having finitely many children, contains an infinite path.

In the algorithms for emptiness check and minimization, we want to reason about sets of nodes that are interconnected.

**Definition 2.17** A strongly connected component (SCC) is a subset \(S\) of a directed graph \(G\) such that for each pair of vertices \((v_1, v_2) \in S\) there is a path from \(v_1\) to \(v_2\) and from \(v_2\) to \(v_1\) inside \(S\).

In order to show \(n! = \Theta(n \log n)\) we use a technique called Stirling’s approximation. We start by taking the natural logarithm on the left-hand side and derive an approximation of this value.
\[
\ln n! = \ln 1 + \ln 2 + \cdots + \ln n = \sum_{k=1}^{n} \ln k \approx \int_1^n \ln x \, dx = [x \ln x - x]_1^n = n \ln n - n + 1
\]

There are better approximations of \(n!\), for example

\[
n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)
\]

in [Cormen et al., 2001] but the derived one will be sufficient here. The dominating factor of the right-hand side is \(n \ln n\) and we write this as \(\ln n^n\). We write both sides as exponents of \(e\), and get:

\[
n! = \Theta(n^n).
\]

Finally we multiply the right-hand side by \(2^{\log}\).

\[
n! = 2^{\log \Theta(n^n)} = 2^{\Theta(n \log n)}
\]
Chapter 3

The simpler operations

In this chapter we describe the simpler operations on Büchi automata: union, intersection, emptiness and the deterministic complement. We refer to them as "simple" since they only require polynomial time and a polynomial blow-up of the number of states. In the next chapter we will deal with the complement of nondeterministic automata, where we have an exponential blow-up.

The algorithms presented here does not try to minimise the resulting automaton in any way; the techniques in section 4.7.3 and 4.7.4 should be applicable.

3.1 Union

The union of two automata is constructed by pairing their states together. In each state of the resulting automaton, we know where we would have been if we where running the two automata separately. The transition function is defined such that every transition in the union corresponds to making the same transition in both automata and pairing together the results.

The automaton describing the union should accept whenever any of the original automata accept, hence the acceptance set of the union is each pair of states where one (or both) of the states are in its automaton's acceptance set.

Construction: Let $B_1 = \langle \Sigma, Q_1, q_1, \delta_1, F_1 \rangle$ and $B_2 = \langle \Sigma, Q_2, q_2, \delta_2, F_2 \rangle$ be two Büchi automata. We construct a Büchi automaton $B$ such that $L(B) = L(B_1) \cup L(B_2)$ as follows: Let $B = \langle \Sigma, Q, \langle q_1, q_2 \rangle, \delta, F \rangle$ where $Q \subseteq Q_1 \times Q_2$, $F \subseteq Q$ and $\delta : Q \times \Sigma \rightarrow 2^Q$ be the transition function defined by

$$\delta((p,q),a) = \{ (p',q') \mid p' \in \delta_1(p,a) \text{ and } q' \in \delta_2(q,a) \}$$

$$F = \{ (p,q) \mid p \in F_1 \text{ or } q \in F_2 \}$$

The union of the automata in Figure 3.1 and Figure 3.2 is shown in Figure 3.3. Some states have been omitted (for example $\langle q_1, q_2 \rangle$) because they are not reachable from the start state. In our implementation we avoid these by constructing the union inductively, starting at the start state and exploring only the states that we can reach.

The union construction above relies on the automata being completely defined.
Definition 3.1 Let $B = \langle \Sigma, Q, q_0, \delta, F \rangle$ be a Büchi automaton. $B$ is completely defined iff $\delta(q, a) \neq \emptyset$ for every $q \in Q$ and every $a \in \Sigma$.

If the automata are not completely defined when computing the union, it could result in a case when a transition cannot be constructed since one of the automata has no definition for the current configuration of state and letter. This could however easily be resolved either by redefining the union transition function to handle such cases or by simply adding a fail state and appropriate transitions to the automata.

From the definition, the size of the union of two automata $B_1$ and $B_2$ with size $n_1$ and $n_2$ would be $n_1n_2$. In practice, some of these states will be unreachable from the start state and the resulting automaton will be smaller than $n_1n_2$.

We will now prove that the construction is sound and complete.

Proposition 3.1 Let $B_1$ and $B_2$ be two completely defined Büchi automata with acceptance sets $F_1$ and $F_2$. Let $B$ be the automaton constructed by the union construction with the acceptance set $F$. If $B_1$ or $B_2$ accepts the input $\alpha$ then $B$ accepts the input $\alpha$.

Proof: Let $\sigma_\alpha$ be an accepting run on $B_1$. Then, there must be at least one $q_0 \in F_1$ such that $q_0 \in \inf(\sigma_\alpha)$. By the definition of $F$, there must be some element $(q_0, q_\delta) \in F$ that is visited an infinite number of times, and hence $B$ accepts $\alpha$. The case when $B_2$ is accepting $\alpha$ can be shown in the same way.

\[\Box\]

Proposition 3.2 Let $B_1$ and $B_2$ be two completely defined Büchi automata with acceptance sets $F_1$ and $F_2$. Let $B$ be the automaton constructed by the union construction with the acceptance set $F$. If $B$ accepts the input $\alpha$ then $B_1$ or $B_2$ accepts the input $\alpha$. 
3.2. \textit{INTERSECTION}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{buchi_automaton.png}
\caption{A deterministic Büchi automaton accepting the language \(a(a+ba)'^\omega\).}
\end{figure}

\textbf{Proof:} Let \(\sigma_\alpha\) be an accepting run on \(B\) on the input \(\alpha\). Then, there must be some state \((q_a, q_b) \in F\) such that \((q_a, q_b) \in \inf(\sigma_\alpha)\). By the construction of \(F\) it holds that \(q_a \in F_1\) or \(q_b \in F_2\), and hence \(\alpha\) must be accepted by \(B_1\) or \(B_2\).

\[\square\]

3.2 Intersection

The intersection construction is quite similar to the union construction. Each state is a three-tuple where the first two elements represent states of the original automata (like in the union construction) and the last one is used as a flag to make sure both automata would have accepted the input sequence. The flag can be interpreted as indicating which of the two automata has our attention at the moment while reading the input.

From start, we run on the first automaton. When we reach an accept state in that automaton, we switch attention to the second one. When it reaches an accept state we switch back to the first automaton. If the input sequence is in the language of both automata, this will result in an infinite number of switches back and forth.

\textbf{Construction:} Let \(B_1 = \langle \Sigma, Q_1, q_1, \delta_1, F_1 \rangle\) and \(B_2 = \langle \Sigma, Q_2, q_2, \delta_2, F_2 \rangle\) be two Büchi automata. We construct a Büchi automaton \(B\) such that \(\mathcal{L}(B) = \mathcal{L}(B_1) \cap \mathcal{L}(B_2)\) as follows: Let \(B = \langle \Sigma, Q, \langle q_1, q_2, 1 \rangle, \delta, F \rangle\) with \(Q \subseteq Q_1 \times Q_2 \times \{1, 2\}\) and \(F \subseteq Q\) where the transition function

\[
\delta : \{Q_1 \times Q_2 \times \{1, 2\}\} \times \Sigma \rightarrow \{Q_1 \times Q_2 \times \{1, 2\}\}
\]

is defined by
\[ \delta((p, q_1), a) = \{(p', q_1, 1) \mid p' \in \delta_1(p, a) \text{ and } q' \in \delta_2(q_1, a)\} \text{ if } p \notin F_1 \]
\[ \delta((p, q_1), a) = \{(p', q_2, 2) \mid p' \in \delta_1(p, a) \text{ and } q' \in \delta_2(q_1, a)\} \text{ if } p \in F_1 \]
\[ \delta((p, q_2), a) = \{(p', q_2, 1) \mid p' \in \delta_1(p, a) \text{ and } q' \in \delta_2(q_2, a)\} \text{ if } q \notin F_2 \]
\[ \delta((p, q_2), a) = \{(p', q_2, 2) \mid p' \in \delta_1(p, a) \text{ and } q' \in \delta_2(q_2, a)\} \text{ if } q \in F_2 \]

The acceptance set \( F \) is defined as follows:
\[ F = \{(q_1, q_2, 2) \mid q_1 \in Q_1 \text{ and } q_2 \in F_2\} \]

The intersection between the automaton in Figure 3.1 and Figure 3.2 by the intersection construction is in Figure 3.4. As in the union construction the automaton in Figure 3.4 is generated from the \( \delta \)-function and only reachable states are shown.

By the definition, the size of the intersection of two automata with size \( n \) and \( m \) will be \( 2nm \). As in the union case, some of these states will be unreachable from the start state and can be ignored.
3.3. **EMPTINESS**

We will now show that the intersection construction is sound and complete:

**Proposition 3.3** Let $B_1$ and $B_2$ be two Büchi automata. Let $B$ be the automaton constructed by the intersection construction. If $B_1$ and $B_2$ accept the input $\alpha$ then $B$ accepts the input $\alpha$.

**Proof:** Let $\sigma_1^\alpha$ be an accepting run of $B_1$ and $\sigma_2^\alpha$ be an accepting run of $B_2$ on the input $\alpha$. Let $\sigma_\alpha$ be the run from the intersection construction. Let $i_k$ be a position of $\sigma_\alpha$ where $\sigma_\alpha(i_k) = q_j$ and $q_j \in F_2$. Let $i_{k'}$ be a position such that $\sigma_\alpha(i_{k'}) = q_j$, $q_j \in F_1$ for some $i_{k'} \geq i_k$. Let $i_{k''}$ be the smallest position such $i_{k''} > i_{k'}$ and $\sigma_\alpha(i_{k''}) = q_j$, $q_j \in F_2$. The part of $\sigma_\alpha$ from $i_k$ to $i_{k''}$ corresponds to the sequence of states $\langle q_i, q_j, 1 \rangle \ldots \langle q_{i''}, q_j^{i''}, 2 \rangle$ where $\langle q_{i''}, q_{i''}, 2 \rangle \in F$. Since both $\sigma_1^\alpha$ and $\sigma_2^\alpha$ is accepting there are infinitely many positions $i_k$, $i_{k'}$ and $i_{k''}$, hence $\sigma_\alpha$ is an accepting run of $B$.

\[ \square \]

**Proposition 3.4** Let $B_1$ and $B_2$ be two Büchi automata with acceptance sets $F_1$ and $F_2$. Let $B$ be the automaton constructed by the intersection construction. If $B$ accepts the input $\alpha$ then $B_1$ and $B_2$ accept the input $\alpha$.

**Proof:** Let $\sigma_\alpha$ be an accepting run of $B$ on the input $\alpha$. Let $\langle q_i, q_j, s_i \rangle$ be a state in $B$. By the definition of acceptance, $\sigma_\alpha$ visit at least one state $\langle q_i, q_j, s_i \rangle \in F$ infinitely often. Then there exists a infinite sequence of position $i_0, i_1, \ldots$ on $\sigma_\alpha$ such that $q_i \in F_2$ and $s_{i_k} = 2$ which means that $B_2$ is accepting. $s_{i_k} = 2$ infinitely many times gives that there is a $s_i = 1$ and $q_i \in F_1$ for some $l > i_{k+1}$. Therefore $F_1$ must be visited infinitely many times, and thus $\sigma_\alpha$ is accepted by both $B_1$ and $B_2$.

\[ \square \]

3.3 **Emptiness**

Checking emptiness of a Büchi automaton is rather straight-forward. For the automaton to accept any string at all, there must exist a path from the start state $q_0$ to at least one accept state $q_F \in F$. The automaton must then pass the accept state $q_F$ infinitely often; $q_F$ must be part of a cycle, or more precisely an SCC.

Note that since we only care about existence of a language for the automaton, the labels on the edges of the automaton have no meaning and we can safely regard two (or more) transitions from one state to another reading different letters as a single edge in a directed graph.

Decomposing a graph into its strongly connected components can be done in linear time [Cormen et al., 2001]. Checking an automaton for emptiness could be done as follows:

- Regard the automaton as a directed graph $G = (V, E)$.
- Calculate the set $Q_F$ of accept states reachable from the start state.
- For each $q \in Q_F$, check if it is member of an SCC. If it is, the automaton has a non-empty language. Return false.
If no reachable accept states is a member in an SCC, the automaton has an empty language. Return true.

3.4 Complement of deterministic Büchi

Finding the complement of a deterministic Büchi automaton is also a straightforward construction. The complement automaton has two parts, a 0-part (with states on the form \( \langle q, 0 \rangle \)) and a 1-part (with states on the form \( \langle q, 1 \rangle \)). There are transitions from the 0-part to the 1-part, but not the other way around.

The 0-part is a copy of the original automaton, with two exceptions: The accept states from the original automaton are changed to ordinary states in the 0-part, and if a transition in the original automaton leads to a non-accepting state, a nondeterministic transition to the corresponding state in the 1-part is added.

The 1-part is also a copy of the original automaton, with one exception: There are no states representing the accept states of the original automaton. Instead, all of the states in the 1-part are accept states.

It is easy to see that this indeed describes the complement. At some time the automaton will ‘guess’ that it is time to move into the 1-part to accept the sequence. Since the states that where accepting in the original automaton only exist in the 0-part, they are visited only a finite number of times if the complement accepts.

Construction: From a deterministic Büchi automaton \( B = (\Sigma, Q, q_0, \delta, F) \) there is a nondeterministic Büchi automaton \( \overline{B} = (\Sigma, \overline{Q}, \overline{q}_0, \overline{\delta}, \overline{F}) \) as follows:

\[
\overline{Q} = \{ Q \times \{0\} \} \cup \{ Q \setminus F \times \{1\} \}
\]

\[
\overline{q}_0 = \langle q_0, 0 \rangle
\]

\( \overline{\delta} : \overline{Q} \times \Sigma \to 2^{\overline{Q}} \) is defined (with \( q \in Q \) and \( a \in \Sigma \)) as follows:

\[
\overline{\delta}(\langle q, 0 \rangle, a) = \begin{cases} 
\{ \langle \delta(q, a), 0 \rangle \} & \text{if } \delta(q, a) \in F \\
\{ \langle \delta(q, a), 0 \rangle, \langle \delta(q, a), 1 \rangle \} & \text{if } \delta(q, a) \notin F
\end{cases}
\]

\( \overline{\delta}(\langle q, 1 \rangle, a) = \{ \langle \delta(q, a), 1 \rangle \} \), if \( \delta(q, a) \notin F \)

\[\overline{F} = \{ Q \setminus F \times \{1\} \}\]

A deterministic Büchi automaton that accepts the language \( a^\omega \) is shown in Figure 3.1. From the complement construction we get the automaton in Figure 3.5 that accepts the language \( \Sigma^\omega \setminus a^\omega \). The automaton in the figure is generated from its \( \delta \)-function.

The construction has a linear blow-up in states, \( \overline{B} \) has \( 2n - |F| \) states if \( B \) had \( n \) states. We now show that the construction is sound and complete.

Proposition 3.5 Let \( \overline{B} = (\Sigma, \overline{Q}, \overline{q}_0, \overline{\delta}, \overline{F}) \) be the nondeterministic Büchi automaton constructed with the complement construction above from the deterministic Büchi automaton \( B = (\Sigma, Q, q_0, \delta, F) \). If \( B \) reject an input \( \alpha \in \Sigma^\omega \) then \( \overline{B} \) accepts \( \alpha \).
3.4. COMPLEMENT OF DETERMINISTIC BÜCHI

Proof: Let \( \sigma_\alpha = q_0 q_1 \cdots \) be a rejecting run of \( B \) on the input \( \alpha \). \( \sigma_\alpha \) will not visit \( F \) infinitely many times. There must be some finite \( k \in \mathbb{N} \) such that \( \forall i > k, q_i \notin F \). This corresponds to a unique run \( \langle q_0, 0 \rangle \langle q_1, 0 \rangle \cdots \langle q_k, 0 \rangle \langle q_{k+1}, 1 \rangle \langle q_{k+2}, 1 \rangle \cdots \) of \( \overline{B} \). This run visit infinitely many states of the form \( \langle q, 1 \rangle \) so this run is accepted by \( \overline{B} \).

\[ \square \]

Proposition 3.6 Let \( \overline{B} = \langle \Sigma, \overline{Q}, \overline{q}_0, \overline{\delta}, \overline{F} \rangle \) be the nondeterministic B"uchi automaton constructed with the complement construction above from the deterministic B"uchi automaton \( B = \langle \Sigma, Q, q_0, \delta, F \rangle \). If \( \overline{B} \) accepts an input \( \alpha \in \Sigma^\omega \) then \( B \) rejects \( \alpha \).

Proof: Let \( \sigma_\alpha \) be an accepting run of \( \overline{B} \) on the input \( \alpha \). \( \overline{F} \) is visited infinitely many times and by the construction of \( \overline{F} \) so is \( (Q \setminus F) \times \{1\} \). Since the construction does not allow any transitions from states of the form \( \langle q, 1 \rangle \) back to states like \( \langle q, 0 \rangle \), \( \sigma_\alpha \) must be on the form \( \langle q_0, 0 \rangle \langle q_1, 0 \rangle \cdots \langle q_k, 0 \rangle \langle q_{k+1}, 1 \rangle \langle q_{k+2}, 1 \rangle \cdots \). The corresponding unique run on \( B \) on the input \( \alpha \) is then \( q_0 q_1 \cdots q_k q_{k+1} q_{k+2} \cdots \). This run is not accepting. To see this, \( q_{k+1} \notin F \) because by the construction of \( \overline{F} \) the only way to have \( \langle q_{k+1}, 1 \rangle \) is that \( q_{k+1} \notin F \). Similar reasoning holds for all \( k+i \) where \( i \geq 1 \), so it holds for every \( q_{k+i} \) that \( q_{k+i} \notin F \). This means that \( F \) will not been visited infinitely many times so \( B \) will reject the input \( \alpha \).

\[ \square \]
Figure 3.5: The nondeterministic Büchi automaton describing the complement of the automaton in Figure 3.1.
Chapter 4

Complementation

In this chapter we will show a method for constructing the complement of non-deterministic Büchi automata. We will first show a construction in three steps to complement a nondeterministic automata. In the first step we interpret the nondeterministic automata as an alternating automata and use the fact that complementing an alternating automata is easy. The second step is to remove the infinite representation of the computation of an alternating automata. In the third step we will remove alternation. These three step will then be collapsed into a single algorithm.

We will also show some techniques for minimizing Büchi automata.

4.1 From Büchi to co-Büchi

Kupferman & Vardi [Kupferman and Vardi, 2000] stated that a Büchi automaton can be converted to an alternating co-Büchi automaton by obtaining the dual of the δ function. The foundation of this result is however by Muller & Schupp [Muller and Schupp, 1987]. They show that there is a dual alternating automaton that accepts the complement language to an alternating automaton.

Muller & Schupp work with alternating automata on infinite trees. Their construction is general and can adress other structures like pushdown automata and Turing machines in a uniform manner. In this thesis we only look at finite state automata over infinite sequences, so we will give a less general definition of an alternating automaton.

The dual of a positive boolean formula is obtained by switching every disjunction to a conjunction, every conjunction to a disjunction, true to false and false to true. This dual formula is not on DNF, but Conjunctive Normal Form, CNF, and can be transformed back by using the Distributive Laws.

To keep the formula on DNF, we use a lattice structure to dualize a formula. To dualize the formula $\bigvee_i C_i$ where each $C_i$ is a conjunction of generators, we choose a generator from each $C_i$ and let $S_j$ be a conjunction of those generators. The dual formula is $\bigvee_j S_j$ where the different $S_j$ are the minimal choice sets of $C_i$. By using minimal choice sets we get no redundant terms in the formula. It is easy to see that the dual formula is on DNF.

Example 4.1 In figure 2.5 the formula $(q_1 \land q_2) \lor q_3 \lor q_4$ is indicated with circles. From $(q_1 \land q_3) \lor q_4$ we choose the generator $q_1$. The sets $q_3$ and $q_4$ are
already generators, so we have only one choice for each of them. These three
generators form the set $S_1$.

From the first set we can instead choose $q_2$. This together with $q_3$ and $q_4$
gives us the set $S_2$. $S_1$ and $S_2$ are the two possible ways to choose generators
from the formula, so $\bigvee_j S_j = (q_1 \land q_3 \land q_4) \lor (q_2 \land q_3 \land q_4)$ is the dual formula.
In figure 2.5 this formula is illustrated with polygons.

Now we can define the dual automaton:

**Definition 4.1** Let $A = \langle \Sigma, Q, q_0, \delta, F \rangle$ be an alternating automaton. The
dual automaton is $\tilde{A} = \langle \Sigma, Q, q_0, \delta, \bar{F} \rangle$ where $\delta$ is the function obtained by
dualizing $\delta$ and $\bar{F} = Q \setminus F$ is the complement of $F$.

**Theorem 4.1** [Muller and Schupp, 1987] Let $A = \langle \Sigma, Q, q_0, \delta, F \rangle$ be an alternating Büchi automaton. Then the dual automaton $\tilde{A} = \langle \Sigma, Q, q_0, \tilde{\delta}, \bar{F} \rangle$ accepts the complement of the language accepted by $A$.

To prove Theorem 4.1 we will first show that each level in the computation
tree of the alternating automaton is the dual to same level in the dual automaton. The lemmas and proofs in this section are due to Muller & Schupp
[Muller and Schupp, 1987]. We are not interested in the case with infinite trees
(which corresponds to branching time logic) as input. Hence, we have simplified
the proofs. We will need the two following lemmas and a definition of the total
expression for a level in the computation tree.

**Definition 4.2** Let $T(A, \alpha)$ be the computation tree of an alternating automaton $A$ on input $\alpha$. Let $n \geq 0$ be a level of $T(A, \alpha)$ and let $C_1, \ldots, C_m$ be
the labels of the vertices at level $n$. Then the **total expression** of level $n$ is
$e_n = \bigvee_{i=1}^m C_i$.

**Lemma 4.1** Let $T(A, \alpha)$ and $T(\tilde{A}, \alpha)$ be the computation trees of $A$ and $\tilde{A}$ on input $\alpha$. If $e_n$ is the total expression on DNF of level $n$ in $A$ then $\tilde{e}_n$ is the total expression on DNF of level $n$ in $\tilde{A}$.

**Proof:** Proof by induction on $n$. For $n = 0$ the definition of computation trees
says that the root is $q_0$ and on DNF. Since that $e_0 = q_0 = \tilde{e}_0$, $e_0$ and $\tilde{e}_0$ are
also on DNF. Assume that the Lemma holds for $n$. Then, $\delta(e_n) = e_{n+1}$ and
$\tilde{e}_{n+1}$ is on DNF by the definition of computation trees. From the same
definition it follows that $\tilde{\delta}(\tilde{e}_n) = \tilde{e}_{n+1}$ and $\tilde{e}_{n+1}$ is on DNF.

**Lemma 4.2** Let $T(A, \alpha)$ and $T(\tilde{A}, \alpha)$ be the computation trees of $A$ and $\tilde{A}$ on input $\alpha$. For $n \geq 0$, the terms labelling the vertices of level $n$ in $T(A, \alpha)$ are
exactly the minimal choice sets of the vertices labelling level $n$ in $T(\tilde{A}, \alpha)$.

**Proof:** Let $e_n = \bigvee_i C_i$ be the total expression of level $n$ in $T(A, \alpha)$ and
$\tilde{e}_n = \bigvee_j S_j$ be the total expression of level $n$ in $T(\tilde{A}, \alpha)$. By Lemma 4.1 $\tilde{e}_n$ is
dual to $e_n$ and both are on DNF. Thus $S_j$ are exactly the minimal choice sets
of generators belonging to $C_i$.
4.2. RANK

For finite words this would be sufficient since every level of the respective computation trees is the dual of the other. This is however not always true for infinite sequences. To be sure, the acceptance condition $F$ must be a Borel subset of $Q^\omega$. Borel sets are constructed by the following definition.

**Definition 4.3** The class $B$ of Borel sets in Euclidean $\mathbb{R}^n$ is the smallest collection of sets that includes the open and closed sets such that if $E_1, E_2, \ldots$ are in $B$, then so are $\bigcup_{i=1}^{\infty} E_i, \bigcap_{i=1}^{\infty} E_i$ and $\mathbb{R}^n \setminus E$.

The set of the rational numbers is a Borel set. The set $Q^\omega$ is countable and therefore it is a one-to-one mapping between the rational numbers and $Q^\omega$. Thus $Q^\omega$ must be a Borel set and $F$ is a Borel subset of $Q^\omega$.

In an infinite game with perfect information, the game is said to be determined if one player has a winning strategy. Martin [Martin, 1975, Martin, 1985] proved that there is exactly one winner when the winning condition is Borel. If we see the computation tree as a game we have that either the original automaton or the dual, not both nor none, has a winning strategy if the acceptance condition is Borel.

We are now ready to prove Theorem 4.1.

**Proof:** (of Theorem 4.1) The acceptance condition is Borel, Martin proves that there is exactly one accepting automaton of $A$ and the dual $\bar{A}$ if the acceptance is Borel. By Lemma 4.1 and Lemma 4.2 $A$ accepts the complement language of $A$.

\[ \square \]

In practice it is difficult to construct $\bar{F}$, since it would involve checking if an infinite sequence belongs to another. To avoid this, Kupferman & Vardi proposed to instead construct an alternating co-Büchi automaton describing the complement:

**Theorem 4.2** [Kupferman and Vardi, 2000] For an alternating Büchi automaton $A = (\Sigma, Q, q_0, \delta, F)$ the alternating co-Büchi automaton $\bar{A} = (\Sigma, Q, q_0, \delta, F)$ satisfies $L(\bar{A}) = \Sigma^\omega \setminus L(A)$.

The following proof is new for this thesis:

**Proof:** From Theorem 4.1 we see that if an alternating automaton accepts $L(A)$ the dual automaton accepts $L(\bar{A}) = \Sigma^\omega \setminus L(A)$. The definition of co-Büchi automaton says that it has an acceptance condition $\bar{F} = \Sigma^\omega \setminus F$ and accepts the same language as the corresponding Büchi automaton. If we interpret $\bar{A}$ as an alternating co-Büchi automaton, $\bar{A} = (\Sigma, Q, q_0, \bar{\delta}, \bar{F})$, it will accept $\Sigma^\omega \setminus L(A)$. Since $\bar{F} = F$, we are done.

\[ \square \]

4.2 Rank

In the previous section we showed how to translate a Büchi automaton to an alternating co-Büchi automaton. In the next section we will show how to translate
it into a weak alternating automaton. To make the automaton weak, we must construct a partial order over its states (see Definition 2.16). Most of the work in this section is by Kupferman & Vardi [Kupferman and Vardi, 2000], unless otherwise stated.

Since the alternating co-Büchi automaton is constructed by computing the dual of a nondeterministic (existential) Büchi, it is always universal. This is because a nondeterministic (existential) Büchi automaton only have deterministic and nondeterministic transitions represented by disjunctions. The dual of a deterministic transition is the same transition. The dual of a nondeterministic transition is done by switching the disjunctions to conjunctions forming a universal transition. Hence, the alternating co-Büchi can only have deterministic and universal transitions and is therefore a universal co-Büchi automaton. This fact will be used in some proofs.

**Definition 4.4** A run $\sigma_{\alpha}$ is **memoryless** iff $\delta(q, a)$ (with $q \in Q$ and $a \in \Sigma$), is independent of what has been read earlier.

**Theorem 4.3** [Emerson and Jutla, 1991] If an alternating co-Büchi automaton $A$ accepts an input $\alpha$, then there exists a memoryless accepting run of $A$ on $\alpha$.

Actually Emerson & Jutla show a more general result for infinite tree automata. The important property of the theorem for this section is that from a vertex in the run tree of $A$ on the input $a \in \Sigma$ there will be a transition to the same vertices, independent of what has been read earlier. This means that in every level of the run tree there cannot be two equivalent vertices, therefore the width of any level is bound to the number of states $n$.

The run of an alternating (co-)Büchi automaton will be represented with an infinite $DAG$ (Directed Acyclic Graph). It is constructed as follows:

**Construction:** For a computation tree $T(U, \alpha)$ of a universal co-Büchi automaton $U$ and an input $\alpha$, we create an infinite $DAG$ $G_{\tau} = (V, E)$, where

- $V \subseteq Q \times \mathbb{N}$ is such that $(q, l) \in V$ iff there exists a vertex $v \in T(U, \alpha)$ with label $q$ on level $l$.
- $E \subseteq \bigcup_{l \geq 0} (Q \times \{l\}) \times (Q \times \{l + 1\})$ is such that $(q, l) \in E$ is such that $E((q, l), (q', l + 1))$ iff there exists a vertex $(q, l) \in T(U, \alpha)$ and $q' \in \delta(q, a)$ for some $a \in \Sigma$.

In order to define the ranking function, we need to cut the infinite DAG into a finite number of partitions. We will make use of the following three properties for vertices:

**Definition 4.5** In a (possibly finite) $DAG$ $G \subseteq G_{\tau}$, we say a vertex $(q, l)$ is an **F-vertex** iff $q \in F$. A vertex $(q, l)$ is **finite** in $G$ if only finitely many vertices in $G$ are reachable from $(q, l)$. We say that a vertex $(q, l)$ is **F-free** in $G$ if all the vertices in $G$ that are reachable from $(q, l)$ are not F-vertices and $(q, l)$ is not an F-vertex.

With help from the definitions of finite and F-free we will define a sequence of subgraphs of $G_{\tau}$.

**Definition 4.6** Given an accepting computation tree $T(U, \alpha)$ and its corresponding $DAG$ $G_{\tau}$, we define a sequence $G_{0} \supseteq G_{1} \supseteq G_{2} \supseteq \ldots$ of $DAG$s inductively as follows:
4.2. RANK

\begin{figure}
\centering
\begin{tikzpicture}
  \node[draw, shape=circle] (q0) at (0,0) {$q_0$};
  \node[draw, shape=circle] (q1) at (4,0) {$q_1$};
  \node[draw, shape=circle] (q2) at (2,-3) {$q_2$};
  \draw[->] (q0) -- node[above] {a} (q1);
  \draw[->] (q1) -- node[below] {a} (q2);
  \draw[->] (q2) -- node[below] {a,b} (q0);
\end{tikzpicture}
\caption{A universal co-Büchi automaton that accepts the language $a^*b(a+b)^\omega$.}
\end{figure}

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

**Example 4.2** From the universal co-Büchi automaton in Figure 4.1, the DAG $G_\ast$ is constructed. By the definition of the DAG sequence, we have $G_r = G_0$. The first eight levels of $G_0$ are shown in Figure 4.2. There are no finite vertices in $G_0$, because every vertex can reach some vertex labeled $(q_2, l)$ and there is an infinite sequence of these. Hence, $G_1 = G_0$.

In $G_1$, there are no edges from $(q_2, l)$ back to the $F$-vertices $(q_1, l)$. This means the vertices $(q_2, l)$ are $F$-free. By the definition, $G_2$ is then $G_1 \setminus \{ (q_2, l) \}$.

In $G_2$, the vertices $(q_1, l)$ must be finite (remember that we start from a co-Büchi and that the run must be accepting, hence we can only visit the accept state $q_1$ a finite number of times). Since $(q_0, l)$ can only reach a finite sequence it is also finite. $G_3$ is thus constructed by removing the finite vertices and we end up with an empty DAG.

We are going to use this sequence to define the rank function. The function must have a finite codomain, so the size of the DAGs in this sequence must be shrinking and at some iteration be finite. This is shown by the following lemma:

**Lemma 4.3** [Gurumurthy et al., 2003] For every $i \geq 0$ and for every level $l \geq 1$, there are at most $n - (|F| + i)$ vertices that are not $F$-vertices of the form $(q, l)$ in $G_{2(n-|F|+i)}$.

We have formulated a shorter version of the original proof:

**Proof:** Inductive proof on $i$. On every level in $G_r$ there can be at most $n$ vertices and $n - |F|$ vertices that are not $F$-vertices, thus the base step holds.
CHAPTER 4. COMPLEMENTATION

In the inductive step we need to show that $G_{2i+2}$ is strictly smaller than $G_{2i}$. We distinguish between the two cases when $G_{2i}$ is finite and when it is infinite. When $G_{2i}$ is finite, then by the definition all vertices will be removed when constructing $G_{2i+1}$. Thus, $G_{2i+1}$ and $G_{2i+2}$ are empty.

When $G_{2i}$ is infinite, the construction removes only finite vertices and $G_{2i+1}$ will still be infinite. By Königs Lemma (Lemma 2.2), there must be an infinite path in $G_{2i}$, thus there must exist an $F$-free vertex in $G_{2i+1}$ to hold this infinite path.

When constructing $G_{2i+2}$ this infinite path will by the definition be removed. Thus, $G_{2i+2}$ is strictly smaller than $G_{2i}$.

The following corollary follows from the lemma above. The proof however is new for this thesis.

**Corollary 4.1** [Gurumurthy et al., 2003] $G_{2(n-|F|)+1}$ is empty.

**Proof:** If we set $i = n - |F|$ in Lemma 4.3 we see that $G_{2(n-|F|)}$ at most has $n - (|F| + (n - |F|)) = 0$ vertices that are not $F$-vertices of the form $\langle q, l' \rangle$ where $l' \geq l_{n-|F|+1}$. There can only be a sequence of $F$-vertices left in

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{dag_sequence.png}
\caption{The sequence of infinite DAGs from the automaton in Figure 4.1 (the first eight levels).}
\end{figure}
$G_{2(n-|F|)}$. Since the DAG $G_r$ is constructed from an accepting run, it can only visit finitely many $F$-vertices. Thus, the sequence of $F$-vertices is finite, $G_{2(n-|F|)}$ is finite and $G_{2(n-|F|)+1}$ is empty.

Now we are ready to define the rank function:

**Definition 4.7** We define rank of a vertex $\langle q, l \rangle$, denoted $\text{rank}(q, l)$, to be:

$$\text{rank}(q, l) = \begin{cases} 2i & \text{if } \langle q, l \rangle \text{ is finite in } G_{2i} \\ 2i + 1 & \text{if } \langle q, l \rangle \text{ is } F\text{-free in } G_{2i+1} \end{cases}$$

From Lemma 4.3 there is a maximal rank that any vertex can get, namely $2(n - |F|)$. Every vertex in $G_r$ has exactly one rank. This means that there is a function that gives each vertex a value between 0 and $2(n - |F|)$.

**Example 4.3** We continue Example 4.2. The vertices $\langle q_2, l \rangle$ are $F$-free in $G_1$ and get rank 1. The vertices $\langle q_1, l \rangle$ and the vertex $\langle q_0, 0 \rangle$ are finite in $G_2$ and get rank 2. Notice that the rank of the $F$-vertices $\langle q_1, l \rangle$ is even.

The following lemmas prove our claims about rank formally.

**Lemma 4.4** For every vertex $\langle q, l \rangle$ in $G_r$ and rank $i$, $0 \leq i \leq 2(n - |F|)$, if $\langle q, l \rangle \notin G_i$ then $\text{rank}(q, l) < i$.

**Proof:** We prove this lemma by induction on $i$. The base case $i = 0$ holds since $G_0 = G_r$ by definition. Under the assumption that the lemma holds for $i$, we distinguish between the cases when $i + 1$ is odd and $i + 1$ is even:

- When $i + 1$ is even, consider a vertex $\langle q, l \rangle \notin G_{i+1}$. If $\langle q, l \rangle \notin G_i$, the lemma holds by the assumption. Else, if $\langle q, l \rangle \in G_i$ then $\langle q, l \rangle$ is $F$-free in $G_i$. This follows from the construction of the sequence of DAGs. Since $\langle q, l \rangle$ is in $G_i$ where $i$ is odd and not in $G_{i+1}$ it must be $F$-free. If $\langle q, l \rangle$ is a $F$-free vertex in $G_i$ then (by Definition 4.7) $\text{rank}(q, l) = i$.

- In the case of $i + 1$ is odd, consider a vertex $\langle q, l \rangle \notin G_{i+1}$. If $\langle q, l \rangle \notin G_i$ the lemma holds by the assumption. Otherwise $\langle q, l \rangle \in G_i$, and from the construction of DAGs $\langle q, l \rangle$ is finite in $G_i$. From the definition of $\text{rank}$ it follows that $\text{rank}(q, l) = i$.

**Lemma 4.5** For every two vertices $\langle q, l \rangle$ and $\langle q', l' \rangle$ in $G_r$, if $\langle q', l' \rangle$ is reachable from $\langle q, l \rangle$ then $\text{rank}(q', l') \leq \text{rank}(q, l)$.

**Proof:** Assume that $\text{rank}(q, l) = i$. We distinguish between the cases where $i$ is odd and where $i$ is even.

- When $i$ is even, the vertex $\langle q, l \rangle$ is finite in $G_i$ by the definition of rank. If $\langle q', l' \rangle \notin G_i$ then, by Lemma 4.4, $\text{rank}(q', l')$ is at most $i - 1$. The vertex $\langle q', l' \rangle$ cannot be infinite, since it is reachable from $\langle q, l \rangle$, and hence $\text{rank}(q', l') = i$.

- In the case when $i$ is odd, the vertex $\langle q, l \rangle$ is $F$-free in $G_i$ by the definition of rank. If $\langle q', l' \rangle \notin G_i$ then, by Lemma 4.4, $\text{rank}(q', l')$ is at most $i - 1$. If $\langle q', l' \rangle \in G_i$ then $\langle q', l' \rangle$ must also be $F$-free in $G_i$ and it follows that $\text{rank}(q', l') = i$. 

\[ \square \]
Lemma 4.6 In every infinite path in $G_r$ there exists a vertex $(q, l)$ with an odd rank such that all vertices $(q', l')$ in the path that are reachable from $(q, l)$ have rank$(q', l') = $ rank$(q, l)$.

Proof: In an infinite path in $G_r$ there is a vertex $(q, l)$ from which all vertices $(q', l')$ in the path are reachable. From Lemma 4.5 we have that all vertices $(q', l')$ have rank$(q', l') \leq$ rank$(q, l)$. Since we have a finite upper bound, the rank can only decrease finitely many times and there must exist a vertex $(q, l)$ such that rank$(q', l') = $ rank$(q, l)$.

Assume (for contradiction) that $i =$ rank$(q, l)$ is even. From the definition we have that $(q, l)$ is finite in $G_r$. Hence, all states reachable from $(q, l)$ must also be finite. But this contradicts that there are an infinite number of vertices $(q', l')$, and thus $(q, l)$ must have odd rank.

We will now show a bound on the rank function. This will have consequences for the size of the automaton in the next section. The construction and the proofs of both the lemma and theorem is new for this thesis.

Theorem 4.4 [Gurumurthy et al., 2003] The maximum rank needed for an alternating co-Büchi automaton is $2(n - |F|)$.

Lemma 4.7 There is a family $A_1, A_2, \ldots$ of universal co-Büchi automata such that $A_n$ has $n + 1$ states, acceptance set of size 1 and rank $2n$.

Construction: The universal co-Büchi automaton $A_n = (\Sigma, Q, q_0, \delta, F)$ from the previous lemma is constructed as follows:

$$\Sigma = \{a_0, a_1, \ldots, a_n\},$$

$$Q = \{q_0, q_1, \ldots, q_n\},$$

$q_0$ is the initial state,

$\delta$ is defined as follows:

$$\delta(q_0, a_i) = q_i \text{ } \forall i \in \{0, 1, \ldots, n\},$$

$$\delta(q_i, a_0) = q_0 \land q_{i+1} \text{ } \forall i \in \{1, 2, \ldots, n - 1\},$$

$$\delta(q_i, a_i) = q_i \text{ } \forall i \in \{1, 2, \ldots, n - 1\},$$

$$\delta(q_i, a_j) = q_n \text{ } \forall i \in \{1, 2, \ldots, n - 1\} \forall j \neq i \in \{1, 2, \ldots, n\},$$

$$\delta(q_n, a_i) = q_n \text{ } \forall i \in \{0, 1, \ldots, n\},$$

$$F = \{q\}$$

The universal co-Büchi automaton $A_3$ is illustrated in Figure 4.3 and the DAGs constructed for $A_3$ in Figure 4.4.

Proof: Let $G_r$ be a DAG that corresponds to an accepting computation tree of $A_n$ and construct a sequence of DAGs starting with $G_0 = G_r$. There are no finite vertices in $G_0$ so $G_1 = G_0$. In $G_1$ the vertices $(q_n, l)$ cannot reach any $F$-vertices, so $G_2 \setminus \{q_n, l\}$. There are no finite vertices in $G_2$, so $G_3 = G_2$. The vertices $(q_{n-1}, l)$ are $F$-free in $G_3$, so $G_4 = G_3 \setminus \{q_{n-1}\}$. Continuing this
Iteration, we end up in $G_{2n-1}$, where the vertices $\langle q_t, l \rangle$ are $F$-free. Thus $G_{2n} = G_{2n-1} \setminus \{ \langle q_t, l \rangle \}$, and in $G_{2n}$ the remaining vertices $\langle q_b, l \rangle$ are finite (since $q_b \in F$) and get rank $2n$.

Proof: (of Theorem 4.4) Corollary 4.1 says that $G_{2(n-|F|)+1}$ is empty and by the definition of rank (Definition 4.7), no vertex in the run tree of an alternating co-Büchi automaton gets a higher rank than $2(n - |F|)$. By Lemma 4.7 there exists such automata.

4.2.1 Minimal rank

The size of the weak alternating automaton $A'$ that we will construct in the next section depends on the highest rank in the original $n$-state automaton $A$. If $x$ is the highest rank of $A$, $A'$ will have exactly $n(x + 1)$ states.

We saw in Theorem 4.4 that there is an upper limit of the highest rank of $A$, namely $2(n - |F|)$. Sometimes it is possible to use lower value as the highest rank, for example the automaton in Example 4.2 can be correctly ranked with $2$ as highest rank. Using this ranking, $A'$ would have $9$ states instead of $15$. Finding the minimal rank needed is however a hard problem:

Theorem 4.5 [Kupferman and Vardi, 2000] Let $A$ be an alternating co-Büchi automaton. The problem of finding the minimal rank required for $A$ is PSPACE-complete.

The construction we present will use the maximal rank bound. This will produce unnecessary big automata, but the polynomial algorithms of Section 4.7 could be used to make them smaller.
Figure 4.4: The sequence of infinite DAGs from the automaton in Figure 4.3 (the first eight levels).

4.3 From co-Büchi to WAA

With the rank definition from the previous section, we can build the partitions needed for weakness (see Definition 2.16). We will give the start state the highest rank, and transitions are only allowed to states with equal or lower rank. All states with odd rank are accept states. Note that we also negate the acceptance condition, to get back to Büchi acceptance.

**Theorem 4.6** [Kupferman and Vardi, 2000] Given an alternating co-Büchi automaton $A$, there exists a weak alternating automaton $A'$ such that $\mathcal{L}(A) = \mathcal{L}(A')$ and the number of states in $A'$ is quadratic to that of $A$.

In this section, we will use $[k]$ to denote the set of natural numbers $\{0, 1, \ldots, k\}$. The odd members of $[k]$ are denoted $[k]^\text{odd}$.

**Definition 4.8** The release function

$$release : B^+(Q) \times [2(n - |F|)] \rightarrow B^+(Q')$$

for an alternating automaton $A = (\Sigma, Q, q_0, \delta, F)$ with size $n$ is defined as follows: Let $\theta \in B^+(Q)$ and $i \in [2(n - |F|)]$:

$$release(\theta, i) = \theta(q \in \theta : q/ \bigcup_{r \leq i} \langle q, i' \rangle)$$
4.3. FROM CO-BÜCHI TO WAA

Construction: Assume \( A = (\Sigma, Q, \delta, F) \) and \( n = |Q| \). Then there is a weak alternating automaton \( A' = (\Sigma, Q', \delta', F') \) accepting the same language defined as follows:

\[
\begin{align*}
Q' &= Q \times [2(n - |F|)] \\
Q'_0 &= \langle q_0, 2(n - |F|) \rangle \\
F' &= \{ \langle q, i \rangle : q \in Q \text{ and } i \text{ is odd} \}
\]

\( \delta' : Q' \times \Sigma \rightarrow 2^Q \) is defined (with \( \langle q, i \rangle \in Q' \) and \( a \in \Sigma \)) as follows:

\[
\delta'((q, i), a) = \begin{cases} 
\text{release}(\delta(q, a), i) & \text{if } q \notin F \text{ or } i \text{ is even} \\
\text{false} & \text{if } q \in F \text{ and } i \text{ is odd}
\end{cases}
\]

That is, for every atom \( q \) in the formula \( \theta \), replace it by a disjunction of the atom combined with every rank lower than or equal to \( i \). An example:

\[
\text{release}((q_1 \land q_2), 2) = ((q_1, 2) \lor (q_1, 1) \lor (q_1, 0)) \land ((q_2, 2) \lor (q_2, 1) \lor (q_2, 0))
\]

To verify the construction we first show that \( A' \) is weak, then that it is sound and finally that it is complete.

Proposition 4.1 The alternating automaton \( A' = (\Sigma, Q', \delta', F') \) from the construction is weak.

Proof: For each rank \( i \in [2(n - |F|)] \), construct a set \( Q_i = Q \times \{i\} \). Define the relation \( \leq \) so that \( Q_i \leq Q_{i'} \) iff \( i \leq i' \). If we consider a state \( \langle q, i \rangle \in Q \) and some letter \( a \in \Sigma \) we see by the definition of \( \delta' \) that \( \delta'(\langle q, i \rangle, a) \) is in a set \( Q_x \) such that \( Q_{i'} \leq Q_i \). We also see that a set \( Q_i \) is in \( F' \) if \( i \) is odd.

\[ \square \]

Proposition 4.2 Let \( A' = (\Sigma, Q', \delta', F') \) be the weak alternating automaton corresponding to the alternating co-Büchi automaton \( A = (\Sigma, Q, \delta, q_0, F) \). If \( A' \) accepts an input \( \alpha \) then \( A \) also accepts \( \alpha \).

Proof: Let \( \sigma_\alpha \) be an accepting branch in \( T(A', \alpha) \) on input \( \alpha \). Since the \( \delta' \) function just marks a state with a rank, \( \sigma_\alpha \) is also a branch in \( T(A, \alpha) \). The acceptance tell us that some state \( q' \in F' \) is visited an infinite number of times by every run of \( \sigma_\alpha \). By the definition of \( \delta' \), we see that a state \( \langle q, i \rangle \) with \( q \in F \) and \( i \text{ odd} \) can never be part of an accepting run. Hence, \( q' \) must belong to the set \( \{ \langle q, i \rangle : q \in Q \setminus F, i \text{ is odd} \} \). This meets the acceptance condition of the alternating co-Büchi \( A \), and hence it accepts \( \sigma_\alpha \).

\[ \square \]

Proposition 4.3 Let \( A' = (\Sigma, Q', \delta', F') \) be the weak alternating automaton corresponding to the alternating co-Büchi automaton \( A = (\Sigma, Q, \delta, q_0, F) \). If \( A \) accepts the input \( \alpha \) then \( A' \) also accepts \( \alpha \).

\[ \square \]
Proof: Let $\sigma_{\alpha}$ be an accepting branch in $T(A, \alpha)$ on input $\alpha$. The root in $T(A', \alpha)$ is labeled $\langle q_0, 2(n - |F|) \rangle$. If $a_0$ is the first letter of $\alpha$ and $q_1 \in \delta(q_0, a_0)$ then by definition of $\delta'$ the next level in the run is labeled $\delta'(\langle q_0, 2(n - |F|) \rangle) = \langle q_1, i \rangle$ and $i \leq 2(n - |F|)$. So $\sigma_{\alpha}$ is a branch in $T(A', \alpha)$. Since $\sigma_{\alpha}$ is an accepting branch, every $q \in F$ in a run $\sigma_{\alpha}$ is visited finitely many times. So all states $\langle q, i \rangle$ in $Q'$ with $q \in F$ and $i$ odd are visited only a finite number of times. Since the run is infinite and accepting in $A$, there must be states $\langle q, i \rangle$ with $q \notin F$ visited an infinite number of times. By lemma 4.6, $i$ must be odd in these states. Thus, $F'$ is visited an infinite number of times and $A'$ accepts $\alpha$.

Now we can prove Theorem 4.6.

Proof: By Proposition 4.1 we know that $A'$ is a weak alternating automaton, Proposition 4.2 shows soundness and Proposition 4.3 says it is complete. If $A$ has $n$ states the definition of $Q'$ gives that $A'$ has $n(2(n - |F|) + 1)$ states, which is $O(n^2)$.

4.4 From WAA back to Büchi

In this section we describe a construction to translate a weak alternating automaton into a nondeterministic Büchi automaton.

Theorem 4.7 [Miyano and Hayashi, 1984] Let $A$ be an alternating automaton. There is a nondeterministic Büchi automaton $B$ with $L(B) = L(A)$, and the size (in states) of $B$ is exponential to that of $A$.

Miyano & Hayashi [Miyano and Hayashi, 1984] where the first to present this theorem, later Kupferman & Vardi [Kupferman and Vardi, 2000] managed to simplify it. We will present the later version with our own proofs.

Construction: Given an alternating automaton $A = \langle \Sigma, Q, q_0, \delta, F \rangle$, we construct a Büchi automaton $B = \langle \Sigma, Q', q_0', \delta', F' \rangle$:

$$Q' \subseteq \{ \langle S, O \rangle \mid S \subseteq 2^Q, O \subseteq S \}$$

$$q'_0 = \langle \{ q_0 \}, \emptyset \rangle$$

$$\delta' : Q' \times \Sigma \rightarrow 2^{Q'}$$

$$F' = \{ \langle S, \emptyset \rangle \mid S \subseteq 2^Q \}$$

The $\delta'$ function is constructed as follows. A transition $\delta'(\langle S, O \rangle, a)$ (with $a \in \Sigma$ and $\langle S, O \rangle \in Q'$) is:
If \( O = \emptyset \):
\[
\delta'(\langle S, O \rangle, a) = \{ \langle S', S' \setminus F \rangle \mid S' = \bigwedge_{q \in S} \delta(q, a) \}
\]
If \( O \neq \emptyset \):
\[
\delta'(\langle S, O \rangle, a) = \{ \langle S', O' \setminus F \rangle \mid S' = \bigwedge_{q \in S} \delta(q, a) \}
\]
\[
O' = \bigwedge_{q \in O} \delta(q, a) \}
\]

A state in the nondeterministic Büchi automaton is a pair \( \langle S, O \rangle \), where \( S \) and \( O \) are sets representing conjunctions of states from the original automaton. \( S \) is the set of states the original automaton can currently be in, somewhat similar to the subset construction. The set \( O \) is used as a memory of states that are not accepting. In the start state, \( O \) is empty. After the first transition, it is “loaded” with all non-accepting states from the \( S \) set. During the following transitions, these states will be updated to reflect where the original automaton could have been. If a state turns out to reach an accept state it is removed from the \( O \) set. Eventually this will lead to the \( O \) set being empty, meaning that all possible runs up to this state pass an accepting state. In the constructed Büchi automaton, the states with \( O = \emptyset \) form the accept set.

If a run is not accepting, then this run will stay forever in the memory and we will never reach the empty set.

The memory construction

Why is the construction with a memory set necessary? Consider a branch that has two runs. The branch should be accepting if both runs accept. Assume now that the two runs by the automaton definition take turns visiting their accept states, such that they never are in an accept state at the same time. Clearly both runs contain an infinite number of accept states and are accepting. If we did not use a memory, we would have to compare the runs level by level and never realize this.

This construction is sound and complete:

**Proposition 4.4 (Soundness)** Let \( B = \langle \Sigma, Q', q'_0, \delta', F' \rangle \) be the Büchi automaton corresponding to the alternating automaton \( A = \langle \Sigma, Q, q_0, \delta, F \rangle \). If \( B \) accepts an input \( \alpha \in \Sigma^\omega \) then \( A \) accepts \( \alpha \).

**Proof:** If \( B \) is accepting then there is a unique accepting run \( \sigma_\alpha \) on the input \( \alpha \), describing a unique branch in \( T(A, \alpha) \). Since \( \sigma_\alpha \) is an accepting run in \( B \) we know that \( \inf(\sigma_\alpha) \cap F' \neq \emptyset \). Thus there is some state \( q' \in F' \) that is visited infinitely often, and the memory of \( B \) is empty infinitely many times. Because of this, every run in branch \( \sigma_\alpha \) must be accepting; a rejecting branch would make the empty memory unreachable. This shows that \( A \) will be accepting.

\[ \]

**Proposition 4.5 (Completeness)** Let \( B = \langle \Sigma, Q', q'_0, \delta', F' \rangle \) be the Büchi automaton corresponding to the alternating automaton \( A = \langle \Sigma, Q, q_0, \delta, F \rangle \). If \( A \) accepts an input \( \alpha \in \Sigma^\omega \) then \( B \) accepts \( \alpha \).
**Proof:** If $A$ is accepting then there is a unique accepting branch $\sigma_\alpha$ in $T(A, \alpha)$ on the input $\alpha$. The branch $\sigma_\alpha$ corresponds to a unique run $\sigma_\alpha$ in $B$. With $\sigma_\alpha$ being accepting, so is every run on $\sigma_\alpha$. Therefore every run will visit an accepting state infinitely often and the memory will be empty an infinite number of times, and $B$ is accepting.

\[\square\]

Now we are ready to prove Theorem 4.7:

**Proof:** By Proposition 4.4 we see that the construction is sound and by Proposition 4.5 that it is also complete. Hence if $|Q| = n$ then $|Q'| = 2^{2(n-|F|)}$.

\[\square\]

### 4.5 The composed algorithm

The constructions from Theorem 4.2, Theorem 4.6 and Theorem 4.7 can be composed into one single construction, thus eliminating the need for an intermediate representation of an alternating Büchi automaton.

**Construction:** Given an Büchi automaton $B = (\Sigma, Q, \delta, F)$ with $n$ states, construct a nondeterministic Büchi automaton $B' = (\Sigma, Q', q'_0, \delta', F')$ such that:

$$
Q' \subseteq \{ \langle S, O \rangle \mid S \in 2^{Q \times [2(n-|F|)]}, O \subseteq S \} \\
q'_0 = \{ \{ \langle q_0, 2(n - |F|) \rangle \}, \emptyset \} \\
F' = \{ \langle S, \emptyset \rangle \mid S \in 2^{Q \times [2(n-|F|)]} \}
$$

In the $\delta$ function, a transition from a state $\langle S, O \rangle \in Q'$ on the letter $a$ is constructed as follows:

If $O \neq \emptyset$ then:

$$
\delta'((S, O), a) = \left\{ \langle S', O' \rangle \mid \{ Q \times [2(n-|F|)]^{\text{odd}} \} \mid \\
\begin{array}{ll}
\text{release}(\delta(q, a), i) & \text{if } q \in F \text{ or } i \text{ is even} \\
\text{false} & \text{if } q \in F \text{ and } i \text{ is odd}
\end{array}
\right\}
$$

$$
S' = \bigwedge_{(q, i) \in S} \left\{ \begin{array}{ll}
\text{release}(\delta(q, a), i) & \text{if } q \in F \text{ or } i \text{ is even} \\
\text{false} & \text{if } q \in F \text{ and } i \text{ is odd}
\end{array} \right\}
$$

If $O = \emptyset$ then:

$$
\delta'((S, O), a) = \left\{ \langle S', S' \rangle \mid \{ Q \times [2(n-|F|)]^{\text{odd}} \} \mid \\
\begin{array}{ll}
\text{release}(\delta(q, a), i) & \text{if } q \in F \text{ or } i \text{ is even} \\
\text{false} & \text{if } q \in F \text{ and } i \text{ is odd}
\end{array}
\right\}
$$

$$
S' = \bigwedge_{(q, i) \in S} \left\{ \begin{array}{ll}
\text{release}(\delta(q, a), i) & \text{if } q \in F \text{ or } i \text{ is even} \\
\text{false} & \text{if } q \in F \text{ and } i \text{ is odd}
\end{array} \right\}
$$
4.5. THE COMPOSED ALGORITHM

To make this a bit more readable, Kupferman & Vardi introduced the terms possible, covers and consistent [Kupferman and Vardi, 2000] for describing the state sets.

**Definition 4.9** A set \( P \in 2^Q \times 2^{|F|} \) is **possible** iff there are no pairs \( (q, i) \in P \) such that \( q \in F \) and \( i \) is odd.

This definition expresses exactly the same as the false cases of \( S' \) and \( O' \) in the definition of \( \delta \). Hence, we can rewrite the definition to (the case for \( O' \) is similar):

\[
S' = \bigwedge_{(q, i) \in S} \text{release}(\delta(q, a), i) \text{ and } S' \text{ is possible}.
\]

**Definition 4.10** For a pair of sets \( P_1, P_2 \in 2^Q \times 2^{|F|} \) and some letter \( a \), \( P_2 \) **covers** \( (P_1, a) \) iff there for every pair \( (q, i) \in P_1 \) and state \( q' \in \delta(q, a) \) exists an \( i' \leq i \) such that \( (q', i') \) is in \( P_2 \).

In the definition of \( \delta \), this expresses that transitions only can go to equal or lower rank. By definition, this is what the release function will produce. Hence the \( S' \) definition can once again be rewritten:

\( S' \) covers \( S \) and is possible.

**Definition 4.11** A set \( P \subseteq Q \times 2^{|F|} \) is **consistent** iff for every pair \( (q_1, i_1), (q_2, i_2) \in P, q_1 = q_2 \) implies \( i_1 = i_2 \).

We will use the concept of consistent sets to reduce the number of states in the resulting automaton. In its present form, the construction has a blow-up of \( 2^{O(n^2)} \) in the worst case; Theorem 4.2 (where we construct the dual of the original Büchi) involves no blow-up. When we transform it into a weak alternating automaton (Theorem 4.6) we get a quadratic blow-up, ending up with \( O(n^2) \) states. The transformation back to Büchi (Theorem 4.7) is exponential, thus giving us \( 2^{O(n^3)} \) states.

By only considering consistent states, we will reduce this to \( 2^{O(n \log n)} \) states and still find the sought complement:

**Proposition 4.6** [Kupferman and Vardi, 2000] Let \( B = (\Sigma, Q', q_0', \delta', F') \) be the Büchi automaton constructed in Theorem 4.7 from the weak alternating automaton \( A = (\Sigma, Q, q_0, \delta, F) \). If we restrict \( Q' \) to only contain states \( (S, O) \) where \( S \) is a consistent subset of \( Q \times 2^{|F')} \), \( B \) will still accept the complement language of \( A \).

**Proof:** Let \( \sigma_\alpha \) be an accepting run of the alternating co-Büchi automaton \( A \) from Theorem 4.6 on the input \( \alpha \). By that theorem, there is a weak automaton \( A' \) that has an accepting run on the same input. The run of \( A \) corresponds to a DAG \( G_\tau \), and since \( A' \) is weak it will follow the ranking of \( G_\tau \). Every vertex in \( G_\tau \) has one and only one rank associated with it. By Theorem 4.3 there are at most \( n \) vertices on a level of \( G_\tau \) with each state \( q_i \) occurring at most once, and hence each level must be consistent. Because the automaton only is at one level at a time, the run will also be consistent. Therefore, the states visited by \( \sigma_\alpha \) in \( G_\tau \) can form a consistent set.
CHAPTER 4. COMPLEMENTATION

The construction in Theorem 4.7 for transforming $A'$ into a Büchi automaton
$B$ is a subset construction. If all states visited in $A'$ are consistent, then so are
the states visited in $B$.
Thus, we only need to consider states of consistent sets.

Notice that since $O \subseteq S$ by definition, $O$ will be a consistent set too.

**Proposition 4.7** [Kupferman and Vardi, 2000] There are $2^{O(n \log n)}$ consistent
subsets of $Q \times [2(n - |F|)]$.

**Proof:** The consistent set $S$ is a set of pairs $(q, i)$ where every $q$ is assigned
exactly one $i$. First the projection $\pi_Q : S \rightarrow Q$ takes out the states $q$ from the
set, then we assign an $i$ to every $q$ by the function $f : Q \rightarrow [2(n - |F|)]$. There
are $2^n$ such projections and the number of assignments are
$n^{2(n - |F|)} = 2^{O(n \log n)}$. Hence there are $2^{O(n \log n)}$ consistent sets.

We now give the construction defined by the terms possible, consistent and covers:

**Construction:** Given a Büchi automaton $B = (\Sigma, Q, q_0, \delta, F)$, construct a
non-deterministic Büchi automaton $B' = (\Sigma, Q', q'_0, \delta', F')$ such that:

\[
Q' \subseteq \{ (S, O) \mid S \in 2^Q \times [2(n - |F|)], O \subseteq S \text{ and } S \text{ is possible and consistent.} \}
\]
\[
q'_0 = \{ (\{q_0, 2(n - |F|)\}, \emptyset) \}
\]
\[
F' = \{ (S, \emptyset) \mid S \in 2^Q \times [2(n - |F|)] \text{ and } S \text{ is possible and consistent} \}
\]

In the $\delta'$ function, a transition from a state $(S, O) \in Q'$ on the letter $a$ is
constructed as follows:

\[
O \neq \emptyset: \quad \delta'( (S, O), a) = \{ (S', O') \mid \{ Q \times [2(n - |F|)]^{odd} \} \mid S' \text{ covers } (S, a),
\]
\[
\text{and } S' \text{ is possible and consistent} \}
\]
\[
O = \emptyset: \quad \delta'( (S, O), a) = \{ (S', S') \mid \{ Q \times [2(n - |F|)]^{odd} \} \mid S' \text{ covers } (S, a),
\]
\[
\text{and } S' \text{ is possible and consistent} \}
\]

4.6 Lower bound of complementation

The complement construction that only considered consistent states had a worst
case behavior of $2^{O(n \log n)}$ states. Michel's bound in the next theorem shows
that the best worst case behavior a construction can have is $2^{O(n \log n)}$ states.
Thus the complement construction is optimal in the sense that one cannot make
a better guarantee on the blow-up in states for any automaton.

Note that this is the complexity of the worst case. In the next section, we
will show classes of automata where we can use more efficient algorithms that
perform better than $2^{O(n \log n)}$. 

4.6. LOWER BOUND OF COMPLEMENTATION

\[ q_0 \]
\[ q_1 \]
\[ q_2 \]
\[ q_{n+1} \]
\[ q_n \]

Figure 4.5: One nondeterministic Büchi automaton that satisfy Lemma 4.8 and Lemma 4.9.

**Theorem 4.8** For each \( n \geq 0 \), there exists a set \( \mathcal{L}_n \) of languages of infinite sequences recognized by a Büchi automaton with \( n + 2 \) states, such that any Büchi automaton recognizing \( \mathcal{L}_n \) has at least \( n! \) states, i.e. \( 2^{\Theta(n \log n)} \) states.

There is a nice proof of this theorem by Perrin & Pin [Perrin and Pin, 2002]. To prove Michel's bound one must show that there exists a nondeterministic Büchi automaton that cannot be complemented without a blow-up of at least \( n! \) states, which is equivalent to \( 2^{\Theta(n \log n)} \) states. This can be shown by using Stirling's approximation (Section 2.3). There are several possible definition of such nondeterministic Büchi automata, one is shown in Figure 4.5.

In the proof, Perrin & Pin use the two following lemmas to define the languages \( \mathcal{L}_n \) that \( B \) must recognize. Then they show that any nondeterministic Büchi automaton that recognizes the complement language \( \overline{\mathcal{L}_n} \) must have at least \( n! \) blow-up in states.

**Lemma 4.8** Let \( \{i_1, i_2, \ldots, i_k\} \) be a subset of \( \{i_1, i_2, \ldots, i_n\} \). If an \( \omega \)-sequence \( \alpha \) contains infinitely many occurrences of each of the factors \( (i_1i_2, i_2i_3, \ldots, i_{k-1}i_k) \) and there is a finite path from \( q_i \) to \( i_1 \) in a prefix of \( \alpha \) in \( B \) then \( \alpha \in \mathcal{L}_n \).

**Lemma 4.9** Let \( P \) be a permutation on \( \{1, 2, \ldots, n\} \), and let the infinite sequence \( \alpha_P = (P(1)P(2) \cdots P(n))^{\omega} \). For any permutation \( P \) of \( \{1, 2, \ldots, n\} \), the infinite sequence \( \alpha_P \) is not in \( \mathcal{L}_n \).

Let \( \overline{B} \) be a nondeterministic Büchi automaton that recognize the complement of \( \mathcal{L}_n \). By Lemma 4.9, each infinite sequence \( \alpha_P \) is accepted by \( \overline{B} \). Each \( \alpha_P \) corresponds to a unique accepting run \( \sigma_{\alpha_P} \). There are \( n! \) permutations of \( \{1, 2, \ldots, n\} \) and therefore \( n! \) different \( \alpha_P \). Hence there are \( n! \) different \( \sigma_{\alpha_P} \).
since they are unique and to prove the theorem it will be sufficient to show that each run visits states that are disjoint to the states that another run will visit. Or more formally, if \( P \neq P' \) then \( \text{inf}(\sigma_{\alpha P}) \cap \text{inf}(\sigma_{\alpha P'}) = \emptyset \).

The fact that the two different run gives two disjoint sets is shown by contradiction. Assume that the state \( q \) belong to both \( \text{inf}(\sigma_{\alpha P}) \) and \( \text{inf}(\sigma_{\alpha P'}) \). Then we can construct a run \( \sigma \) of \( \overline{B} \) by visiting every state in \( \text{inf}(\sigma_{\alpha P}) \) once, move through \( q \) to \( \text{inf}(\sigma_{\alpha P'}) \), visit every state there and iterate back to \( \text{inf}(\sigma_{\alpha P}) \) through \( q \) again. Since \( q \) is a member of both sets, they must be reachable from each other. Thus, \( \text{inf}(\sigma) \) will be the union of \( \text{inf}(\sigma_{\alpha P}) \) and \( \text{inf}(\sigma_{\alpha P'}) \). This means that the acceptance conditions of both \( \overline{B} \) and \( B \) are satisfied (by Lemma 4.8 and 4.9), and this is a contradiction. Hence \( \text{inf}(\sigma_{\alpha P}) \cap \text{inf}(\sigma_{\alpha P'}) = \emptyset \), and there must be \( n! \) disjoint sets of \( \overline{B} \).

4.7 Minimization

We have shown that the complementation algorithm has an optimal complexity in the worst case. This does not mean that the nondeterministic automaton constructed is optimal; first, there are automata that are easier to complement than the worst case. Secondly, we only measure the complexity by the number of states. The transition function does also take up memory which we could save by identifying and removing unnecessary transitions.

From the theory of finite automata on finite words, we know that there is an optimal automata for each language. When we move to languages over infinite sequences, the existence of an optimal automaton is still an open question. We do not know if there in the general case is a minimal automata or what its structure might be.

There are however several techniques to reduce the size of an automaton over infinite sequences. We will describe a simple rank-reduction method and the simulation technique “fair simulation”, because they seem to improve the automaton the most.

4.7.1 Choosing the right algorithm

By recognizing properties of the automaton we are complementing, we can choose the algorithm with the lowest blow-up in states.

Deterministic

As we saw in Section 3.4, complementation of a deterministic Büchi automaton involves only a linear blow-up. Checking if an automaton is deterministic (by Definition 2.8) is also easy. Hence, it makes sense to test this to avoid using a unnecessary complex algorithm.

Testing weakness

If the original Büchi automaton is weak, then the alternating automaton from the Theorem 4.2 will also be weak; The dualization will not introduce any new transitions that could break the weakness.
4.7. MINIMIZATION

We have constructed the algorithm \texttt{WeakTest} for checking if a nondeterministic Büchi automaton is weak. The idea is to try to form the partitions needed for weakness and resolve conflicts when they occur.

\begin{algorithm}
\textbf{WeakTest} \(B\)
\begin{enumerate}
\item \( G \leftarrow \text{MakeGraph}(B) \)
\item \( n \leftarrow |Q| \)
\item \( q \leftarrow \emptyset \)
\item \( \text{Expanded} \leftarrow \emptyset \)
\item \textbf{repeat}
\item \( \text{Expanded} \leftarrow \text{Expanded} \cup q \)
\item \( \text{NumberState}(G, F, \langle q, n \rangle) \)
\item \( \langle q, n \rangle \leftarrow \text{FindFault}(G, \text{Expanded}) \)
\item \textbf{until} \( \langle q, n \rangle = \emptyset \)
\item \textbf{return} \( \text{TestEdges}(G) \)
\end{enumerate}
\end{algorithm}

\begin{algorithm}
\textbf{NumberState} \(G, F, \langle q, n \rangle\)
\begin{enumerate}
\item \textbf{if} \( q \in F \)
\item \textbf{then} for all \( \langle q', n' \rangle \) where \( q' \) reachable from \( q \)
\item \textbf{do} if \( q' \in F \)
\item \textbf{then} \( n' \leftarrow n \)
\item \textbf{else} \( n' \leftarrow n - 1 \)
\item \textbf{else} for all \( \langle q', n' \rangle \) where \( q' \) reachable from \( q \)
\item \textbf{do} if \( q' \in F \)
\item \textbf{then} \( n' \leftarrow n - 1 \)
\item \textbf{else} \( n' \leftarrow n \)
\end{enumerate}
\end{algorithm}

\texttt{MakeGraph} in Line 1 will transform a nondeterministic Büchi automaton \( B \) to a graph \( G \). The vertices of \( G \) are the states in \( B \) and there is an edge between two vertices if there exists a transition between the corresponding states in \( B \). \texttt{Expanded} is the set of visited states.

The algorithm begins with the start state of the automaton, and labels the nodes with \texttt{NumberState}. Then it checks for conflicts (that is, an unexplored node labeled \( n \) that has a neighbour with a label \( > n \)). If one is found, the graph is relabeled according to this node and the iteration continues.

Since we relabel the graph at most once per node, the loop will terminate after at most \( n \) iterations. Then we run \texttt{TestEdges} to check if our labeling is free from conflicts. If it is, we have a partial order of the states. If it is not, we have an unresolvable conflict somewhere and hence a non-weak automaton.

\texttt{MakeGraph} can be done in \( O(|\delta|) \) time where \( |\delta| \) is the number of transitions in \( B \). \texttt{NumberState} runs in \( O(m) \) time in the worst case, where \( m \) is the number of edges in \( G \). \texttt{FindFault} runs in \( O(n) \) time where \( n \) is the number of nodes in \( G \) (which is the same as the number of states in \( B \)).
The loop of \textsc{WeakTest} runs at most \(n\) times. Inside the loop, \textsc{NumberState} requires \(O(m)\) time, and since \(m \leq n^2\) this corresponds to \(O(n^2)\) time. Hence, the overall complexity is \(O(n^3)\) in the worst case.

\textbf{Proposition 4.8} If Algorithm 4.7.1 accepts the nondeterministic Büchi automaton \(B = (\Sigma, Q, q_0, \delta, F)\) then the corresponding graph \(G\) must have a partial order and all states in a partition must either belong to \(F\) or \(Q \setminus F\).

\textbf{Proof:} If the algorithm accepts, this means that the test at line 10 found that all nodes are legally labeled. Hence, the labels describe a partial order on \(G\). The procedure \textsc{NumberState} ensures that the label will be different between states in \(F\) and in \(Q \setminus F\).

\[\square\]

\textbf{Proposition 4.9} If the graph \(G\) corresponding to the nondeterministic Büchi automaton \(B = (\Sigma, Q, q_0, \delta, F)\) has a partial order such that all states in a partition either belong to \(F\) or \(Q \setminus F\), then Algorithm 4.7.1 will accept \(B\).

\textbf{Proof:} The algorithm gives all reachable states labels, the start state \(q_0\) get labeled \(n\). Assume that \(q_0 \in F\). The states \(q\) that are reachable from the start state get the label \(n\) iff \(q \in F\), and label \(n - 1\) otherwise. If a state \(q'\) is illegally labeled (if there is a transition from \(q'\) to a state with a higher label) then all states reachable from it are relabeled. Since \(G\) has a partial order, the relabeling will only affect the same and lower partitions. The relabeling will also introduce a new partition. The labeling of this partition will not introduce any errors into earlier partitions. Hence, the algorithm will eventually find a legal labeling of \(G\).

\[\square\]

To complement a weak nondeterministic Büchi automaton we dualize \(\delta\) by Theorem 4.2 and get an alternating co-Büchi automaton. To use the translation from alternating Büchi automaton to a nondeterministic Büchi automaton in Section 4.4 we need to transform the alternating co-Büchi to an alternating Büchi automaton. For this, we use the following construction:

\textbf{Construction:} Let \(A = (\Sigma, Q, \phi, \delta, F)\) be a weak alternating co-Büchi automaton. We construct an alternating Büchi automaton \(A' = (\Sigma, Q, q_0, \delta, F')\) where

\[F' = \{q_i \mid q_i \notin F\}.\]

Since the change is only in the acceptance set, this involves no blow-up in the size of the automaton. The transformation back to a nondeterministic is done with \(2^{O(n)}\) states, hence a weak automaton can be complemented with a blow-up of “only” \(2^{O(n)}\) states.

\textbf{Weak and deterministic}

A deterministic automaton will suffer no blow-up when translated back to a nondeterministic Büchi automaton. We saw above that a weak automaton can be translated to an weak alternating automaton with nothing but the acceptance set \(F\) changed. Hence, a weak deterministic Büchi automaton can be
complemented with no blow-up by the weak construction above. Its size will be smaller than the deterministic complement in Section 3.4; The deterministic complement produces an automaton with size \(2n - |F|\), the weak construction has only \(n\) states.

4.7.2 Reducing the size of \(\delta\)

The idea in the definition of release (in Definition 4.8) is that, from a state \(q\) the automaton can go to a state with the same or any lower rank. This is done by making transitions to every possible state with a lower rank. The number of transitions is then \(|\delta|O(k^2)\) where \(k\) is the maximal rank of the automaton.

Instead of taking a big step down to a lower state, we can take several small steps. From this idea we state a new definition of release which have \(3|\delta|O(k)\) transitions [Kupferman and Vardi, 2000].

**Definition 4.12** The release function

\[
release : \mathcal{B}^+(Q) \times [2(n - |F|)] \rightarrow \mathcal{B}^+(Q')
\]

is defined as follows: Let \(\theta \in \mathcal{B}^+(Q)\) and \(i \in [2(n - |F|)]\):

\[
release(\theta, i) = \theta \{ \forall q \in \theta : q / \langle q, i \rangle \lor \langle q, i - 1 \rangle \lor \langle q, i - 2 \rangle \}
\]

An example:

\[
release(q_1 \land q_2, 4) = (\langle q_1, 4 \rangle \lor \langle q_1, 3 \rangle \lor \langle q_1, 2 \rangle) \land (\langle q_2, 4 \rangle \lor \langle q_2, 3 \rangle \lor \langle q_2, 2 \rangle)
\]

With this new definition of release, Proposition 4.1, 4.2 and 4.3 still holds.

4.7.3 Simulations of nondeterministic Büchi automata

Minimizing a Büchi automaton can be done by merging two equivalent states into one state or removing redundant edges. This can only be done if the modified automaton accepts the same language as the original automaton. Since the problem to check language containment is EXPSPACE-complete we rather simulate the original automaton, since this can be done in polynomial time.

Simulations are like limited inclusion tests. If a state \(q'\) simulate the state \(q\) then \(q'\) can follow \(q's\) transitions without knowing the input beforehand. An automaton \(A\) is simulated by an automaton \(A'\) if the start state \(q_0\) of \(A\) is simulated by the start state \(q'_0\) of \(A'\). Hence, if the new automaton simulate the original automaton then the language of original automaton is a subset of the language of the new automaton. In this section we show a way to reduce the size of Büchi automata with simulation, a method by Gurumurthy [Gurumurthy et al., 2002] and Etessami [Etessami et al., 2001]. Their work is based on an algorithm by Jurdziński [Jurdziński, 2000]. In this thesis we follow mainly the work of Gurumurthy, with minor changes to match the definition of Büchi automata in the rest of this thesis.

We see the simulation as an infinite game between two players, an antagonist and a protagonist. The protagonist makes moves from an antagonist state to an antagonist state trying to prove simulation, while the antagonist makes moves from an antagonist state to a protagonist state and tries to find moves that can not be simulated. The game is played on a graph that we call an infinite game.
A run of an infinite game is like a run of a Büchi automaton, the players take turns and make every second move. We will call this a play \( \sigma \). A plan for how to choose moves is called a strategy.

**Definition 4.13** An infinite game \( G \) is a tuple \((Q_o, Q_p, \delta_G, F_G)\). \( Q_o \) and \( Q_p \) are disjoint sets of states, \( Q_o \) are the antagonist states and \( Q_p \) the protagonist states. \( \delta : Q_o \times Q_p \times \Sigma \rightarrow Q_o \times Q_p \) is the transition function and \( F \) is a condition that defines a subset of \((Q_o \cup Q_p)^\omega\) where \((Q_o \cup Q_p)^\omega\) is the set of all plays.

We use \([ \)\] to indicate an antagonist state and \( ( )\) to indicate a protagonist state. The infinite game is constructed from two Büchi automata as follows:

**Construction:** From two Büchi automata \( B = \langle \Sigma, Q, q_0, \delta, F \rangle \) and \( B' = \langle \Sigma, Q', q'_0, \delta', F' \rangle \) construct the infinite game \( G_{B, B'} = \langle Q_o, Q_p, \delta_G, F_G \rangle \) as follows:

\[
Q_o = \{ [q_1, q'_1] | q_1 \in Q, q'_1 \in Q' \text{ and } (q_l \in F \text{ or } q'_l \notin F') \},
\]

\[
Q_p = \{ (q_l, q'_l) | q_l \in Q \text{ and } q'_l \in Q' \}
\]

let \( a \in \Sigma \), \([q_1, q'_1] \in Q_o \) and \((q_2, q'_2) \in Q_p \) then

\[
\delta_G([q_1, q'_1], a) = (q_2, q'_2) \text{ if } \delta(q_l, a) = q_2,
\]

let \( a \in \Sigma \), \((q_1, q_2) \in Q_p \) and \([q_1, q'_1] \in Q_o \) then

\[
\delta_G((q_1, q'_1), a) = [q_1, q'_2] \text{ if } \delta'(q'_l, a) = q'_2,
\]

\[
F_G = \{ (G, R) \}\text{ where,}
\]

\[
G = \{ (q_1, q'_1) | q_1 \in F \text{ and } q'_1 \in Q' \},
\]

\[
R = \{ (q_l, q'_l) | q_l \in Q \text{ and } q'_l \in F' \}.
\]

If the game reaches a state with no successors and the last state is an antagonist state the protagonist wins. If a state with no successors is never reached the protagonist wins if \( \sigma \in F_G \). Otherwise the antagonist wins.

\( R \) is the set of winning states for the protagonist. \( G \setminus R \) is the set of winning states for the antagonist. The protagonist has a winning strategy if from every state it can reach some protagonist winning state.

We next define acceptance of an infinite game. Parity condition is defined in Gurumurthy [Gurumurthy et al., 2002]. The relationship between Streett acceptance condition and parity condition for parity games allows us to use Jurdiński’s algorithm.

**Definition 4.14** In an infinite game \( G_{B, B'} = \langle Q_o, Q_p, \delta_G, F_G \rangle \), \( F_G \) is a pair of sets \( F_G = \langle G, R \rangle \) and \( F_G \) is accepted iff \( \inf(\sigma) \cap G = \emptyset \) or \( \inf(\sigma) \cap R \neq \emptyset \). The acceptance condition \( F_G \) corresponds to the parity condition \( (R, G \setminus R, (Q_o \cup Q_p) \setminus (G \setminus R)) \).

There are different types of simulations. The three we will discuss in this thesis are **direct**, **fair** and **delayed**. Direct is the least powerful of these simulation but the easiest to compute. Fair is the most powerful, and will be used in this section. Delayed is somewhere between direct and fair in power, and as complex as fair to implement. We will use delayed and direct simulation for theoretical purposes in this section. In the next section, direct simulation will be used.
4.7. MINIMIZATION

Definition 4.15 Let $B = (\Sigma, Q, q_0, \delta, F)$ and $B' = (\Sigma, Q', q'_0, \delta', F')$ be two nondeterministic Büchi automata with the states $q_1, q_2$ and $q_3$ in $Q$ and the states $q_1', q_2'$ and $q_3'$ in $Q'$. We say $q_1$ is directly simulated by $q_1'$ if $q_1 \in F$ implies $q_1' \in F$. Moreover, $q_1$ is fairly simulated by $q_1'$ if the protagonist has a winning strategy from $[q_1, q_1'] \in Q_a$ in the game $G_{B,B'}$. We also say that $q_1$ is delayed simulated by $q_1'$ if for any play $\sigma$ starting in $[q_1, q_1'] \in Q_a$, if $|\sigma(i) = (q_2, q_2')$ where $q_2 \in F$ then there is a $j \geq i$ such that $\sigma(j) = (q_3, q_3')$ and $q_2' \in F'$.

To be able to make the fair simulations algorithm efficient we want to add and remove edges in the infinite game that correspond to adding or removing edges in the Büchi automaton. First a definition of how to add and remove edges in the infinite game.

Definition 4.16 Let $B = (\Sigma, Q, q_0, \delta, F)$ and $B' = (\Sigma, Q', q_0', \delta', F')$ be two nondeterministic Büchi automata, $\Delta \delta \subseteq \delta$ be a set of transitions and $G_{B,B} = (Q_a, Q_p, \delta_G, F_G)$ be an infinite game.

We define $\text{rem}(B, \Delta \delta) = (\Sigma, Q, q_0, \delta \Delta \delta, F)$, let $B' = \text{rem}(B, \Delta \delta)$ and let $\text{rem}(G_{B,B'}, \Delta \delta)$ be the game $(Q_a, Q_p, \delta_G, F_G)$, where

$$
\delta'_G = \delta_G \setminus \{(q_1, q_2), a\} = [q_1, q_2'] \quad | \quad (q_1, q_2) \in Q_p, [q_1, q_2] \in Q_a \text{ and } \delta(q_1, a) = q_2 \text{ for some } a \in \Sigma).
$$

We define $\text{add}(B, \Delta \delta) = (\Sigma, Q, q_0, \delta \cup \Delta \delta, F)$, let $B' = \text{add}(B, \Delta \delta)$ and let $\text{add}(G_{B,B'}, \Delta \delta)$ be the game $(Q_a, Q_p, \delta_G, F_G)$, where

$$
\delta'_G = \delta_G \cup \{((q_1, q_2), a) = [q_1, q_2'] \quad | \quad (q_1, q_2) \in Q_p, [q_1, q_2] \in Q_a \text{ and } \delta(q_1, a) = q_2 \text{ for some } a \in \Sigma).
$$

By this definition we can add or remove one edge or several edges of the automaton and derive the corresponding game. This means that we do not need to construct a new game after modifying the automaton, it is sufficient to just modify the game. This is formalized in the following theorem.

Theorem 4.9 Let $B = (\Sigma, Q, q_0, \delta, F)$ be a Büchi automaton and $G_{B,B}$ an infinite game. Let $\Delta \delta \subseteq \delta$ and $\Delta \delta' \subseteq \delta$ be two sets of transitions. Then $G_{B, \text{rem}(B, \Delta \delta)} = \text{rem}(G_{B,B}, \Delta \delta)$ and

$$
\text{rem}(\text{rem}(G_{B,B}, \Delta \delta), \Delta \delta') = \text{rem}(G_{B,B}, \Delta \delta \cup \Delta \delta').
$$

Similar holds for $\text{add}$, namely $G_{B, \text{add}(B, \Delta \delta)} = \text{add}(G_{B,B}, \Delta \delta)$ and

$$
\text{add}(\text{add}(G_{B,B}, \Delta \delta), \Delta \delta') = \text{add}(G_{B,B}, \Delta \delta \cup \Delta \delta').
$$

To calculate the fair simulation relation we use a function $r(p)$ which we call progress measure for a state $p$. The progress measure is a measure of cycles in the game, and $\infty$ represents that there is no cycle. The co-domain $\{0, 1, \ldots, n_1\} \cup \{\infty\}$ for $r(F)$ is an ordered set, with $n_1 + 1 = \infty$ and $\infty + 1 = \infty$.

The progress measure $r(p) < \infty$ if the protagonist has a winning strategy from $p$. We need the following definitions:
Algorithm 4.7.3 Calculate progress measure

\begin{algorithm}
\textbf{PROGRESSMEASURE}(G)
\begin{algorithmic}
\STATE for each \( p \in Q_G \)
\STATE do \( r(p) \leftarrow 0 \)
\STATE \textbf{UPDATEPROGRESSMEASURE}(G)
\STATE return \( G \)
\end{algorithmic}
\end{algorithm}

Algorithm 4.7.4 Update progress measure

\begin{algorithm}
\textbf{UPDATEPROGRESSMEASURE}(G)
\begin{algorithmic}
\STATE while exist a \( p \in Q_G \) such that \( \text{update}(r, p) \neq r(p) \)
\STATE do \( r \leftarrow \text{lift}(r, p) \)
\STATE return \( G \)
\end{algorithmic}
\end{algorithm}

Definition 4.17 Let \( G_{B, B'} = (Q_a, Q_p, \delta_G, F_G) \) be an infinite game and \( p, p' \in Q_G \) where \( Q_G = Q_a \cup Q_p \) and the Büchi automata \( B = (\Sigma, Q_a, q_0, \delta, F) \), \( B' = (\Sigma, Q'_a, q'_0, \delta', F') \). \( r : Q_G \to \{0, 1, \ldots, n_1\} \cup \{\infty\} \) is the progress measure where \( n_1 = |Q_p| \) is the number of protagonist states. We define (for \( a \in \Sigma \)):

\[
\text{update}(r, p') = \begin{cases} 
\max(\{r(p) \mid \delta(p', a) = p\}) + 1 & \text{if } p' = (q, q') \in Q_p \text{ where } q \in F \text{ and } q' \notin F' \\
\max(\{r(p) \mid \delta(p', a) = p\}) & \text{if } p' = (q, q') \in Q_p \text{ where } q \notin F \text{ or } q' \in F' \\
\min(\{r(p) \mid \delta(p', a) = p\}) & \text{if } p' \in Q_a
\end{cases}
\]

\[
\text{lift}(r, p') = \lambda p. \begin{cases} 
\text{update}(r, p') & \text{if } p = p' \\
r(p) & \text{otherwise}
\end{cases}
\]

Progress measure is computed by Algorithm 4.7.3 with the help of Algorithm 4.7.4.

The progress measure can be computed incrementally under some conditions, this will allow us to update the progress measure instead of computing it from start after every modification of the game. The conditions are that we only add transitions from an antagonist state to a protagonist state or remove transitions from a protagonist state to an antagonist state. This means that the game becomes harder and the progress measure will not decrease, so it is sufficient to update the progress measure and see if a state simulate some other state. This will be expressed by the following lemma.

Lemma 4.10 Let \( G_{B, B'} \) be an infinite game, and let \( \Delta \delta \subseteq \delta \) be a set of transition. Let the progress measure be \( r(p) \) of a state \( p \) in the game \( G_{B, B'} \). If \( r'(p) \) is the progress measure of \( \text{rem}(G_{B, B'}, \Delta \delta) \) then \( r(p) \leq r'(p) \). If \( r'(p) \) is the progress measure of \( \text{add}(G_{B, B'}, \Delta \delta) \) then \( r(p) \leq r'(p) \).

We try to merge two fair simulation equivalent states \( q \) and \( q' \) by adding transitions to the game graph. We construct \( \Delta \delta \) so that \( q \) has the same predecessors as \( q' \) and \( q' \) the same as \( q \). We also add transitions so that \( q \) and \( q' \) have the same successors. Then \( B' = \text{add}(B, \Delta \delta) \) and if \( B' \) simulates \( B \) then
Algorithm 4.7.5 Fair simulation

FAIRSIMULATION($B$)
1. Construct the infinite game $G$ by the construction of $G_{B,B}$.
2. PROGRESSMEASURE($G$)
3. for all states $q,q' \in Q$ such that $[q,q'] \in Q_a$:
   do Merge state $q$ and $q'$ by adding transitions.
5. UPDATEPROGRESSMEASURE($G$)
6. if $B$ fair simulate $B'$
   then accept merge
   if $q \in F$
      then remove $q$
   else remove $q'$
8. else undo merge.
9. for all equivalent edges $E$ from fair simulated states.
10. do Remove edge $e \in E$ from $B$.
11. UPDATEPROGRESSMEASURE($G$)
12. if $B$ fair simulate $B'$
13. then accept removal
14. else undo removal
16. return $B'$

the merge is accepted. If $q$ and $q'$ are delayed simulation equivalent then the merge is guaranteed to be accepted. Now how to construct $\Delta \delta$ for merging two states:

Construction: Let $B = \langle \Sigma, Q, q_0, \delta, F \rangle$ be a B"uchi automaton with $q_1, q_2$ and $q_3 \in Q$. To merge two fair simulation equivalent states $q_1$ and $q_2$ construct $\Delta \delta$ as follows:

$$\Delta \delta = \emptyset$$

for all $a \in \Sigma$

if $q_3 \in \delta(q_1, a)$ then $\Delta \delta \cup \delta(q_1, a) = q_3$

if $q_2 \in \delta(q_2, a)$ then $\Delta \delta \cup \delta(q_2, a) = q_3$

if $q_1 \in \delta(q_3, a)$ then $\Delta \delta \cup \delta(q_3, a) = q_2$

if $q_2 \in \delta(q_3, a)$ then $\Delta \delta \cup \delta(q_3, a) = q_1$

If a state $q$ fairly simulates $q'$ then they are fair simulation equivalent and express the same information. One of them should be removed, and this is done according to the following theorem. For two fair simulate equivalent states this is always possible.

Theorem 4.10. Let $B = \langle \Sigma, Q, q_0, \delta, F \rangle$ be a B"uchi automaton, $q_1, q_2, q_3 \in Q$, with the transitions $\delta(q_1, a) = \delta(q_2, a)$ for all $a \in \Sigma$, $q_1 \in \delta(q_3, a) \iff q_2 \in \delta(q_3, a)$ for all $a \in \Sigma$ and for all $q_3 \in Q$, and $q_1 \in F \Rightarrow q_2 \in F$. Then $L(A) = L(A')$ where $Q' = Q \setminus \{ q_1 \}$, if $q_1 = q_0$ then $q_0' = q_2$ else $q_0' = q_0$, $\delta'(q_1, a) = q_2$ if $\delta(q_1, a) = q_2$ and $q_1, q_2 \in Q'$, $F' = F \setminus \{ q_2 \}$.

Algorithm 4.7.5 is an algorithm for fair simulation of nondeterministic B"uchi automata. This is a two phase algorithm; in the first phase we try to merge
states, and in the second phase we try to remove unnecessary edges. The algorithm is greedy, it will do any modification that is allowed.

First we construct the game from the Büchi automaton and compute progress measure for the game. Then we try to merge a pair of mergeable fair simulation equivalent states, \( q \) and \( q' \) (all pairs that corresponds to an antagonist state are candidates). The set \( \Delta \delta \) in Line 4 is constructed as the construction above with \( q \) and \( q' \). This gives a new Büchi automaton \( B' = \text{add}(B, \Delta \delta) \) and also a new game \( G_{B, B'} \) where \( q \) and \( q' \) are merged. We update the progress measure for the game and test if \( B' \) fairly simulates \( B \), that is the start state of \( B' \) fairly simulates the start state of \( B \). If \( B \) is simulated, we accept the merge and one of the states can be removed by Theorem 4.10. If \( B' \) is not simulated we discard the merge.

We only need to try a pair once. If it is not accepted to merge the first time then it will not be accepted at a later time. If the automaton is not simulated by the new automaton then the progress measure is \( \infty \). Adding more edges to the game will not shrink the progress measure (by Lemma 4.10).

In the second phase we try to remove edges from the automaton. For each pair of simulating states, we try to remove equivalent edges one by one. For each edge, we update the progress measure and check if the new automaton fairly simulates the old one. If it does, the edge is removed.

When removing an edge in Line 13, let \( B' \) be the automaton with the removed edge defined as \( B' = \text{rem}(B, e) \) and let \( G_{B, B'} \) be the new game. We need only to check each edge once, this follows from Lemma 4.10 and a similar argument as above.

### 4.7.4 Removal of SCC

In the construction of the weak alternating automaton the states are divided into partitions. An accepting run will eventually enter an accepting partition and stay there, and this means the partition must contain an SCC. The size of the automaton could be reduced by removing SCCs that do not change the language. We will, as in the previous section, use simulation to verify this. This simulation technique is from Gurumurthy et al. [Gurumurthy et al., 2003].

We will use direct simulation since it is easier to compute. The infinite game will need a new definition:

**Construction:** From two Büchi automata \( B = (\Sigma, Q, q_0, \delta, F) \) and \( B' = (\Sigma, Q', q'_0, \delta', F') \) construct the infinite game \( G_{B, B'} = (Q_a, Q_p, \delta_G, F_G) \) as follows:

\[
Q_a = \{ [q_1, q'_1] \mid q_1 \in Q, q'_1 \in Q' \text{ and } (q_1 \in F \text{ or } q'_1 \notin F') \},
\]

\[
Q_p = \{ (q_1, q'_1) \mid q_1 \in Q \text{ and } q'_1 \in Q' \}
\]

let \( a \in \Sigma, [q_1, q'_1] \in Q_a \) and \( (q_2, q'_2) \in Q_p \):

\[
\delta_G([q_1, q'_1], a) = (q_2, q'_2) \text{ if } \delta(q_1, a) = q_2 \text{ and } (q_2 \notin F \text{ or } q'_2 \in F'),
\]

let \( a \in \Sigma, (q_1, q_2) \in Q_p \) and \( [q_1, q'_2] \in Q_a \):

\[
\delta_G((q_1, q'_2), a) = [q_1, q'_2] \text{ if } \delta'(q'_1, a) = q'_2 \text{ and } (q_1 \in F \text{ or } q'_2 \notin F'),
\]


Algorithm 4.7.6 Prune Nondeterministic Büchi automaton

PRUNE_B(NB(B))
1 Construct the game $G_B, B$ for direct simulation.
2 for all maximal SCCs $X$ in $B$ such that $X \cap F \neq \emptyset$ (from the start state)
3 do compute $B'$ by removing outgoing edges from $X$
4 if $B'$ direct simulate $B$
5 then minimize $B$ by removing $X$

$F_G = \{(G, R)\}$ where,
$G = \{(q_i, q'_i) \mid q_i \in F \text{ and } q'_i \in Q'\}$,
$R = \{(q_i, q'_i) \mid q_i \in Q \text{ and } q'_i \in F'\}$.

The correspondence between $\delta^{fair}_G$ (the $\delta_G$ function from fair simulation game) and the $\delta_G$ function from the direct simulation game is (with $a \in \Sigma$):

$$
\delta_G = \delta^{fair}_G \setminus \{(\delta^{fair}_G([q_i, q'_i], a) = (q_i, q'_i) \mid q_i \in F \text{ and } q'_i \notin F'\}
\cup \{(\delta^{fair}_G([q_i, q'_i], a) = [q_i, q'_i] \mid q_i \notin F \text{ and } q'_i \in F'\}.
$$

In a finite game, the protagonist wins if the last state is an antagonist state.
In an infinite game, the protagonist wins if some winning protagonist state $p \in R$ is visited an infinite number of times.

We calculate direct simulations by the algorithm EFFICIENTSIMILARITY [Henzinger et al., 1996] which can be computed in $O(mn)$ (With $m$ being the number of edges, $n$ the number of vertices and $n < m$) time. Another algorithm that solves the same problem and has the same complexity was independently developed by Bloom & Paige [Bloom and Paige, 1996] a year earlier.

We remove SCCs by Algorithm 4.7.6. Note that it uses the term Maximal SCC, this is simply the biggest possible SCC. We search for SCCs topologically from the start state and try to remove the one that is closest to the start state first. If a removal does not change the language, the removal is accepted.

The direct simulation in Line 4 is computed by EFFICIENTSIMILARITY.

Minimization of $B = (\Sigma, Q, q_0, \delta, F)$ in Line 5 is defined as follows. The new minimized automaton $B' = (\Sigma, Q', q'_0, \delta', F')$ (with $X$ being the maximal SCC to remove) is:

$$
Q' = Q \setminus \{q \mid q \in X\}
$$

$$
\delta' = \delta \setminus \{\delta(q, a) = q' \mid q \in X \text{ or } q' \in X\}
$$

$$
F' = F \setminus \{q \mid q \in X\}
$$

$$
q'_0 = \begin{cases} 
q_0 & \text{if } q_0 \notin X \\
q \text{ where } q \text{ direct simulate } q_0 & \text{otherwise}
\end{cases}
$$
Chapter 5

Safra’s construction

Operations on deterministic automata are often easier than the corresponding operations on nondeterministic automata. As we have seen, this property holds for complement constructions of B"uchi automata.

So, is it possible to transform a nondeterministic B"uchi into a deterministic automaton, and take advantage of the much simpler operations? The answer is yes, but there is a catch.

The deterministic automaton cannot be a B"uchi automaton in the general case. As we saw in Corollary 2.1, deterministic B"uchi is less expressive than nondeterministic. Essentially, this means that some nondeterministic B"uchi automata have no deterministic counterpart. Instead, the transformation will be to a deterministic Muller automaton. The complement of a deterministic Muller is an easy construction, but with an exponential blow-up in the acceptance condition.

**Theorem 5.1** Let $M = \langle \Sigma, Q, q_0, \delta, F \rangle$ be a deterministic Muller automaton that accepts the language $L$. Then, the deterministic Muller automaton $M' = \langle \Sigma, Q, q_0, \delta, 2^Q \setminus F \rangle$ accepts $\Sigma^* \setminus L$, that is the complement language of $L$.

However, the transformation from nondeterministic B"uchi to deterministic Muller is hard. In this chapter we will show a version of Safra’s construction [Safra, 1988]. Safra was the first to show a complement construction meeting Michel’s optimal space bound (see Theorem 4.8) which was shown earlier that year. Safra’s construction is well described in his Ph.D. thesis [Safra, 1989] and in the tutorial by Mukund [Mukund, 1996].

5.1 The subset construction

When dealing with automata over finite words, the transformation is done with the subset construction. Each state in the deterministic automaton represents a set of states from the nondeterministic one, the set of states that are reachable by the prefix read so far.

Consider the nondeterministic automaton on finite words in Figure 5.1. It will accept any finite sequence of a’s by “guessing” when it will read the last a and then move to state $q_2$. Figure 5.2 shows the subset construction of the same automaton, with $q_1$ and $q_2$ combined into one state.
Now, think of the automaton in Figure 5.1 as a nondeterministic Büchi. It has an empty language, since there are no loops through the accepting state $q_2$. No run can visit $q_2$ an infinite number of times, however infinitely many different runs visit $q_2$. If we interpret the subset automaton as a Büchi, we see that it wrongly accepts the language $a^\omega$.

Clearly, the subset construction does not work when we are dealing with infinite sequences.

5.2 The marked subset construction

We need some way of keeping track of our visits to final states, to be able to tell if a run should be accepting or not. The *Marked subset construction* does this for us by adding marks to the subsets. Each state in a subset is marked if it either is an accepting state or if the run comes from an accepting state.

If all states in a subset are marked, then this subset is an accepting state in the new automaton. In the transitions out of this state, all marks are reset and the process starts over.

In the subset automaton in Figure 5.2 the start state $q_1$ is not marked. On reading $a$ it will move into the subset $\langle q_1, q_2 \rangle$ with $q_2$ marked since $q_2 \in F$. In the transition from $\langle q_1, q_2 \rangle$ with $q_2$ marked while reading $a$, the transition from $q_1$ is to $\langle q_1, q_2 \rangle$ and $q_2$ marked. From $q_2$ there is no transition, so the marked subset automaton looks as in Figure 5.2.

However, because no run through $q_1$ has an accepting state in its history the state $\langle q_1, q_2 \rangle$ is not fully marked. This means the automaton has no accept
states, and hence correctly has an empty language.

5.2.1 Levels

The marked subset construction still has a problem; There are cases when an accepting run of an automaton does not lead to a fully marked state.

Consider an automaton with an accepting state that is part of a loop with only deterministic transitions on the current input. In the original automaton, this will lead to acceptance. Using marked subsets, it will lead to a subset state with a loop to itself. If this state is not fully marked by the history of the run when entering the loop, it will never be. Hence the constructed automaton will fail to recognize this sequence.

To correct this problem with loops we let each state in the subset construction have a list of levels. Each level is a subset of states from the original automaton, describing one possible run. The first level is equal to the marked subset construction. On the second level are the states that are reachable from the first level when reading the same input again. Only states that are final in the original automaton are allowed below the first level. On the third level are the states that are reachable from the second level by reading the same input again. If two levels are equal then the lowest is an accepting level and all levels below will be empty. There can at most be $n$ nonempty levels if $n$ is the number of states of the original automaton. The automaton accepts a run if it visits an accepting level infinitely often.

5.3 Safra trees

The construction so far still has a problem; consider a level $i$ below the first level. If a run fails then level $i$ in the next state should also be empty. However, if a new run starts in the next state it will be pushed down and the run will not fail.

To correct this we say that each state is a tree of levels. A new run always starts in a new branch of the tree. This way we detect failing runs.

In Safra's construction every state in the deterministic Muller automaton is a Safra tree. First we define the notation for trees.

**Definition 5.1** A tree $T$ is a tuple $(N, n_0, \pi)$ where

- $N$ is a set of nodes.
- $n_0 \in N$, $n_0$ is the root node of the tree.
- $\pi \in T$, $\pi$ is the parenthood function.

**Definition 5.2** In a tree $T$ we say that a node $n$ is an ancestor of the node $n'$ iff there is a $k$ such that $\pi^k(n') = n$. We say that $T$ is ordered iff there is a partial order on the nodes $N$ and for every two nodes $n, n'$ we have either $n \preceq n'$ or $n' \preceq n$.

The ordering property means that we can refer to nodes as being left or right of another node.
Definition 5.3 A Safra tree $S$ is a tuple $(T, S, l, m)$ where

- $T$ is a ordered tree $(N, n_0, \pi)$.
- $S : N \rightarrow 2^\mathbb{Q}$ associate each node with a set of states.
- $l : N \rightarrow \{l_1, l_2, \ldots, l_{2n}\}$ associate each node with a label.
- $m : N \rightarrow \text{final, nonfinal}$ is the marking function.

A Safra tree has the following three properties:

- The union of the sets associated with the children of a node is a proper subset of the set associated with the node, that is for a node $n \in N$ the following holds: $(\bigcup_{n_i \in \{n_i \in N \mid \pi(n_i) = n\}} S(n_i)) \subset S(n)$.
- If two nodes are not related (nodes are related when one is the ancestor of the other) then they are disjoint.
- If $S(n) = \emptyset$ then $n = n_0$.

Now we are ready for Safra’s construction of a deterministic Muller automaton.

Construction: Let $B = \langle \Sigma, Q, \varphi, \delta, F \rangle$ be a nondeterministic Büchi automaton. Construct a deterministic Muller automaton $M_B = \langle \Sigma, Q^S, \varphi^S, \delta^S, F^S \rangle$, where

- $Q^S$ are Safra trees over $Q$,
- $\varphi^S$ is the Safra tree $(T_{\varphi}, \{\varphi\}, \varphi_0, \text{nonfinal})$ where $T_{\varphi}$ is $\langle \{\varphi\}, \varphi_0, \emptyset \rangle$.
- $K_i = \{(N, n_0, \pi), S, m, l_i \mid \exists n \in N : l(n) = l_i \text{ and } m(n) = \text{final}\}$.
- $F_i = \{X \mid X \subseteq K_i\}$.
- $F^S = \bigcup_{1 \leq i \leq n} F_i$.

The transition function

$$\delta^S : Q^S \times \Sigma \rightarrow Q^S$$

from a Safra tree $s = \langle T, S, l, m \rangle$ and an input $a \in \Sigma$ is a new Safra tree $s'$ defined as

$$\delta^S(s, a) = \langle (N', n_0', \pi'), S', l', m' \rangle = s'$$

and is constructed in the following steps:

- Make a copy of $s$ and call it $s'$. For each node $n \in s'$ such that $S'(n) \cap F \neq \emptyset$ add a new rightmost child $n_c$ and associate $S'(n_c)$ with $S'(n) \cap F$. Let $l(n_c)$ be associated with the first free label from $\{l_1, l_2, \ldots, l_{2n}\}$.
- For every node $n \in N'$, apply the subset construction locally:

$$S'(n) = \bigcup_{q \in S'(n)} \delta(q, a).$$
5.3. SAFRA TREES

- For each pair of nodes \( n_1, n_2 \in N' \), remove \( n_2 \) if there is a state \( q \in Q \) such that \( q \in S'(n_1), q \in S'(n_2), n_1 \neq n_2 \) and \( n_1 \) is in a branch to the left of \( n_2 \).

- Remove node \( n \in N' \) and all its descendants if there exists a node \( n' \in N' \) in a branch left of \( n \) such that \( n' \subseteq n \). Remove all nodes labeled with the empty set unless it is the root node.

- For each node \( n \) in \( N' \) such that \( S'(n) = \{ S'(n_j) | n_j \in N' \text{ and } \pi'(n_j) = n \} \), remove all the descendants of \( n \) and let \( m(n) = \text{final} \). For every other node \( n' \in N' \), let \( m(n') = \text{nonfinal} \).

In the first step we add a node for each run that can begin in this state. After the subset step is applied, we remove runs that fail. We always add rightmost nodes, hence the new runs are to the right in the tree. New runs are removed if they are contained in an old run, and therefore a new run can only start if the old one has failed. If the old run failed then the new one does not get the same label and thus it will be separated from the old run.

**Example 5.1** The nondeterministic Büchi automaton \( B \) in Figure 5.3 recognize the language \((a + b)^* (a + ba)^\omega \). This automaton transformed by Safra's construction is the Muller automaton \( M \) shown in Figure 5.4. The sets in \( F^S \) are \{\{6\}, \{7\}, \{6, 7\} \} and \{\{9\}\}.

The input \( ab^\omega \) is accepted by \( B \). In Figure 5.5 the run of \( M \) on the input \( ab^\omega \) is shown. The run on level 1 is not accepting but the run on level 2 is since it will never fail (level 2 will never be empty) and it visits infinitely many final states. The loop is contained in Safra tree \( S6 \) and this belongs to \( F^S \), thus \( M \) is accepting.

The input \( (ba)^\omega \) is not accepted by \( B \). The run of \( M \) on the input \( (ba)^\omega \) is in Figure 5.6. The run on level 1 will not accept since it never visit any final states. On level 2 the run visit final states but will fail since level 2 is empty in \( S8 \) and \( S9 \). Similar holds for the run on level 3. The set of these four Safra trees \{\{5\}, \{7\}, \{8\}, \{9\}\}, this set doesn't belong to \( F^S \) and \( M \) will be rejecting.

5.3.1 Soundness and completeness

Safra's construction is sound, \( B \) accepts all inputs that \( M \) accepts. If \( M \) accepts an input \( \alpha \) then there is an \( F_i \) where all states are visited infinitely many times. The nodes on level \( i \) are marked final and the run on level \( i \) then visits infinitely many final nodes. None of the Safra trees in \( F_i \) has an empty level \( i \) and therefore this is also a run on \( B \). Only nodes that represent some state in \( F \) can be final since only states in \( F \) are allowed below level 1. Hence, the run must visit a state in \( F \) infinitely many times and therefore \( B \) is accepting \( \alpha \).

Safra's construction is also complete. If \( B \) accepts an input \( \alpha \) then there is an infinite sequence of states where some state belongs to \( F \). This corresponds to a set of Safra trees. For a Safra tree in this set there must be an accepting run from \( B \) on some level \( i \). This level is not empty and will be marked final. A node is marked final if the states of the runs that start from this node are the same as the states in the node. This Safra tree will belong to \( K_i \) and the set is a subset to \( K_i \). Hence \( M \) accepts \( \alpha \).
Figure 5.3: A nondeterministic Büchi automaton accepting the language \((a + b)^\omega(a + ab)b^\omega\).

5.4 Complexity

A Safra tree of an \(n\)-state nondeterministic Büchi automaton can be coded with \(O(n \log n)\) bits, thus \(Q^S\) has \(2^{O(n \log n)}\) states. A problem with Muller automata is that \(F^S\) has at least a double exponential number of states, that is at least \(2^{2^n}\). This is because \(F_i\) is the power-set of \(K_i\) and \(K_i\) has an exponential number of states.

The coding of a Safra tree \(s = (T, S, l, m)\) can be done as follows: Given a nondeterministic Büchi automaton with \(n\) states, there can be at most \(n\) nodes in \(s\). Label these nodes \(\{1, 2, \ldots, n\}\) and set \(n_0 = 1\). For each node \(\pi\) can be coded as a list of its children, thus the tree \(T\) can be coded in \(O(n \log n)\) bits. The marking function \(m\) can be coded with a list where each entry is one bit. Both the labeling function \(l\) and \(S\) that associate a node with a subset can be coded as a list where entries are \(\log n\) bits. In \(S\) there is only necessary to list the lowest node a state appears in, since it will appear in every state from the root to the lowest state, thus a state in \(S\) is at most once in \(s\).

5.4.1 How to reach optimal bound

To get the optimal bound with the Safra construction, we need to determinize to a Rabin automaton instead of a Muller automaton. The two automata are similar, but have a key difference in the acceptance condition. In a Rabin automaton, the acceptance set is a pair of state sets \((G, R)\). The automaton is accepting whenever some state in \(G\) is visited an infinite number of times while the states in \(R\) are only visited a finite number of times. In the Safra
construction, this results in \( F_S \) having \( n \) pairs, thus the overall complexity is \( 2O(n \log n) \).

Complementation of a Rabin automaton is done by interpreting the acceptance condition as a Streett automaton and simulating the Streett automaton by a nondeterministic Büchi automaton. The simulation can be done with \( 2\tilde{O}(n \log n) \) states. The Streett acceptance set is also a list of pairs, but the acceptance condition is the dual to the Rabin condition. Both Rabin and Streett are as expressive as a Muller automaton.

Thus, it is possible to complement a nondeterministic Büchi automaton by first using Safra's construction and then using the deterministic algorithm. There is no complexity gain, however; the space complexity is the same as for the general complement algorithms in Chapter 4. The Safra construction is also considered hard to implement due to its complex structure. The only implementation we know about is the one by Tasiran, Hojati & Brayton from 1995 [Tasiran et al., 1995].

This shows however that a deterministic Muller automaton is as expressive as a nondeterministic Büchi automaton and also that the deterministic Muller is equivalent to the nondeterministic Muller.
Figure 5.6: A rejecting run of the automaton on the input \((ba)^\omega\) in Figure 5.4.
Chapter 6

Implementation

We have implemented a Sicstus Prolog library for operations on non-deterministic Büchi automata. The main predicates are nbw_union/3, nbw_intersection/3, nbw_empty/1 and nbw_complement/2. The union, intersection and complement predicates are a bit more efficient than in the definitions, by exploring only reachable states.

Our library also has the predicates nbw_print_dot/1 and nbw_print_dot/2 for writing dot-files, readable by the utilities of the Graphviz package\(^1\).

The source code for our library is listed in Appendix A.

6.1 Design choices

6.1.1 Native Prolog or C/Java?

In Sicstus Prolog, it is possible to call functions written in C or Java. Although this could have resulted in a more efficient program, it would have taken more time to learn the foreign language interface with Prolog. Writing the library in native Prolog would be easier, but with worse performance. We decided to write in native Prolog, to have the time to implement and try out more primitives and explore interesting cases.

6.1.2 Data structure for automata

Our choice of data structure for automata was inspired by a Prolog library for the automata package FSA\(^2\). Each automaton is defined as a compound term nbw(Sigma,Q,Start,Delta,Final) where:

Sigma is a list of unique Prolog ground terms representing the alphabet.

Q is a list of unique Prolog ground terms, representing the states.

Start is a list containing one of the terms in Q.

Delta is a list of transitions. Each transition is a compound term trans(q,a,r)

where q is a term in Q, a is a letter in Sigma and R is a list of terms in Q.

\(^1\)http://www.research.att.com/sw/tools/graphviz/

\(^2\)http://odur.let.rug.nl/~vannoord/Fsa/
Tiny example (weak, non-deterministic, size 2)

Figure 6.1: An automaton accepting $a^\omega$, produced with dot.

`Final` is a list of terms from $Q$, representing the acceptance set.

Figure 6.1 shows the result from dot using the `nbw_dot_output` predicate on the nondeterministic automaton accepting the language $a^\omega$. In our library, it is described with the following compound term:

\[
\text{nbw([a, b],}
\begin{align*}
&[0, 1],
&[0],
&\text{trans}(0, a, [0, 1]),
&\text{trans}(0, b, [1]),
&\text{trans}(1, a, [1]),
&\text{trans}(1, b, [1]),
&[0])
\end{align*}
\]

When graph operations are needed (in the emptiness and weakness tests) we convert our representation to the directed graph representation provided by the Sistus library `ugraphs`.

6.1.3 The predicates

The predicates that construct automata have two things in common. They avoid unreachable states in the resulting automaton by starting at the constructed start state and only exploring reachable states. They also mainly work with the $\delta$ function. Typically, they compute the new $\delta$ function and then extract the set of states from it.

The $\delta$ function is computed by taking a state (initially the start state) and computing the transitions that should be going from that state. The set of
6.1. DESIGN CHOICES

states reachable by these transitions are then added to the queue of states that should be explored, and the transitions are added to the overall \( \delta \) function. To avoid loops, a state is never explored more than once. When the queue of states to search become empty, the new \( \delta \) function is returned.

In our predicate description, plus (as in +B1) before a term indicates that it should be initialized and minus (-B3) that the term should be variable.

\texttt{nbw\_union/3}

The union predicate is implemented according to the construction in Section 3.1.

\texttt{nbw\_union(+B1,+B2,-B3)} (with B1,B2 and B3 being definitions of Büchi automata) will compute an automaton describing the union of B1 and B2 and return it in B3.

\texttt{nbw\_intersection/3}

The intersection is implemented from the construction of Section 3.2 similar to the union implementation.

\texttt{nbw\_intersection(+B1,+B2,-B3)} will assign an automaton describing the intersection of B1 and B2 to B3.

\texttt{nbw\_empty/1}

The emptiness check is performed as outlined in Section 3.3. For graph representation, we use the Sicstus library \texttt{ugraphs}.

\texttt{nbw\_empty(+B)} is true when the automaton B has an empty language.

\texttt{nbw\_weak/1 and nbw\_weak/2}

We check for weakness with Algorithm 4.7.1. There are two variants of this predicate; \texttt{nbw\_weak(+B)} will be true whenever B is a weak automaton. If called with a second argument (\texttt{nbw\_weak(+B,-P)}), P will contain a list of lists of states describing the partial order.

\texttt{nbw\_complement/2}

The complement predicate \texttt{nbw\_complement(+B1,-B2)} is true whenever B2 is the complement of B1. The predicate is actually a wrapper around four different variants of complement.

- **Weak, deterministic**: As we saw in Section 4.7.1, there is a small gain to be made using the weak algorithm on deterministic automata.
- **Deterministic**: As in Section 3.4.
- **Weak**: As in Section 4.7.1.
- **Collapsed (general case)**: As in Section 4.5. This last method produces the largest automata, but will be able to compute the complement of non-weak, nondeterministic automata.
The first suitable variant will be used, and the other will be available via backtracking. The different predicates involved will print a line telling what algorithm is being used.

6.1.4 nbw_print_dot/1 and nbw_print_dot/2

There are two variants of the nbw_print_dot predicates. The first one will accept an automaton as argument and print a dot file describing it. The second will use its second argument as a title for the dot file.

6.2 Verifying our implementation

Verification of the implementation is an interesting issue, since checking language containment rely on having a known correct complement implementation. Without a known good implementation, we put our trust in three heuristics:

- Manual inspection of small cases.
- Manual testing of the primitives that build up each algorithm.
- Verifying that the intersection of an original automaton and its complement is empty.

6.3 Performance

It is hard to find suitable test cases for this kind of problem. We have settled with trying various cases of Michel's worst-case automaton (See theorem 4.8). With our implementation we managed to compute the complement of the $L_2$ automaton (an instance with 3 states). The complementation of $L_3$ ran out of memory after exploring close to 15000 states.

It seems like the structure and size of the $\delta$ function plays a more important role in the overall performance than the number of states. The $\delta$ function has to contain information about the states involved, and will take up memory several times for each state. In $L_2$ the $\delta$ function contained about 5000 states, and $L_3$ ran out of memory describing about 730000 states. Even if we had some kind of pointers linking transitions to states, this would still take up a considerable amount of memory compared to the set of states.

We tried removing the rank optimization from section 4.7.2, and got the expected result: The $\delta$ function grows faster, and the complementation of $L_3$ runs out of memory after exploring about 6000 states. In the case with $L_2$, we managed to compute a complement with a bigger $\delta$ function (of size 6800).
Chapter 7

Conclusion

7.1 Summary

This thesis is a survey of the field of languages over infinite sequences. There is active research going on in this field, during the last year several new results where published.

We have investigated the language containment problem for infinite sequences, with focus on complementation of Büchi automata describing such sequences. This problem has been proved to be in EXPSPACE, so our expectations of what could be done where not high. Some cases of complementation of languages over finite words are also in EXPSPACE, for example complement of nondeterministic automata or when we want a deterministic automaton describing the complement.

7.2 The implementation

We have implemented the algorithms of Kupferman & Vardi in a Prolog library, and seen that worst-case automata (like the one described in Section 4.6) indeed consume a great deal of memory.

On the other hand, all automata are not that hard. Our implementation will try to take advantage of cases when an automaton is deterministic and/or weak, and use more efficient algorithms.

7.2.1 Alternative approaches

We choose the complementation method suggested by Kupferman & Vardi, using weak alternating automata as an intermediate representation. There are other alternatives that should be explored, for example unambiguous B̆uchi automata suggested by Carton & Michel [Carton and Michel, 2003]. Another approach could be to implement Safras construction [Safra, 1988] and make the automaton deterministic before computing the complement. We did not choose this approach, because of the complicated structure of the Safr construction.

Another approach is to avoid B̆uchi automata as representation. The SPIN\footnote{http://spinroot.com/spin/} verification tool use the nondeterministic language PROMELA instead, with the
same expressive power as PLTL. SPIN can use random simulation or exhaustive search of the model state space.

7.2.2 Improvements

We noted that the size and structure of the \( \delta \) function seem to matter more than the number of states. By improving the data structure of this function, it should be possible to get improved performance. Reducing the size of the terms used for state representation by assigning each state a number instead of a list structure could be a start.

The minimization techniques should be fully implemented, to see what they can do to improve the performance. The worst case should still be hard, but in some cases minimization might remove edges turning a non-weak automaton into a weak one and be able to make use of a more efficient algorithm.

The implementation does not deal with automata with several start states. This should be implemented, either by rewriting the algorithms or by converting the start states into one with a subset construction.

The union and complement predicates in the implementation needs a completely defined automaton. This limitation should be resolved. Is

7.3 Future work

Our approach to identify special cases of automata and use specific complementation algorithms for these cases could be explored further. Are there more distinctions than deterministic, weak and general?

7.3.1 Minimization

We did not have the time to implement minimization by simulation. Gurumurthy et al. [Gurumurthy et al., 2003] present results that show that this can be very effective. In their tests, they use a set of 1000 automata with an average size of 6.04 states. After complementation they have a blow-up by thousands of states, but fair simulation reduces this number to automata with an average of 4.08-6.48 states depending on what other minimization techniques are used.

This suggests that most of the states introduced by Miyano & Hayashi's subset construction are unnecessary and can be removed. The current approach is to first complement the automaton and then apply the minimization technique. We must still keep an un-minimized automaton in memory and will still face problems with the space complexity. If we could develop an “on the fly” minimization technique that can identify and remove unnecessary states while computing the complement, we should be able to push the limit of how big automata can be complemented a bit further. It is however not a trivial problem to know “in advance” that a state or transition will be unnecessary in the complementation process.

A minimization technique specifically geared towards making an automaton deterministic or weak (or some other property that simplifies complementation) would also be useful to be able to use more efficient complementation methods.

We saw that the \( \delta \) function plays an important role. There is a minimization technique for alternating automata proposed in Gurumurthy et al.
7.3. FUTURE WORK

[Gurumurthy et al., 2003] by Alur et al. [Alur et al., 1998] that should reduce this problem. We sadly did not have the time to investigate this further.

7.3.2 Transform the specification

It is the specifications that are complemented when checking language containment. Since complementing of a deterministic automaton can be done in polynomial time it would be easy to check systems where all specifications are deterministic. It would be interesting to explore the expressiveness of deterministic specifications and see if it is powerful enough to express the properties of a reactive system.

Another open question is if it is possible to decompose the specification into several smaller cases, which can then be checked more efficiently than the entire specification. The result of the decomposition must be the same as for the overall system, and it is easy to run into difficulties when components of the system depends on each other.
Appendix A

The Sicstus Prolog library

This library is also available at http://www.ida.liu.se/~tcslab/sw/nba/

%%% $Id: nbw.pl,v 1.5 2004/01/12 12:59:56 x03andli Exp $

%%% Exported predicates
:+ module(nbw, [nbw_union/3,
  nbw_intersection/3,
  nbw_complement/2,
  nbw_empty/1,
  nbw_weak/1,
  nbw_weak/2,
  nbw_print/1,
  nbw_print_dot/1,
  nbw_size/2,
  nbw_delta_size/2,
  nbw_delta_minmax/3]).

%%% Sicstus libraries we need.
:+ use_module(library(ugraphs)).
:+ use_module(library(lists)).

%%% Our files.
:+ [union].
:+ [intersection].
:+ [empty].
:+ [dot_output].
:+ [complement].
:+ [weakness].
:+ [utilities].

%%% $Id: union.pl,v 1.3 2003/12/20 11:52:09 x03andli Exp $

%%% Union

nbw_union(B1,B2,U) :-
  B1=nbw(Sigma,_,[B1],D1,F1),
  B2=nbw(Sigma,_,[B2],D2,F2), % Note that union will only accept
  % if the automatas are over the
% same alphabet.

nbw_union_delta([[S1, S2], [], D1, D2, Delta]),
state_set(Delta, Q),
accept_set(Q, F1, F2, F),
U = nbw(Sigma, Q, [[S1, S2], Delta, F]).
intersection_accept_set([I,_,[]]).
intersection_accept_set([Q|Qset],F2,[Q|F]) :=
    Q=[_,Q2,F2],
    member(Q2,F2),
    intersection_accept_set(Qset,F2,F).
intersection_accept_set([Q|Qset],F2,F) :=
    Q=[_,Q2,F2],
    non_member(Q2,F2),
    intersection_accept_set(Qset,F2,F).

intersect_delta(State,T1,T2,F1,_,R) :=
    State = [Q1,Q2,F1],
    member(Q1,F1),
    get_intersect_delta(Q1,Q2,T1,T2,1,1,R).
intersect_delta(State,T1,T2,F1,_,R) :=
    State = [Q1,Q2,F1],
    member(Q1,F1),
    get_intersect_delta(Q1,Q2,T1,T2,1,2,R).
intersect_delta(State,T1,T2,_,F2,R) :=
    State = [Q1,Q2,F2],
    non_member(Q2,F2),
    get_intersect_delta(Q1,Q2,T1,T2,2,2,R).
intersect_delta(State,T1,T2,_,F2,R) :=
    State = [Q1,Q2,F2],
    member(Q2,F2),
    get_intersect_delta(Q1,Q2,T1,T2,2,1,R).
get_intersect_delta(Q1,Q2,T1,T2,N1,N2,Result) :=
    member(trans(Q1,Letter,G1),T1),
    member(trans(Q2,Letter,G2),T2),
cartessian(G1,G2,G),
append_to_all([], G, F),
Result = trans([Q1,Q2,N1],Letter,F).

append_to_all([], []).
\% no_cycles(+NodeList,+Graph)
\% true when no node in NodeList is in a cycle.

\% no_cycles([],).
\% no_cycles([Node|Nodes],Graph) :-
\% not_in_cycle(Node,Graph),
\% no_cycles(Nodes,Graph).

\% The Sicstus lib considers 'current' node to be reachable from
\% itself, even if there is no cycle. Hence, our own version:
\% my_reachable(Node,Graph,Nodeset) :-
\% edge(Graph,Edges),
\% member(Node=Node,Edges),
\% reachable(Node,Graph,Nodeset).

my_reachable(Node,Graph,Nodeset) :-
\% edge(Graph,Edges),
\% non_member(Node=Node,Edges),
\% reachable(Node,Graph,Tmp),
\% delete(Tmp,Node,Nodeset).

\% list_intersection([],[],[]).
\% list_intersection([H|T],L,[H|R]) :-
\% member(H,L),
\% list_intersection(T,L,R).

\% list_intersection([H|T],L,R) :-
\% non_member(H,L),
\% list_intersection(T,L,R).

\% $idi: complement.pl,v 1.9 2004/01/13 08:00:32 z03andli Exp $
\% := [complement_weak].
\% := [complement_collapse].

\% nbw_complement(+DIF, -NEW)
\% when the automaton is deterministic and weak, use
\% the weak algorithm.
\% nbw_deterministic(U1),
\% nbw_weak(U1),
\% U1=nbw(Sigma,_[Start],Delta,Final),
\% format('Running weak complement on a deterministic automaton...''N'',[],]
\% nbwWeakDelta([[Start],[],[],Sigma,Delta,Final,NewDelta],
\% state_set(NewDelta,NewQ),
\% nbw_complement_accept(NewQ,F),
\% R=nbw(Sigma,NewQ,[[Start],[],],NewDelta,F).

\% Use a simpler construction when U1 is deterministic.
\% nbw_deterministic(U1),
\% format('Running deterministic complement...''N'',[],]
\% U1=nbw(Sigma,_[Start],Delta,Final),
\% dbw_complement_delta([[Start,0],[],Delta,Final,NewDelta],
\% state_set(NewDelta,Q),
\% dbw_complement_accept(Q,Final,F),
\% R=nbw(Sigma,Q,[[Start,0]],NewDelta,F).
nbw_complement(U1,R) :-
  % A simpler algorithm when U1 is weak.
  nbw_weak(U1),
  format("// Running weak complement...",[]),
  U1=nbw(Sigma,_,[Start],Delta,Final),
  nbw_weak_delta([[Start],[]],[],Sigma,Delta,Final,NewDelta0),
  state_set(NewDelta,0),
  nbw_complement_accept(Q,F),
  R=nbw(Sigma,[[Start],[]],NewDelta,F).

nbw_complement(U,R,N) :-
  % General case, the collapsed algorithm of Kupferman & Vardi. Assume a nondeterministic automaton.
  U = nbw(Sigma,Q,[Start],Delta,Final),
  format("// Running general complement...",[]),
  length(Q,NewQ),
  length(Final,NewFinal),
  N is (2*(NewQ - NewFinal)),
  nbw_collapsed_delta([[p(Start,N)],[]],[],
    Sigma,Delta,Final,[]),
  state_set(NewDelta,Q2),
  nbw_complement_accept(Q2,F2),
  R=nbw(Sigma,Q2,[[p(Start,N)],[]],NewDelta,F2).

nbw_complement(U,R,N) :-
  % Added the possibility to specify the rank of the start node.
  U = nbw(Sigma,_,[Start],Delta,Final),
  format("// Running general complement with start rank ...",[]),
  nbw_collapsed_delta([[p(Start,N)],[]],[],
    Sigma,Delta,Final,[]),
  state_set(NewDelta,Q2),
  nbw_complement_accept(Q2,F2),
  R=nbw(Sigma,Q2,[[p(Start,N)],[]],NewDelta,F2).

%! The accept set for the general complement
nbw_complement_accept([],[]).

nbw_complement_accept([H|T],F) :-
  H=_[[],[]],
  nbw_complement_accept(T,F).

nbw_complement_accept([H|T],[H|F]) :-
  H=_[[],[]],
  nbw_complement_accept(T,F).

%! The accept set of deterministic complement
dbw_complement_accept([],[]).
dbw_complement_accept([L,G] | T,F,R) :-
  dbw_complement_accept(T,F,R).

dbw_complement_accept([Q,1] | T,F,[Q,1] | R) :-
  non_member(Q,F),
  dbw_complement_accept(T,F,R).

dbw_complement_accept([Q,1] | T,F,R) :-
  member(Q,F),
  dbw_complement_accept(T,F,R).

%! Computes the deterministic complement delta function.
dbw_complement_delta([], [], []).  

dbw_complement_delta([H|T], Explored, Delta, Final, Result) :=  
  non_member(H, Explored),  
  setof(R, dbw_complement_trans(H, Delta, Final, R), Trans),  
  reachable_states(Trans, States),  
  list_difference(States, Explored, NewStates),  
  my_append(T, NewStates, SearchStates),  
  dbw_complement_delta(SearchStates,  
                        [H|Explored],  
                        Delta, Final, Result1),  
  append(Trans, Result1, Result).  

dbw_complement_delta([H|T], Explored, Delta, Final, Result) :=  
  member(H, Explored),  
  dbw_complement_delta(T, Explored, Delta, Final, Result).  

dbw_complement_trans([Q, O], Delta, Final,  
                      trans([Q, O], Letter, [[Goal, O]])) :=  
  member(trans(Q, Letter, [Goal]), Delta),  
  member(Goal, Final).  

dbw_complement_trans([Q, O], Delta, Final,  
                      trans([Q, O], Letter, [[Goal, O], [Goal, 1]])) :=  
  member(trans(Q, Letter, [Goal]), Delta),  
  non_member(Goal, Final).  

dbw_complement_trans([Q, 1], Delta, Final,  
                      trans([Q, 1], Letter, [[Goal, 1]])) :=  
  member(trans(Q, Letter, [Goal]), Delta),  
  non_member(Goal, Final).  

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
% nbw_deterministic(+NEW)  
% True whenever each transition only has a single goal state.  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  

nbw_deterministic(NEW) :=  
  NEW=nbw([_, _, Trans, _]),  
  singleton_trans(Trans).  

singleton_trans([]).  

singleton_trans([H|T]) :=  
  H=trans(_, S),  
  length(S, 1),  
  singleton_trans(T).  

non_empty_list([_|_]).  

print_list([]).  

print_list([H|T]) :=  
  format("p "^S, [H]),  
  print_list(T).  

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
% $Id: complement_weak.pl,v 1.5 2003/12/19 09:24:57 x03andli Exp $  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
% The weak complement.  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  

nbw_weak_delta([], [], []).
nbw_weak_delta([H|T], Explored, Sigma, Delta, Final, Result) :-
  non_member(H, Explored),
  weak_letterloop(Sigma, H, Delta, Final, Trans),
  reachable_states(Trans, States),
  list_difference(States, Explored, NewStates),
  my_append(T, NewStates, SearchStates),
  nbw_weak_delta(SearchStates, [H|Explored], Sigma, Delta, Final, Result),
  append(Trans, Result1, Result).

nbw_weak_delta([H|T], Explored, Sigma, Delta, Final, Result) :-
  member(H, Explored),
  nbw_weak_delta(T, Explored, Sigma, Delta, Final, Result).

nbw_weak_trans(State, Letter, Delta, F, R) :-
  State=[S, [0|0t]],
  weak_coverset(S, Letter, Delta, Sc),
  weak_coverset([0|0t], Letter, Delta, 0c),
  carthesian(Sc, 0c, R1),
  weak_subset(R1, R2),
  weak_remove_f(R2, F, R3),
  R=trans(States, Letter, R3).

nbw_weak_trans(State, Letter, Delta, F, R) :-
  State=[S, []],
  weak_coverset(S, Letter, Delta, Sc),
  weak_pair(Sc, F, R1),
  R=trans(States, Letter, R1).

%/ weak_cover_state_set(+StateSet +Letter +Delta +NewStateSet)
weak_cover_state_set(StateSet, Letter, Delta, R) :-
  weak_cover_state_set(StateSet, Letter, Delta, StateSets),
  weak_combine_state_sets(StateSets, [R]).

weak_cover_state_set([], [], []).
weak_cover_state_set([L|T], Letter, Delta, R) :-
  member(trans(Q, Letter, R1), Delta),
  weak_cover_state_set(T, Letter, Delta, R2),
  append([R1], R2, R).

weak_combine_state_sets([], X, Y) :-
  sort(X, Y). % To make sure the sets [2,1] and [1,2] become the same.
weak_combine_state_sets([H|T], [], R) :-
  weak_combine_state_sets(T, [H], R).
weak_combine_state_sets([H|T], A, R) :-
  non_empty_list(A),
  weak_add_elements(H, A, R2),
  weak_combine_state_sets(T, R2, R).
weak_add_elements([], X, X).
weak_add_elements([H|T], X, A) :-
  weak_add_element(H, X, R1),
  weak_add_elements(T, R1, R).
weak_add_element(_, [], []). weak_add_element(H, [A | T], [A | R]) :=
  member(H, A), weak_add_element(H, T, R).
weak_add_element(H, [A | T], [H | A | R]) :=
  non_member(H, A), weak_add_element(H, T, R).

weak_subset([], []). weak_subset([H | T], [H | R]) :=
  H=[S, 0], subset(0, S), !, \% \leftarrow \text{CUT}
  weak_subset(T, R).
weak_subset([_| | T], [R]) :=
  weak_subset(T, R).

\\% weak_remove_f(+Stateset, +F, -NewStateSet)
\% \leftarrow \text{CUT}
\% \leftarrow \text{CUT}
weak_remove_f([], [], []). weak_remove_f([H | T], F, [[S, 0] | R]) :=
  H=[S, 0], remove_f(0, F, 02),
  weak_remove_f(T, F, R).
remove_f([], [], []). remove_f([H | T], F, [H | R]) :=
  member(H, F),!, \% \leftarrow \text{CUT}
  remove_f(T, F, R).
remove_f([_| | T], F, R) :=
  remove_f(T, F, R).
weak_pair([], [], []). weak_pair([H | T], F, [H, 0] | R) :=
  remove_f(H, F, 0), weak_pair(T, F, R).

weak_letterloop([], [], []). weak_letterloop([H | T], State, Delta, Final, [R1 | R2]) :=
  nbw_weak_trans(State, H, Delta, Final, R1),
  weak_letterloop(T, State, Delta, Final, R2).

\\% $d$: complement_collapsed.pl,v 1.8 2004/01/13 08:00:42 x0Sandii Exp $
%% format("Exploded, Delt size:\"p \"p\"N\", [L, Ds]),

nbw_letterloop(Sigma, H, Delta, Final, [], Trans),
reachable_states(Trans, States),
list_difference(States, Exploded, [], NewStates),
my_append(T, NewStates, SearchStates),
append(Trans, Result, R2),
nbwCollapsed_delta(SearchStates, [\{Exploded\}, Sigma, Delta, Final, R2, X]).

%

nbw_letterloop([], ___, X, R).
nbw_letterloop([H|T], State, Delta, Final, R, X) :-
nbwCollapsed_trans(State, H, Delta, Final, [R1], !, \% \<-- CUT
nbw_letterloop(T, State, Delta, Final, [H|R]|R2], X).
nbw_letterloop([T], State, Delta, Final, R, X) :-
nbw_letterloop(T, State, Delta, Final, R, X).

%

nbwCollapsed_trans(State, Letter, Delta, Final, R) :-
State=[S, []], \% \<-- CUT S is a set of pairs: \{p<q, i>, p<q, i>, ...\}.
coverset(S, Letter, Delta, Final, R1),
pair_and_remove_odd(R1, R2),
R2 = [\_\_], \% We don't want empty transitions.
R = trans([S, []], Letter, R2).

nbwCollapsed_trans(State, Letter, Delta, Final, R) :-
State=[S, 0], \% S and 0 are on the form
\% \{p<q, i>, p<q, i>, ...\}.
coverset(S, Letter, Delta, Final, R1), \% R1 are sets covering S.
coverset(0, Letter, Delta, Final, R2), \% R2 are sets covering 0.
cartesian(R1, R2, R4), \% Pair them together
check_subset(R4, R5), \% Remove the ones that doesn't meet % subset criteria.
remove_odd(R5, R6), \% Remove odd ranks from the 0 side.
R6 = [\_\_], \% Check that this isn't an empty transition.
sort_states(R6, R7), \% Makes more sense to sort here when % we have removed invalid states.
R = trans(State, Letter, R7).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

pair_and_remove_odd([], []).
pair_and_remove_odd([S|T], [[S, 0] | R]) :-
remove_odd_pairs(S, 0),
pair_and_remove_odd(T, R).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

numberset(X, X, [X]) :- !. \% \<-- CUT
numberset(X, Y, [X|R]) :-
X < Y, Z is X+1,
numberset(Z, Y, R).

oddNumberset(X, Y, [X, Y]) :-
X > Y, !. \% \<-- CUT
oddnumberset(X, X, [X]) :- !. %← CUT
oddnumberset(X, Y, [X|R]) :-
    X < Y,
    Z is X+2,
    oddnumberset(Z, Y, R).

% Given a set of states, return all possible state sets that covers % all of the states.
coverset(States, Letter, Delta, Final, R) :-
    % For each state in States, get the set of states that covers it.
cover_state(Coverset(States, Letter, Delta, Final, StateSets),
    % Now combine these sets by taking one from each of them, thus
    % producing sets that cover all states.
    combine_state_sets(StateSets, [R, R]),
    remove_duplicates(R1, R).

% combine_state_sets(+X, +Y, =R)
% X, Y, R are sets of sets of states, for example
% [[{a,b}, {c,d}], [{e,f}, {g,h}]]
% combine_state_sets([X, Y]). % ← sort_state_sets(X, Y).

combine_state_sets([H|T], [R], R) :-
    % If called with an empty list as second argument, use the
    % first element as a "base" for building covering sets.
    % H is on the form {p, q, r, s, t, u, v, w, x, y, z}
    combine_state_sets(T, H, R).

combine_state_sets([H|T], [A1|A2], R) :- % Second argument is split to % not unify with []
    % H is {a,b,c,d}
    add_elements(H, [A1|A2], R),
    combine_state_sets(T, R, R).

sort_state_sets([], []).

sort_state_sets([H|T], [S|R] | R) :-
    H=[S0],
    sort(S, S0),
    sort(R, R),
    sort_state_sets(T, R).

sort_state_sets([], []).

sort_state_sets([H|T], [R|R]) :-
    sort(R, R),
    sort_state_sets(T, R).

% add_elements(+X, +Y, =R)
% X and Y are set of sets of states: [ [ [a,b] [c,d] ] ]
% add_elements([X, X], X).

add_elements([H|T], StateSets, R) :-
    % H should be on the form {a,b, c,d}
    % StateSets should be
    % [ {a,b} <c,d> ], {a,b} <c,d> ]
    % add_element_set(H, StateSets, R),
    add_elements(T, R, R).
%\% add_element_set(+X,+Y,-R)
%\% X is a set of pairs \{\langle q, i \rangle, \langle q, i \rangle, \ldots \}
%\% Y is a set of set of pairs \{\{\langle q, i \rangle, \langle q, i \rangle, \ldots \}\}
%\% add_element_set([\{X\}, X, X].
add_element_set([\{H\}|T], StateSets, R) :-
    add_element(H, StateSets, R1),
    add_element_set(T, R1, R).

%\% add_element(+Element,+Sets,-R)
%\% Element is a set of such pairs. Add Element to each set in Sets.
%\% add_element([\{\}, \{\}].
add_element(Element, [\{StateSet|T\}, [\{R1|R2\}]) :-
    %\% No such element exists, it should be added.
    Element=\{Q,\},
    no_such_q(Q, StateSet), !, \% \leftarrow CUT
    append(StateSet, [Element], R1),
    add_element(Element, T, R2).
add_element(Element, [\{StateSet|T\}, [\{StateSet|R2\}]) :-
    %\% Already present.
    member(Element, StateSet), !, \% \leftarrow CUT
    add_element(Element, T, R2).
add_element(Element, [\{StateSet|T\}, [\{StateSet,R1\}|R2\}]) :-
    %\% An element with lower rank exists.
    Element=\{P,\},
    member(p(P, X), StateSet),
    X < \{\}, !, \% \leftarrow CUT
    delete(StateSet, p(P, X), NewStateSet),
    append(NewStateSet, [Element], R1),
    add_element(Element, T, R2).
add_element(Element, [\{\}, T], R) :-
    %\% Element should not be added (when there exist an element
    %\% with higher rank)
    add_element(Element, T, R).

no_such_q([\{\}], \{\}).
no_such_q(Q, [\{\}|\{\}]) :-
    \= p(Q, \{\}),
    Q \= \{Q2, \},
    no_such_q(Q, T).

%\% cover_stateset(+StateSet +Letter +Delta +Final +Set_of_statesets)
%\% StateSet is on the form \{\langle q, i \rangle, \langle q, i \rangle, \ldots \}.
%\% Return is on the form \{\{\langle a, b \rangle, \langle c, d \rangle, \langle e, f \rangle, \ldots \}\}
%\% cover_stateset([\{\}, \ldots , \{\}].
cover_stateset([\{H\}|T], Letter, Delta, Final, R) :-
    %\% H is on the form \{\langle q, i \rangle\}.
    H=\{Q,\},
    setof(trans(Q, Letter, Set), member(trans(Q, Letter, Set), Delta), Trans),
    ...
reachable_states(Trans, StateSet),

% Here is the rank optimization!
X is I = min(I, 2),
% You might want to replace it with:
% X is 0,
% numberset(I, I, AllNum),
oddnumberset(I, I, EvenNum),
coverset_combine(StateSet, Final, AllNum, EvenNum, R1),
% R1 (the set of states covering H is
% [\{a, b\}, c, d...], [e, d, e, ...]... ]
cover_stateset(T, Letter, Delta, Final, R2),
R - [R1 | R2].

coverset_combine([], [], []): = !.
coverset_combine([H|T], Final, AllNum, EvenNum, R) :=
member(H, Final), !, \% <= CUT
pair_with_numset(H, EvenNum, R1),
coverset_combine(T, Final, AllNum, EvenNum, R2),
coverset_cartesian(R1, R2, R).
coverset_combine([H|T], Final, AllNum, EvenNum, R) :=
pair_with_numset(H, AllNum, R1),
coverset_combine(T, Final, AllNum, EvenNum, R2),
coverset_cartesian(R1, R2, R).

pair_with_numset([], [], []): = !.
pair_with_numset(Q, [H|T], [p(Q, H)|R]) :=
pair_with_numset(Q, T, R).
coverset_cartesian([], [], []).
coverset_cartesian([H|T], [], [[H]|R]) :=
coverset_cartesian(T, [], R).
coverset_cartesian([H|T], [H|T1]|R) :=
cover_append_to_all([H|T], [H|T1], R1),
coverset_cartesian(T, [H|T1], R2),
append(R1, R2, R).
cover_append_to_all([], [], []).
cover_append_to_all(X, [H|T], [R1 | R2]) :=
append(X, H, R1),
cover_append_to_all(X, T, R2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% check_subsets(+Set of pairs, -Set of subset pairs)
% With each pair \langle S, 0 \rangle, check that 0 is a subset of S.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
check_subsets([], []).
check_subsets([[S, 0]|T], [[S, 0]|R]) :=
subset(0, S), !, \% <= CUT
check_subsets(T, R).
check_subsets([[], |T]|R) :=
check_subsets(T, R).
subset([], []).
subset([H|T], S) :=
member(H, S),
subset(T, S).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% remove_odd(+Set of pairs, -new set of pairs)
% With each pair \(<S_0, O_0>\), make sure that the \(O_0\) side does not contain any state with odd rank.

remove_odd([], []). remove_odd([\([H]_T, [[S, New0] | R]\)] :-
\(N=[S,0]\),
remove_odd_pairs([0, New0]),
remove_odd([T,R]).

remove_odd_pairs([], []). remove_odd_pairs([\([H]_T, [[S] | R]\)] :-
\(N=[S,0]\),
0 is \(N \mod 2\), \(\forall X < \leftarrow \text{ cut}
\)
remove_odd_pairs([T,R]).

% $Id: weakness.pl,v 1.7 2003/12/20 11:52:38 xosandli Exp$

nbw_weak(NBW) :-
NBW=nbw([], [], [Start], [], F),
nbw_to_graph(NBW, Graph),
nbw_size(NBW, N),
partial_order([Start, [], Graph, F, [[Start, N]], []]).

nbw_weak(NBW,P0) :-
NBW=nbw([], [], [Start], [], F),
nbw_to_graph(NBW, Graph),
nbw_size(NBW, N),
partial_order([Start, [], Graph, F, [[Start, N]], N]),
po_swap(N, Swap),
sort(Swap, Sort),
reverse(Sort, Rev),
po_list_to_sets(Rev, P0).

% partial_order(State,F,NodeNumbers,N)

% partial_order([], [], [], NodeNumbers, NodeNumbers) :-
% % No failing states found. Either all of them are explored,
% % or we have found a partial order. A final check:
% po_consistent(NodeNumbers, NodeNumbers, Graph).

partial_order(State, Explored, Graph, F, NodeNumbers, P0) :-
member([State, N], NodeNumbers),
my_reachable(State, Graph, NodeSet),
po_number(NodeSet, F, State, N, NumberSet),
numberset_union(NumberSet, NodeNumbers, NewNumberSet, Explored, Graph, FailState),
partial_order(FailState, [State | Explored], Graph, F, NewNumberSet, P0).

% po_consistent(NodeSet, Graph)

% po_consistent([], []). po_consistent([[H] | T], NumberSet, Graph) :-
H=[\([Q, N]\],
neighbours(Q, Graph, NeighbourSet),
consistent_neighbours(NeighbourSet, NumberSet, N),
po_consistent(T, NumberSet, Graph).
%\% failing_state

%\% Use [] to indicate no failing states.

failing_state([H|T],NumberSet,Explored,Graph,N) :-
  N=[1,...,1],
  member(q,Explored), !\% \(< \) CUT
  failing_state(T,NumberSet,Explored,Graph,N).

failing_state([H|\_],NumberSet,\_,Graph,q) :-
  N=[q,N],
  neighbours(q,Graph,NeighbourSet),
  any_inconsistent_neighbour(ExploredSet,N,NumberSet), !\% \(< \) CUT
  failing_state(T,NumberSet,Explored,Graph,N).

%\% consistent_neighbours

%\% We have at least one inconsistency. Cut!

%\% We have at least one inconsistency. Cut!

%\% enumerated_states(+States +Final +N +Numbered states)
%\% Enumerate states according to our algorithm.

po_number([],\_,\_,\_).
po_number([H|T],F,State,N,[[H,N]|\_] ) :-
  non_member(State,F),
  non_member(N,F),
  po_number(T,F,State,N,\_).
po_number(H[T],F,State,N,[[H,N] | R]) :-
  non_member(State,F),
  member(N,F),
  K is N=1,
  po_number(T,F,State,N,R).

po_number(H[T],F,State,N,[[H,N] | R]) :-
  member(State,F),
  non_member(H,F),
  K is N=1,
  po_number(T,F,State,N,R).

po_number(H[T],F,State,N,[[H,N] | R]) :-
  member(State,F),
  member(H,F),
  po_number(T,F,State,N,R).

% Below are utility functions and a simple pretty-printer.

% po_list_to_sets(+NodeNumbers,-StateSets)
po_list_to_sets([],[]).
po_list_to_sets([H|T], [R1|R]) :-
  H=[N,1],
  setof(X, member([N,X],[H|T]),R1),
  po_remove_rank([H|T],N,NewList),
  po_list_to_sets(NewList,R).

po_remove_rank([],_,[R]).
po_remove_rank([H|T],N,[H|R]) :-
  H=[N,1],
  X \= N,
  po_remove_rank(T,N,R).

po_remove_rank([H|T],N,R) :-
  H=[N,1],
  po_remove_rank(T,N,R).

% po_print(+NodeNumbers +Final)
% pretty-printer for node-number sets.
po_print(L,F) :-
  po_swap(L,S),% List is now on the form <n,state>.
  sort(S,Sorted),
  po_list_print(Sorted,O,F).

po_swap([],[]).
po_swap([Q,N]|T,[[N,Q] | R]) :-
  po_swap(T,R).

po_list_print([],_,_).
po_list_print([H|T],N,F) :-
H=[N,Q],
format(""p",[Q]),
po_list_print(T,N,F).

po_list_print([H|T],X,F) :-
H=[N,Q],
X \= N,
member(Q,F),
format(""N:p: (accept) "p ",[N,Q]),
po_list_print(T,N,F).

po_list_print([H|T],X,F) :-
H=[N,Q],
X \= N,
non_member(Q,F),
format(""N:p: "p ",[N,Q]),
po_list_print(T,N,F).

% $Id: dot_output.pl,v 1.7 2004/01/12 12:59:04 xoSandli Exp $

% nbw_print_dot(+NBW,Title)
% nbw_print_dot(NBW) := nbw_print_dot(NBW,"No title").

nbw_print_dot(NBW,Title) :-
NBW=nbw(_,States,[Start],Delta,Final),
% Init stuff for dot
format("digraph finite_state_machine {"N",[]),
% We print at (roughly) A4 paper.
format("  size = \"8,9\";center=\"\";\"N",[]),
% Label for the graph
dot_status_string(NBW,Title),
% The start node.
format("  node [shape=plaintext,label=\"\";start;"N",[]),
% The start state should be number 0.
delete(States,Start,Tmp),
dot_states([Start|Tmp],0,Final,Delta,TempDelta),
expand_delta(TmpDelta,NewDelta),
% Transition to start node
format(""N start -> 0"N",[]),
print_dot_trans(NewDelta),
format(""N",[]).

dot_status_string(NBW,Title) :=
nbw_size(NBW,Size),
nbw_weak(NBW),
nbw_deterministic(NBW), % Out!
format("\"s (weak, deterministic, size "p")\";"N", [Title,Size]),!.

dot_status_string(NBW,Title) :=
APPENDIX A. THE SICSTUS PROLOG LIBRARY

nbw_size(NBW,Size),
nbw_weak(NBW),
format(" label="s (weak, non-deterministic, size "p")"; "N",
[Title,Size]) !.

dot_status_string(NBW,Title) :-
nbw_size(NBW,Size),
format(" label="s (non-weak, non-deterministic, size "p")"; "N",
[Title,Size]) !.

dot_states([],_,X,X).

dot_states([H|T],N,Final,Delta,NextDelta) :-
member(H,Final),
format(" node [shape="doublecircle",label="p"] ; "p; "N",
[N,N]),
replace_statename(H,N,Delta,TmpDelta),
S is S + 1,
dot_states(T,S,Final,TmpDelta,NextDelta).

dot_states([H|T],N,Final,Delta,NextDelta) :-
non_member(H,Final),
format(" node [shape="circle",label="p"] ; "p; "N",
[N,N]),
replace_statename(H,N,Delta,TmpDelta),
S is S + 1,
dot_states(T,S,Final,TmpDelta,NextDelta).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Expand_delta(+delta, -newdelta) expands all nondeterministic
% choices in a delta function.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
expand_delta([],[]).

expand_delta([H|T],R) :-
H=trans(State,Letter,Stateset),
setof(R, expand_trans(State,Letter,Stateset,R),R1),
expand_delta(T,R2),
append(R1,R2,R).

expand_trans(Q,L,[H|_],R) :-
H=trans(Q,L,[H]).

expand_trans(Q,L,[T|_],R) :-
expand_trans(Q,L,T,R).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Group delta transitions by letter (for drawing neat automata with
% graphviz)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

print_dot_trans([]).
print_dot_trans([Trans|Delta]) :-
Trans = trans(State,_,[StateSet]),
setof(trans(State,L,[StateSet]),
member(trans(State,L,[StateSet]),[Trans|Delta]),T),
setof(L,member(trans(S,L,T),Letters),
format(" p -> p [label="p",[State,StateSet]],
print_letters(Letters),
format(" l",[T]),
delete_list(T,Delta,NewDelta),
pin dot trans(NewDelta).
delete_list([], X, X).
delete_list([H|T], X, Y) :-
  delete(X, H, Z),
  delete_list(T, Z, Y).

print_letters([], []).
print_letters([H|T]) :-
  format("p", [H]),
  print_letters(T).

replace_statename(OldState, NewState, [OldTrans|T1], [NewTrans|T2]) :-
  OldTrans = trans(OldState, Letter, StateSet),
  replace_in_stateSet(OldState, NewState, StateSet, NewStateSet),
  NewTrans = trans(NewState, Letter, NewStateSet),
  replace_statename(OldState, NewState, T1, T2).

replace_in_stateSet([], [], []).
replace_in_stateSet(OldState, NewState, [OldState|T1], [NewState|T2]) :-
  replace_in_stateSet(OldState, NewState, T1, T2).
replace_in_stateSet(OldState, NewState, [H|T], [H|T2]) :-
  OldState = H,
  replace_in_stateSet(OldState, NewState, T, T2).

nbw_print(NBW) :-
  NBW = nbw(States, Start, Delta, Final),
  format("#Sigma : " , [Sigma]),
  format("#States : " , [States]),
  format("#Start state : " , [Start]),
  format("#Accept states : " , [Accept]),
  format("#Delta : " , [Delta]),
  print_delta(Delta),
  format("#Final : " , [Final]).
Appendix A. The Sicstus Prolog Library

?-trans(State, Letter, Goal),
   format(" p", [State, Letter, Goal]),
   print_delta(T).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Useful functions for showing the operations.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

print_intersection(U1, U2) :- print_intersection(U1, "A", U2, "B").

print_intersection(U1, T1, U2, T2) :-
   nbw_print_dot(U1, T1),
   format("  "-----------------------"N", []),
   nbw_print_dot(U2, T2),
   format("  "-----------------------"N", []),
   nbw_intersection(U1, T1, U2, T2),
   append("The intersection of ", T1, T2),
   append(T, ", and ", T3),
   append(T3, T2, Text),
   nbw_print_dot(I, Text).

print_union(U1, U2) :- print_union(U1, "A", U2, "B").

print_union(U1, T1, U2, T2) :-
   nbw_print_dot(U1, T1),
   format("  "-----------------------"N", []),
   nbw_print_dot(U2, T2),
   format("  "-----------------------"N", []),
   nbw_union(U1, T1, U2, T2),
   append("The union of ", T1, T2),
   append(T, ", and ", T3),
   append(T3, T2, Text),
   nbw_print_dot(I, Text).

print_complement(U1) :- print_complement(U1, "A").

print_complement(U1, T1) :-
   nbw_print_dot(U1, T1),
   format("  "-----------------------"N", []),
   nbw_complement(U1, U2),
   append("The complement of ", T1, T2),
   nbw_print_dot(U2, Text).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Std: utilities.pl, v 1.5 2004/02/11 16:22:47 www InterruptedException$%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% nbw_to_graph(NBW, -Graph)
% Converts an nbw into a prolog ugraph.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

nbw_to_graph(NBW, G) :-
   NBW = nbw(_, _, Trans, _),
   trans_to_edges(Trans, Edges),
   vertices_to_igraph([], Edges, G).

trans_to_edges([], []).

trans_to_edges([E | Ts], E) :-
   H = trans(State, Term, Goal),
   edge_to_list(State, S1, S2),
   edge_to_list(Term, Term1, Term2),
   edge_to_list(Goal, Goal1, Goal2),
   append(S1, S2, S1_E),
   append(Term1, Term2, Term_E),
   append(Goal1, Goal2, Goal_E),
   edge_to_list(_[E1, E2], E1_E, E2_E),
   edgelist(E1, E2, E),
   append(E1, E2, E).
edge_list([S1, [S2] | T], [S1, S2 | E]) :-
   edge_list(S1, T, E).

carthesian([], [], []).% Carthesian product

carthesian([X1 | X2], [Y1 | Y2], Z) :-
   carthesian(X2, Y2, Z2),
   combine_with_all(X1, [X1, Y1 | Z2]).

combine_with_all([], [], []).% combine with all
combine_with_all([X | Xs], [Y | Ys], Z) :-
   combine_with_all(X, [X | Ys | Z], Z).

reachable_states(+Trans, -States) :=
   reachable_states([], []).% given a transition function, computes the states
% that are reachable with it.
reachable_states([S1 | S2], L) :-
   reachable_states(S2, L1),
   reachable_states(S1, L2),
   append(L1, L2, L).

my_append([], S, S).% my append
my_append([X | Xs], S, L) :-
   member(X, S), !, % cut
   my_append(Xs, S, L).

my_append([], [X | Xs], L) :-
   append(X, Xs, L).

state_set(States, State) :=
   state_set([], State),
   state_set(State, State).
state_set([S | Ss], State) :-
   reachable_states(S, L),
   state_list_union(L, State),
   state_set(Ss, State).

state_list_union([], []).% state list union
state_list_union([S | Ss], State) :-
   state_list_union(S, State),
   state_list_union(Ss, State).

list_difference([X | Xs], [Y | Ys], Diff) :-
   member(X, Ys), !, % cut
   list_difference(Xs, Ys, Diff).% list difference
list_difference(X, Y, [], Diff).
list_difference([], [], R).
list_difference([H|T], L2, X, R) :-
  member(H, L2),! , X <- CUT
  list_difference(T, L2, X, R).
list_difference([H|T], L2, X, R) :-
  list_difference(T, L2, [H|X], R).

% some_member(+List1,+List2)
% Some element in List1 is in List2.
some_member([H|_], L) :- member(H, L),! . X <- CUT
some_member([_|T], L) :- some_member(T, L).

% no_member(+List1,+List2)
% No element in List1 is in List2.
no_member([], []).
nomeber([H|_]).
nomeber([H|T], L) :- non_member(H, L), no_member(T, L).

% nbw_size(+NBW,-Size)
nbw_size(NBW, Size) :-
  NBW=nbw(_, Q, _,_),
  length(Q, Size).

% nbw_delta_size(+NBW,-Size)
nbw_delata_size(NBW, Size) :-
  NBW=nbw(_, _, Delta, _),
  count_delta(Delta, Size).

count_delta([], 0).
count_delta([H|T], R) :-
  H=trans(_, _, Stateset),
  length(Stateset, N1),
  count_delta(T, R2),
  R is R1+N2.

% nbw_delata_minmax(+NBW,-min,-max)
nbw_delata_minmax(NBW, Min, Max) :-
  NBW=nbw(_, Q, _, Delta, _),
  length(Q, N),
  count_minmax(Delta, N, 0, Min, Max).

count_minmax([], Min, Max, Min, Max).
count_minmax([H|T], Min, Max, R1, R2) :-
  H=trans(_, _, Stateset),
  length(Stateset, N),
  R < Min!, X <- CUT
count_minmax(T, R, Max, R1, R2).

\begin{align*}
\text{count_minmax}([H|T], \text{Min}, \text{Max}, \text{R1}, \text{R2}) & := \\
& \text{H-trans}(_, _, \text{Stateset}), \\
& \text{length}(\text{Stateset}, R), \\
& R > \text{Max}, /, \% \leftarrow \text{CUT} \\
& \text{count_minmax}(T, \text{Min}, R, \text{R1}, \text{R2}).
\end{align*}

\begin{align*}
\text{count_minmax}([_ | T], \text{Min}, \text{Max}, \text{R1}, \text{R2}) & := \\
& \text{count_minmax}(T, \text{Min}, \text{Max}, \text{R1}, \text{R2}).
\end{align*}
Bibliography


BIBLIOGRAPHY


**Abstract**

This thesis is a survey of the field of languages over infinite sequences. There is active research going on in this field, during the last year several new results where published.

We investigate the language containment problem for infinite sequences, with focus on complementation of Büchi automata. Our main focus is on the approach with alternating automata by Kupferman & Vardi. The language containment problem has been proved to be in EXPSPACE. We identify some cases when we can avoid the exponential blow-up by taking advantage of properties of the input automaton.

Some of the algorithms we explain are also implemented in a Sicstus Prolog library.

**Keyword**

Infinite sequences, Alternating automata, Complementation, Omega-regular, Büchi automata