UNIVERSITÉ DE FRIBOURG SUISSE
UNIVERSITÄT FREIBURG SCHWEIZ

Implementation of an Algorithm for Büchi Complementation

Christian Göttel
Rte. des Pommiers 80
CH-1723 Marly
<christian.goettel@unifr.ch>

November 17, 2013

Assistant: Joël Allred
Supervisor: Prof. Dr. Ulrich Ultes-Nitsche

Keywords: Algorithm, Automata, Büchi, Complementation

Bachelor Thesis

Department of Informatics - University of Fribourg

Téléphone: +41 (26) 300 84 65
Télécopie: +41 (26) 300 97 26
E-Mail: <diuf-secr@unifr.ch>
Site web: http://diuf.unifr.ch

Adresse:
Département d'Informatique
Université de Fribourg
Boulevard de Pérolles 90
1700 Fribourg, Suisse

Telefon: +41 (26) 300 84 65
Fax: +41 (26) 300 97 26
E-Mail: <diuf-secr@unifr.ch>
Webseite: http://diuf.unifr.ch

Adresse:
Departement fürInformatik
Universität Freiburg
Boulevardde Pérolles 90
1700 Freiburg, Schweiz

Phone: +41 (26) 300 84 65
Fax: +41 (26) 300 97 26
E-Mail: <diuf-secr@unifr.ch>
Website: http://diuf.unifr.ch

Address:
Department of Informatics
University of Fribourg
Boulevard dePérolles 90
1700 Fribourg, Switzerland

Contents

1. Introduction

Complementing Büchi automata - a complex process, which has not attracted a lot of attention over the last decades. Büchi complementation is missing simpler algorithms compared to complementation of non-deterministic finite automata on finite words [7, p. 261], a problem described by the authors of the paper *State of Büchi Complementation* [7]. My thesis is mainly based on the statements and conclusions described in this paper.

Source of the Büchi complementation difficulty is the $n!$ blow-up of automata, so described by Michel in 1988 [6, 7, p. 261]. Yan sharpened the lower bound Büchi complementation blow-up to $(0.76n)^n$ [11, p. 11], while Friedgut, Kupferman and Vardi found an upper bound of $(0.96n)^n$ [3, p. 12]. Finally Schewe "[...] concluded the quest for optimal Büchi complementation [...]" [9, p.661] by proposing an algorithm close to the lower bound. There are four approaches for the classification of complementation constructions: the Ramsey-based approach, the determinization-based approach, the rank-based approach and the slice-based approach [7, p. 262]. Vardi et al. found that Büchi complementation lacks empirical studies "[...] to evaluate the performance of these complementation approaches" [7, p. 262]. For this reason they compared the performance of the different approaches and suggested different optimization heuristics. At the Department of Informatics [1] of the University of Fribourg [2] (Switzerland) professor Ultes-Nitsche and his PhD student Joël Allred have developed a new slice-based Büchi complementation algorithm, that needs to be evaluated for performance.

In my bachelor thesis I am implementing the new slice-based Büchi complementation algorithm and evaluating its performance based on the two test sets presented in Vardi e.a. paper [7].

The new complementation algorithm makes use of tuple notation for the labels of states of the complementary Büchi automaton. The algorithm separates the complementary Büchi automaton into two parts, to which I will refer as *finite part* and *infinite part*. The finite part represents the part of the complementary Büchi automaton, where it spends a finite number of transitions on ω-words. It is implemented by modified subset construction, where states are combined to sets of states similar to the subset construction. The modification of the subset

[1] `http://diuf.unifr.ch`
[2] `http://www.unifr.ch`

construction is due to a *splitting* 2.4, that processes sets of states further. The splitting deals with so called *"mixed sets of states"* and preserves all other sets. These mixed sets of states are split into sets of non-accepting and accepting states. The automaton obtained at the end of the construction of the first part will be referred to as labeled transition system $\mathcal{T}'$, since it has no accepting condition.

The infinite part of the complementary Büchi automaton is also obtained by modified subset construction except that we add colors to all sets of states. These colors are related to information about the set of states that will be needed to determine the accepting condition. There are three colors a set of states can have: ordinary, on hold and discontinued. I will refer to the automaton constructed in the infinite part as labeled transition system $\mathcal{T}''$. The complementary Büchi automaton is finally obtained by connecting the labeled transition systems $\mathcal{T}'$ and $\mathcal{T}''$ and applying the accepting condition on the tuples in the infinite part. A state in $\mathcal{T}''$ is accepting if its label (e.g. the tuple) does not have a discontinued set of states.

The implementation consists of a library and a command line tool written in C for performance reasons. The library provides basic functions to manipulate Büchi automata as well as ordinary automata. Part of the library is a small reference tool, which takes as input one or multiple Büchi automata that are manipulated according to the algorithm parameters and writes them back to disk as output. In order to evaluate the performances of the Büchi complementation algorithms, different non-deterministic Büchi automata will be complemented, whose complementary Büchi automaton is already known.

Chapter 2 outlines the problem of complementation with a simple example of an ordinary automaton in 2.2 before going into the more complex Büchi complementation and elaboration of the new Büchi complementation algorithm. In chapter 3 follows a description of the practical part of the bachelor thesis, consisting of the library and the tool used to complement Büchi automata. The last two chapters 4 and 5 evaluate the performance and results of this new Büchi complementation algorithm to other complementation algorithms of Büchi automata and finish with a conclusion.

The reader will often encounter the word *automaton* in my thesis. To avoid any confusion, in chapters 3, 4 and 5 I will talk exclusively about *Büchi automata* and for reasons of simplicity only use the word "automata". In chapter 2 on the other hand, where I am talking about *ordinary and Büchi automata*, I will explicitly point out the type of automaton which are being discussed. In my thesis I completely ignore the existence of $\epsilon-$transitions, since they are not of relevance for the topic - algorithms are adapted accordingly.

2. The Büchi complementation algorithm

2.1. Notation and properties of ordinary finite automata and finite automata in general

The naming and notation of finite automata properties that follows in this section is inspired by *"Introduction to Automata Theory, Languages and Computation"* written by Hopcroft, Ullman and Motwani [4], a standard work on Automata Theory.

An ordinary automaton $\mathcal{A}$ is described by 5-tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ [4, p. 46], having the following meaning:

- Q is the set of all states of $\mathcal{A}$ and $q_i \in Q$ is the ith state
- Σ is the alphabet of $\mathcal{A}$ and $\sigma_j \in \Sigma$ is the jth symbol
- $\delta : Q \times \Sigma \to 2^Q$ is the transition function, given by the Cartesian product $Q \times \Sigma$ (transition table)
- $q_0 \in Q$ is the initial state (an automaton can have more than one initial state, in which case a set notation would be used)
- $F \subseteq Q$ is the set of accepting states with $q_k \in F$ being an accepting state

An automaton is called *non-deterministic* if $\exists q \in Q, \exists \sigma \in \Sigma : |\delta(q_i, \sigma)| > 1$ and *deterministic* if $\forall q \in Q, \forall \sigma \in \Sigma : |\delta(q_i, \sigma)| \leq 1$ (e.g. *deterministic* and *complete* if $\forall q \in Q, \forall \sigma \in \Sigma : |\delta(q_i, \sigma)| = 1$) applies.

Ordinary automata accept only finite sequences of symbols called *finite words*. The empty word ϵ is a word having zero occurrences of symbols. Words are classified by their length, given by the *number of positions* for symbols in the word, but commonly referred to by the *number of symbols* in the word [4, p. 29]. The word "010" for example has length 3 on the binary alphabet $\Sigma = \{0, 1\}$. The set of all words of a given length k, called *power of an alphabet* [4, p. 29], is denoted as Σ^k. The union of all powers of an alphabet is described by $\Sigma^+ = \bigcup_{i=1}^{n} \Sigma^i$, with respect to the empty word by $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$.

Given a finite word $w = \sigma_0\sigma_1 \cdots \sigma_n \in \Sigma^n$, passing it to the automaton $\mathcal{A}$ would give a run $\rho = q_0q_1 \cdots q_i \in Q^*$. States q_j that appear in the run sequence are called *live* states, all others are called *dead* states [7, p. 261-262]. A finite word is accepted by an automaton if the run ρ is accepting. An accepting run is a run, where the automaton stops in an accepting state $q_k \in F$ of the accepting set.

The language L of an automaton $\mathcal{A}$, denoted $L(\mathcal{A})$, is the set of all accepted words by the automaton. Ordinary automata recognize words in the regular language, which means that the regular language L has the relation $L(\mathcal{A}) \subseteq L$ to the language of ordinary automata. It is therefore possible to express ordinary automata by regular expressions.

2.2. The complementation of ordinary automata by subset construction

Summary: a complete DFA can be complemented by simply switching its accepting and non-accepting states.

A complete DFA can be obtained by either subs_constr(complete(NFA)) or complete(subs_constr(NFA))

The second way is recommended because it doesn't increase the size of the input to the subset construction.

NFA to complement DFA

In this section I am describing in more detail the complementation of ordinary automata. This allows me to explain the necessary complementation steps as well as how to keep the size of the complement ordinary automaton low. These findings can also be applied to the slice-based Büchi complementation algorithm described in section 2.4, with results in section 4.2.4.

We will begin with the non-deterministic finite automaton (NFA) $\mathcal{N}$ in figure 2.1 described by the tuple $\mathcal{N} = (Q_\mathcal{N}, \Sigma, \delta_\mathcal{N}, q_0, F_\mathcal{N})$. The set of states is $Q_\mathcal{N} = \{0, 1\}$, the alphabet is $\Sigma = \{a, b\}$, the transition function is $\delta_\mathcal{N} : Q_\mathcal{N} \times \Sigma \to 2^{Q_\mathcal{N}}$, the initial state is $q_0 = 0$ and the set of accepting states is $F_\mathcal{N} = \{1\}$. This automaton recognizes the regular expression $RE_\mathcal{N} = (a + b)^*bb^*$ on finite words, meaning all words that end with a finite number of bs.

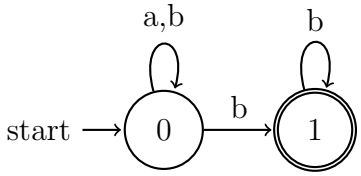


Figure 2.1.: Example NFA $\mathcal{N}$

In order to find all words which are not accepted by a given ordinary automaton, it is first completed and then complemented, since complementation requires a complete automaton [4, p. 135]. A complete deterministic or complete non-deterministic ordinary automaton is an automaton, where each symbol σ of the alphabet Σ has to leave all states, thus for a deterministic finite automaton $\forall q \in Q, \forall \sigma \in \Sigma, |\delta(q, \sigma) = 1|$ and $\delta(q, \sigma) = p$, with $p \in Q$ applies. There are two

complete: in every state a leaving transition for every symbol of the alphabet.

methods to obtain a complete deterministic automaton from the non-deterministic automaton $\mathcal{N}$: by subset construction or by completion and then subset construction. By doing only the subset construction there is already a good chance to obtain a complete deterministic automaton, however, it is not guaranteed. In order to keep the size of the complement ordinary automaton low it is a good idea to do directly a subset construction instead of completing the non-deterministic ordinary automaton first and then applying a subset construction. If we look at the sizes of the complement automata we obtain by applying both methods, we will see that completing the non-deterministic ordinary automaton first might yield a deterministic ordinary automaton which is twice as large. This factor of 2 is due to the size complexity of the subset construction algorithm, that can produce up to 2^n states. By first completing the non-deterministic ordinary automaton we increase the complexity to 2^{n+1}, which explains the factor 2. This means that unnecessary computation can be avoided, which would lead to a noticeable longer time for determinization of larger finite automata.

Algorithm 2.2.1: Subset construction

input : Working list W holding unmarked sets of states of T
output: A DFA $\mathcal{D} = (T, \Sigma, \delta_{\mathcal{D}}, \{q_0\}, U)$

```
T ← {q0};                    /* T: set holding sets of states of D */
W ← {q0};
while W ≠ ∅ do
    s ∈ W;                       /* select a set of states s from W */
    W ← W \ s;                   /* mark s by removing it from W */
    for σ ∈ Σ do
        t ← δN (s, σ);       /* compute the successor set of states */
        if t ∉ T then
            T ← T ∪ t;           /* add new set of states t to T */
            W ← W ∪ t;             /* t is unmarked, add it to W */
        end
        δD (s, σ) ← t;           /* update the DFA transition table */
    end
end
```

The NFA $\mathcal{N}$ is thus transformed into a deterministic finite automaton (DFA) by subset construction [5, p. 121]. A DFA has usually more transitions and states than a NFA representing the same language. In the worst case, if a NFA has n states, a DFA can have up to 2^n states, whereas some states might not be reachable [4, p. 60]. By subset construction of the complete NFA $\mathcal{N}_{\mathcal{C}}$ in figure 2.2 a complete DFA $\mathcal{D}_{\mathcal{C}} = (T, \Sigma, \delta_{\mathcal{D}_{\mathcal{C}}}, q_0, U)$ is obtained, with S the set of all sets of

states, $T \subseteq S$ is a subset of the set of all sets of states because not all states of $\mathcal{D}_\mathcal{C}$ are reachable from q_0 in general. The initial state q_0 remains the same by becoming the set $\{q_0\}$ and U is the set of states having one or more accepting states q_k with $U \subseteq S$ such that $s_i \in S$ and $q_k \in s_i$. It should be noticed that a subset construction of a complete NFA yields a complete subset construction with a complete DFA having at most 2^n states. Hopcroft, Motwani and Ullman did a well described summary of the time complexity of the subset construction algorithm for an ordinary automaton with n states and found $O\left(n^3 2^n\right)$ as upper time bound [4, p. 151-152].

To show the difference between both methods, we first complete the NFA $\mathcal{N}$ in figure 2.1 and obtain the complete NFA $\mathcal{N}_\mathcal{C}$ in figure 2.2 denoted $\mathcal{N}_\mathcal{C} = (Q_{\mathcal{N}_\mathcal{C}}, \Sigma, \delta_{\mathcal{N}_\mathcal{C}}, q_0, F_{\mathcal{N}_\mathcal{C}})$.

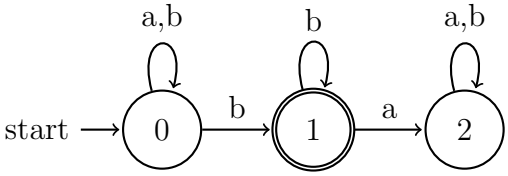
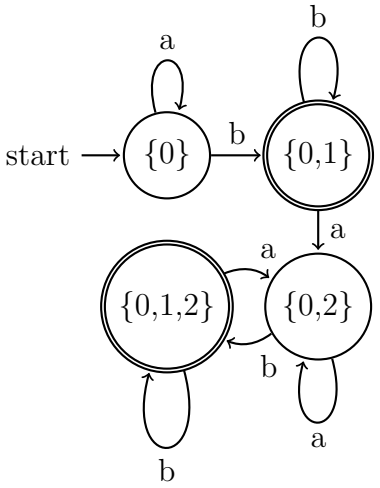


Figure 2.2.: Complete NFA $\mathcal{N}_\mathcal{C}$

The following two DFAs in figure 2.3 and figure 2.4 are obtained, by applying the subset construction to both non-deterministic ordinary automata $\mathcal{N}$ and $\mathcal{N}_\mathcal{C}$:

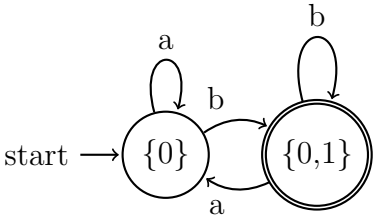


(2.3.1) Complete DFA $\mathcal{D}''_\mathcal{C}$

	a	b
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow \{0\}$	$\{0\}$	$\{0, 1\}$
$* \{1\}$	$\{2\}$	$\{1\}$
$\{2\}$	$\{2\}$	$\{2\}$
$* \{0,1\}$	$\{0,2\}$	$\{0,1\}$
$\{0,2\}$	$\{0,2\}$	$\{0,1,2\}$
$* \{1,2\}$	$\{2\}$	$\{1,2\}$
$* \{0,1,2\}$	$\{0,2\}$	$\{0,1,2\}$

(2.3.2) Complete subset construction

Figure 2.3.: Complete DFA $\mathcal{D}''_\mathcal{C}$ and its complete subset construction based on the complete NFA $\mathcal{N}_\mathcal{C}$. "$\rightarrow$" denotes initial states, "*" accepting states and $\emptyset$ is the empty set.



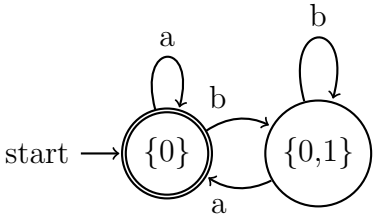
(2.4.1) Complete DFA $\mathcal{D}'_{\mathcal{C}}$

	a	b
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow$ {0}	{0}	{0, 1}
* {1}	$\emptyset$	{1}
* {0,1}	{0}	{0,1}

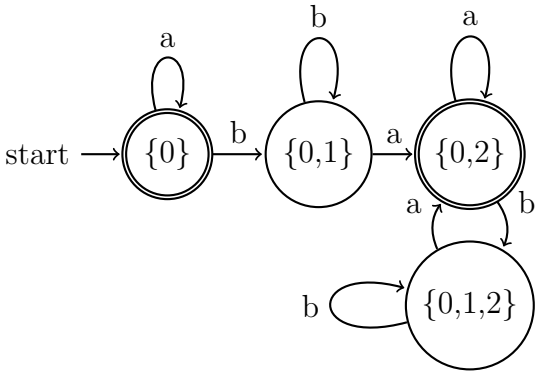
(2.4.2) Complete subset construction

Figure 2.4.: Complete DFA $\mathcal{D}'_{\mathcal{C}}$ and its complete subset construction based on the non-complete NFA $\mathcal{N}$.

In order to obtain the complement DFAs $\overline{\mathcal{D}'_{\mathcal{C}}}$ and $\overline{\mathcal{D}''_{\mathcal{C}}}$ in figure 2.5 of the complete DFAs $\mathcal{D}'_{\mathcal{C}}$ and $\mathcal{D}''_{\mathcal{C}}$, their accepting states have to be complemented [4, p. 135]: p. 135 —> Sec. 4.2.1 Motwani, Hopcroft, Ullman, 3rd Edition



(2.5.1) Complement DFA $\overline{\mathcal{D}'_{\mathcal{C}}}$



(2.5.2) Complement DFA $\overline{\mathcal{D}''_{\mathcal{C}}}$

Figure 2.5.: Complement DFAs $\overline{\mathcal{D}'_{\mathcal{C}}}$ and $\overline{\mathcal{D}''_{\mathcal{C}}}$

As mentioned before, the two obtained complement DFAs differ in size by a factor of two and accept the same language (the empty word and all finite words ending on a). For this small example the difference does not account too much, however, the size problem becomes quickly more severe for bigger NFAs. It is therefore recommended to apply first the subset construction and then complete the DFA if necessary for complementation.

2.3. Büchi automata

A Büchi automaton, named after Swiss mathematician Julius Richard Büchi, can be a deterministic or non-deterministic finite automaton on infinite words, that

recognizes ω-regular languages. Deterministic Büchi automata (DBA) are special cases of non-deterministic Büchi automata (NBA) and form a subset of non-determinstc Büchi automata. Thus the class of languages recognized by NBAs is strictly larger than the class of languages recognized by DBAs. It is therefore not possible to convert a non-deterministic Büchi automaton into a deterministic Büchi automaton . Büchi automata are closed under the boolean operations union, insertion and complementation [2, p. 5] and use the same 5-tuple notation as ordinary automata. Büchi automata accept ω-words only if an accepting state appears infinitely many times in a run sequence ρ. The acceptance condition can therefore be written as $F \subseteq Q$ and $\inf(\rho) \cap F \neq \emptyset$ [7, p. 263].

Büchi himself studied the complementation of his automata in 1960 to solve a decision problem for secondary-order logic [1, p. 6-14] [2, p. 1] [7, p. 261].

We take as example the Büchi automaton $\mathcal{A} = (\Sigma, Q, q_0, \delta, F)$ in figure 2.6 with $\Sigma = \{a, b\}$, $Q = \{0, 1\}$, $q_0 = 0$, $\delta : Q \times \Sigma \to 2^Q$ and $F \subseteq Q$.

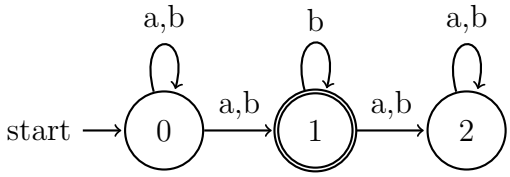


Figure 2.6.: Example NBA $\mathcal{A}$

This Büchi automaton accepts the ω-words having finitely many as such as the ω-word ab^ω for example.

2.4. Büchi complementation - Part 1

The first part of the slice-based Büchi complementation algorithm 2.4.4 consists of determinizing the NBA $\mathcal{A}$ by applying the subset construction algorithm on the initial automaton. We then split *"mixed sets of states"* returned by the subset construction algorithm into non-accepting and accepting sets of states, which requires tuple labels for the states of the complement Büchi automaton and intermediate transition systems. We call a *mixed set of states* a set, that consists of accepting and non-accepting states. Finally we remove reappearing states from sets of states in the tuple labels. The reason for removing reappearing states is, that we want to keep only those states that have first visited an accepting state. Those states can be found in the rightmost branches of the run tree, which is why we remove the states in the tuples from right to left. We now close the first part of the slice-based Büchi complementation by obtaining a labeled transition system. This transition system forms the finite part of the complement Büchi automaton.

We begin the construction of the finite part with the Büchi automaton $\mathcal{A}$ in figure 2.6 and apply in a first step the subset construction algorithm 2.2.1 on the set of states in the tuples of the work list W to find the successor set. The algorithm traverses the sets of states of the tuple in reverse order. This reverse traversal is necessary, because a right-recurring run tree will be accepting [7, p. 263] in the second part of the algorithm 2.5. It should be noticed that we do not apply the complete subset construction algorithm but only the inner "alphabet loop". We then split the sets of states if they are mixed sets of states and prepend them to the sets of states in the successor tuple t'. After having removed reappearing states we end up with a new tuple. Once all tuples are computed we obtain the labeled transition system $\mathcal{T}'$ in figure 2.7.

Algorithm 2.4.1: Subset splitting

input : A mixed set of states m and the set of accepting states F of the initial Büchi automaton

output: A list of sets of states S

$S \leftarrow \emptyset$;
$a \leftarrow \emptyset$; `/* set of states with accepting states */`
$b \leftarrow \emptyset$; `/* set of states with non-accepting states */`
`/* for each state` t `in the mixed set of states` m `*/`
foreach $t \in m$ **do**
  `/* is` t `an accepting state of the original automaton? */`
  if $t \cap F \neq \emptyset$ **then** $a \leftarrow a \cup t$;
  else $b \leftarrow b \cup t$;
end
if $a \neq \emptyset$ ***and*** $b \neq \emptyset$ **then**
  $S \leftarrow b \cup a$; `/* non-accepting come before accepting */`
end
if $a \neq \emptyset$ ***and*** $b = \emptyset$ **then**
  $S \leftarrow a$;
end
if $a = \emptyset$ ***and*** $b \neq \emptyset$ **then**
  $S \leftarrow b$;
end

Mixed sets of states are split by the subset splitting algorithm 2.4.1. The algorithm takes as input a mixed set of states m and the set of accepting states F of the initial Büchi automaton. We then intersect each state t of the set of states m with the accepting set F. Is the result not empty, then we assign the state t to the set of accepting states a, otherwise we assign it to the set of non-accepting states

b. A ordered list of sets of state S is then returned, were the set of non-accepting states b precedes the set of accepting states a.

The removal of reappearing states is done by the state removal algorithm 2.4.2. The algorithm loops through all sets of states of the tuple t in reverse order, because we only want to keep the rightmost states which have visited an accepting state before the states on the left of the tuple. If a state w is already present in the tuple t, we remove it from the sets of states u.

Algorithm 2.4.2: State removal

input : A tuple t and the sets of states Q of the initial Büchi automaton
output: A modified tuple t

```
S ← Q;                                          /* copy the set of states */
/* loop through each set of states u in tuple t                          */
foreach u ∈ t do
    /* loop through each state w in the set of states u                  */
    foreach w ∈ u do
        /* is state w in set of states S?                                */
        if w ∩ S ≠ ∅ then
            S ← S \ w;   /* remove the state from the set of states
            but keep the state in the tuple */
        else
            u ← u \ w;   /* remove the state from the set of states
            of the tuple */
        end
    end
end
```

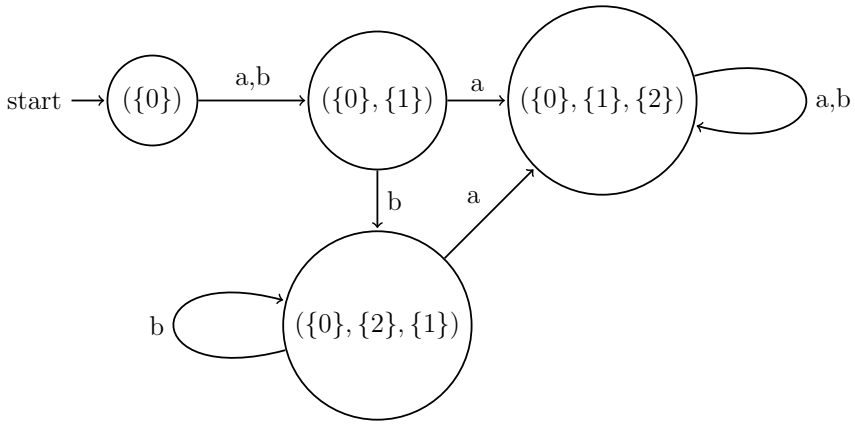


Figure 2.7.: Labeled transition system $\mathcal{T}'$

	a	b
($\emptyset$)	($\emptyset$)	($\emptyset$)
$\rightarrow$ ({0})	($\emptyset$)	({0},{1})
({0},{1})	({0},{1},{2})	({0},{2},{1})
({0},{1},{2})	({0},{1},{2})	({0},{1},{2})
({0},{2},{1})	({0},{1},{2})	({0},{2},{1})

Figure 2.8.: Splitting. Note: the states are tuples of sets of states.

It might be necessary at the end of the construction to complete the labeled transition system $\mathcal{T}'$. In order to check that a transition system is complete, we apply the completeness algorithm 2.4.3. The completeness algorithm creates first an additional state q_{n+1}, that will receive all transitions of states, which do not satisfy $\forall q_i \in Q, \sigma_j \in \Sigma; |\delta(q_i, \sigma_j)| \geq 1$. On any symbol $\sigma_j \in \Sigma$ the state q_{n+1} loops over itself. If a state q_p does not satisfy the previous condition for a given symbol σ_l, then it gets an additional transition $\delta(q_p, \sigma_l) = q_{n+1}$ to the new state q_{n+1}. As soon as all states satisfy the condition a complete automaton is obtained. The labeled transition system $\mathcal{T}'$ in 2.7 is already complete.

Algorithm 2.4.3: Completeness

input : A DFA, NFA or transition system $\mathcal{A}$
output: A complete DFA, NFA or transition system $\mathcal{B}$

$Q_\mathcal{B} \leftarrow Q_\mathcal{A} \cup \{q_{n+1}\}$; `/* add a state for all non-complete states */`
foreach $q_i \in Q_\mathcal{B}$ **do**
  foreach $\sigma_j \in \Sigma$ **do**
    if $\delta(q_i, \sigma_j) = \emptyset$ **then**
      $\delta(q_i, \sigma_j) \leftarrow q_{n+1}$;
    end
  end
end

The construction of the labeled transition system $\mathcal{T}'$ in 2.9 finishes the first part of the slice-based algorithm. The labeled transition system $\mathcal{T}'$ represents the finite part of the later complement Büchi automaton and does not take into account accepting states.

Algorithm 2.4.4: Slice-based Büchi complementation algorithm
input : A Büchi automaton $\mathcal{A}$
output: A transition system $\mathcal{T}'$ with tuple labels

```
W ← ({q_0});          /* work list initialized to initial tuple */
T ← ({q_0}); /* transition system initialized to initial tuple */
while W ≠ ∅ do
    t ∈ W;
    W ← W \ t;                /* select the first tuple t in W */
    foreach σ ∈ Σ do
        t' ← ∅;               /* initialize the successor tuple */
        /* loop through each set of states s_l in reverse order of
           the tuple t                                            */
        for l ← |t|, l > 0, l ← l − 1 do
            L ← ∅;          /* initialize list of set of states */
            u ← subset construction (s_l, σ); /* compute successor set */
            /* is u a mixed set of states?                       */
            if u ∩ F_A ≠ ∅ and u ∩ (Q_A \ F_A) ≠ ∅ then
                L ← subset splitting (u, F_A);
            else
                L ← u;
            end
            L ← color (L, s_l);          /* color the set of states */
            t' ← L ∪ t';                  /* prepend sets of states */
        end
        /* remove reappearing states in the tuple t'             */
        t' ← state removal (t', Q_A);
        if t' ∉ T then
            T ← T ∪ t';
            W ← W ∪ t';
        end
        δ_T' (t, σ) ← t';            /* update the transition table */
    end
end
```

2.5. Büchi complementation - Part 2

For the construction of the infinite part of the complement Büchi automaton we make use of the same algorithms as in 2.4 part 1. In addition, the set of states of the tuples in the infinite part get information assigned. In my thesis I will use

two visual notations to describe the three kinds of information of a set of states. The kind of information a set of states can have is: set of states is an *ordinary* set of states (green color with curly braces "{ }"), set of states is *on hold* (orange color with parenthesis "( )") and set of states representing a *discontinued branch* of the run tree (red color with square brackets "[ ]"). Professor Ulrich Ultes-Nitsche and Joël Allred use a more accurate mathematical tuple notation, where the first element in the tuple describes the set of states and the second element describes the information:

- *ordinary* set of states (finite part): $(\{\text{states}\}, -1)$
- *ordinary* set of states (infinite part): $(\{\text{states}\}, 0)$
- set of states *on hold*: $(\{\text{states}\}, 1)$
- set of states representing *discontinued branch*: $(\{\text{states}\}, 2)$

Again we begin by adding the initial state to the work list of the slice-based Büchi complementation algorithm 2.4.4. New sets of states are generated since each set has now information assigned. The information that gets assigned to a set of states is based on transition rules. To explain the transition rules for the infinite part of the complement Büchi automaton we assume that S is either a mixed set of states or not a mixed set of states, N is the set of non-accepting states with $S \cap (Q \setminus F) = N$, and A is the set of accepting states with $S \cap F = A$. Whenever A or B is the empty set "$\emptyset$" we ignore the set of states for the transition rule.

set of states in predecessor tuple	finite part	infinite part
{ }	{N},{A}	{N},[A]
{ } (without [ ])		{N},[A]
{ } (with [ ])		{N},(A)
( ) (without [ ])		[N],[A]
( ) (with [ ])		(N),(A)
[ ]		[N],[A]

In a first step the subset construction 2.2.1 is again applied followed by the subset splitting algorithm 2.4.1. After the splitting the information is assigned to the sets of states by coloring them according to the transition rules. We obtain the labeled transition system $\mathcal{T}''$ in figure 2.9 representing the infinite part of the complement Büchi automaton. States in $\mathcal{T}''$ have an acceptance condition. Whenever the tuple label of a state in $\mathcal{T}''$ does not have a discontinued set of states it is an accepting state.

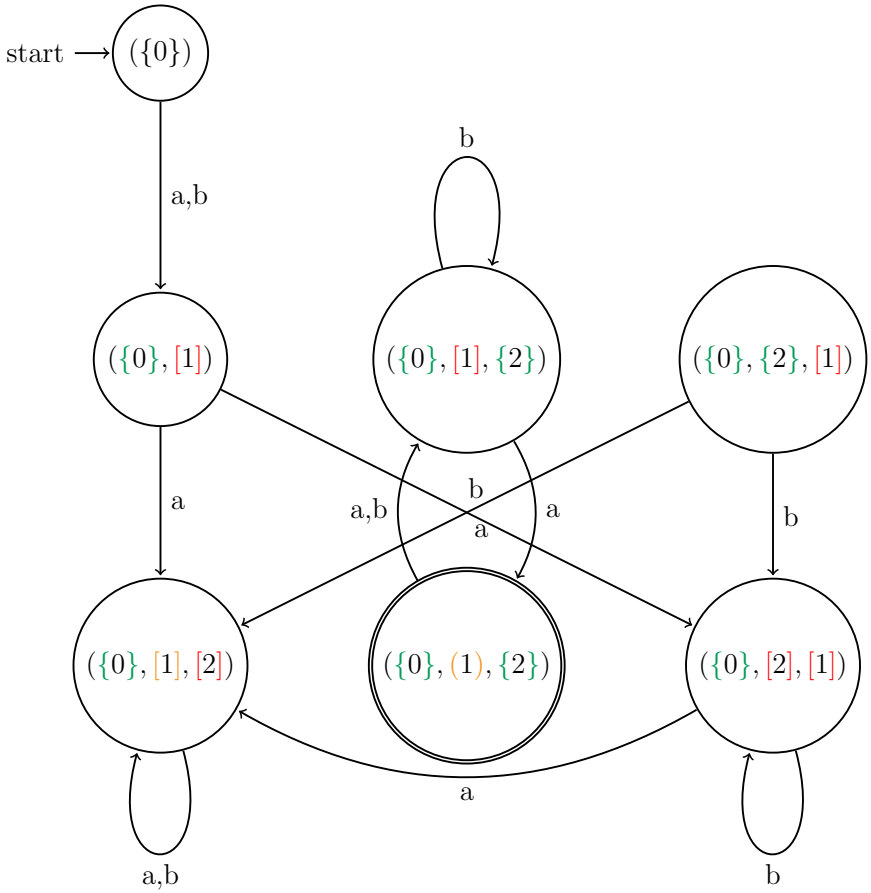


Figure 2.9.: Labeled transition system $\mathcal{T}''$

To obtain the complement Büchi automaton we need to connect $\mathcal{T}'$ and $\mathcal{T}''$. This can be easily done by putting all states of the finite part excluding the initial state on a work list of the slice-based Büchi complementation algorithm 2.4.4. We finally obtain the complement Büchi automaton $\overline{\mathcal{A}}$ in figure 2.11.

The complement Büchi automaton $\overline{\mathcal{A}}$ in figure 2.11 can be further optimized by removing those states, whose tuple labels end with a set of states representing a discontinued branch. This leads to the optimize complement Büchi automaton $\overline{\mathcal{A}_{opt}}$ in figure 2.12.

	a	b
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow$ ({0})	({0},{1}),({0},[1])	({0},{1}),({0},[1])
({0},{1})	({0},{1},{2}),({0},[1],{2})	({0},{2},{1}),({0},{2},[1])
({0},{1},{2})	({0},{1},{2}),({0},[1],{2})	({0},{1},{2}),({0},[1],{2})
({0},{2},{1})	({0},{1},{2}),({0},[1],{2})	({0},{2},{1}),({0},{2},[1])
({0},[1])	({0},(1),[2])	({0},[2],[1])
({0},[1],{2})	({0},(1),{2})	({0},[1],{2})
({0},{2},[1])	({0},(1),[2])	({0},[2],[1])
({0},(1),[2])	({0},(1),[2])	({0},(1),[2])
*({0},(1),{2})	({0},[1],{2})	({0},[1],{2})
({0},[2],[1])	({0},(1),{2})	({0},{2},[1])

Figure 2.10.: Transition table of transition system $\mathcal{T}''$

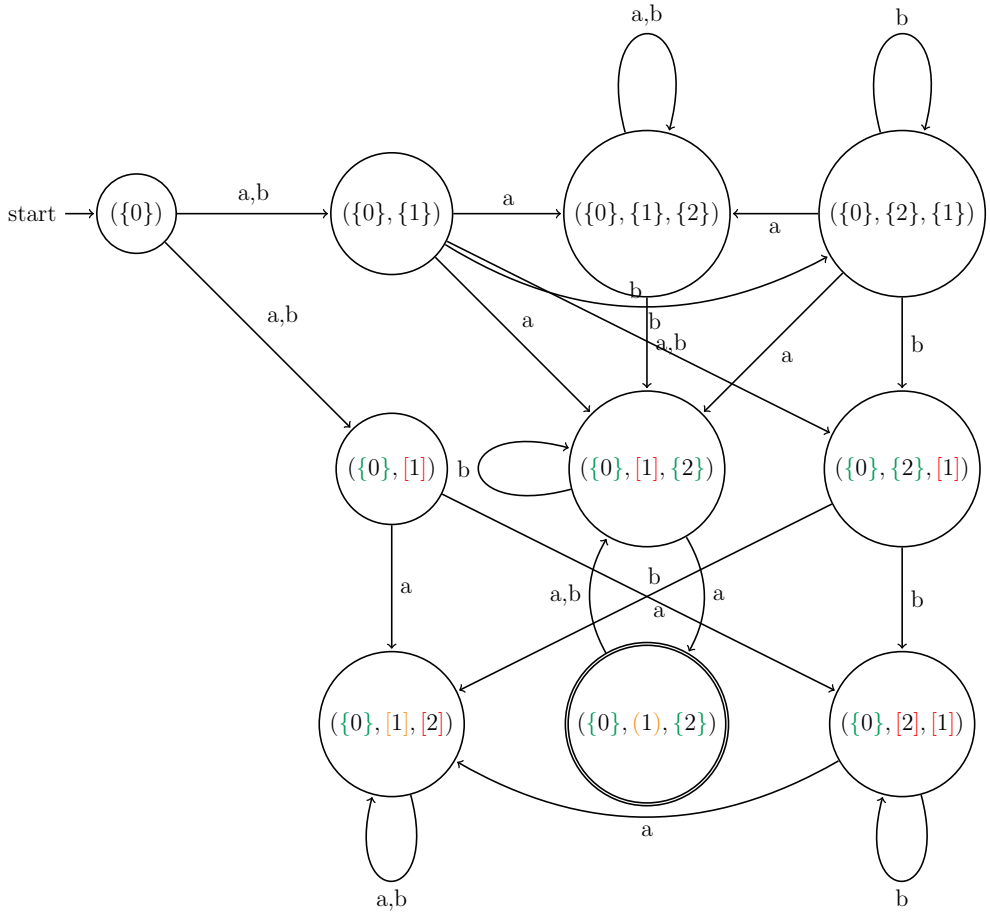


Figure 2.11.: Complement Büchi automaton $\overline{\mathcal{A}}$

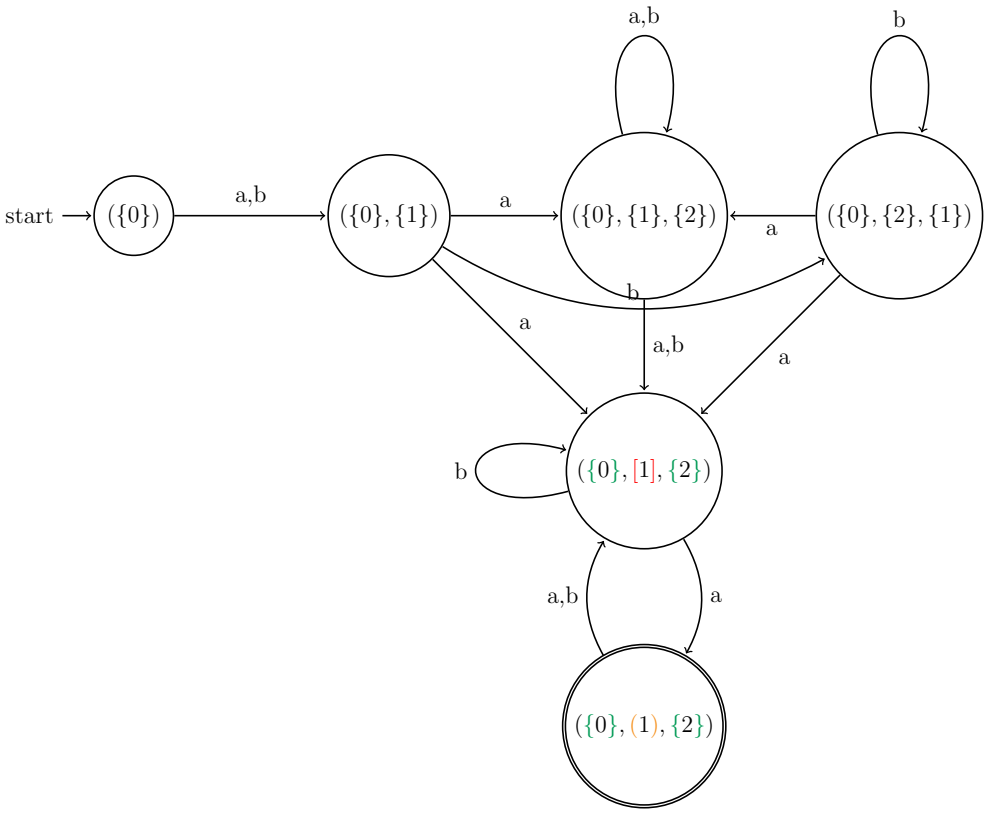


Figure 2.12.: Optimized complement Büchi automaton $\overline{\mathcal{A}_{opt}}$

3. The Implementation

3.1. The EFA library

The EFA library is programmed in C and has a modular design, which allows the user to limit its capabilities at compile time. Overall there are six modules the user can choose of excluding the core module: buechi, famod, fadot, faxml, hash and rbtree. C was selected as programming language for the library because of the complementation blow up of Büchi automata. In order to deal with up to n^n states it is important to have an efficient and fast programming language. C is known as programming language of choice for operating systems, embedded systems, distributed systems and complex programs that have to work with large amounts of data such as databases for example.

The goal of the thesis is to implement the Büchi complementation algorithm developed at the University of Fribourg. libefa became more of a framework for finite automata in general than just for Büchi automata, due to the complexity of the algorithm and the need to deal with states, set of states and tuples during the implementation process. The library therefore provides a lot of functionalities, which are not implemented, because they are not directly related to the goal of the thesis and would have required more time to be implemented.

3.1.1. Modules

core

The core of the EFA library comes with useful functions for memory allocation and deallocation, macros for list operations, error handling, ordinary automata data structures and timing functionality. Users of the library are strongly encouraged to use the memory handling functions provided by the core of the library, as they are wrapper functions of the C standard library functions treated in a safe way. Furthermore the module has an array of all available finite automaton algorithm functions provided by the library and offers one function to parse argument options directly into corresponding data structures for custom programs.

buechi

buechi is the key module of my thesis, as it implements the Büchi complementation algorithms, which is analyzed in chapter 4. The module comes with a declaration for the tuple data structure as it is needed for the slice-based complementation algorithm. It provides functions for Büchi complementation and transformation of a tuple data structure into a state data structure.

famod

The famod module as well as a couple of source file in the core module, come originally from the finite automaton library libfa part of the augeas [1] project from **red**hat. libfa has support for finite automata and regular expressions and comes with its own data structures for automata, mostly based on linked lists. I came to the conclusion that this data structure with linked lists for automata is not well suited for the Büchi complementation, therefore I rewrote the module with only a part of libfas original functionality in order to work with composed data structures and arrays.

fadot

Originally a function in libfa, I have outsourced it to the fadot module. The original function was modified to write automata stored in a composed data structure to a properly formatted graphviz dot file. The module is only capable of writing graphviz dot files. There is no parser implemented that could read a finite automaton into memory from a graphviz dot file.

faxml

The faxml module consists of a parser, that can read into memory a finite automaton from a XML file in FAXML format. The module comes with a XML Schema file that defines the FAXML format 3.1.2. XML files in FAXML format are always validated first before being stored in memory. A debug build of the libefa library automatically dumps the XML Schema and XML file. The module is capable of writing in memory finite automata back to a XML file in FAXML format. faxml needs to be linked against libxml2 as it makes use of its DOM and SAX parsers.

[1] `http://augeas.net`

hash

The hash module was a source file of libfa that was outsourced from the kazlib [2], a generic hash data structure library programmed by Kaz Kylheku. The original source file provided a quite solid implementation of a static as well as a dynamic hash table. However, as the implementation was very generic, it had to be adapted for the composed data structure of a finite automaton in memory. With the efatool 3.3 the user has the possibility to choose among several hash functions at run time. The hash functions that are implemented in the library are public domain or open source and were extracted from Bob Jenkins' home page [3].

rbtree

The rbtree module was added to libefa at the beginning, in order to offer an alternative to the hash table data structure with a red black tree data structure. However, the red black tree data structure is not implemented and should be disabled at compile time.

3.1.2. FAXML format

In order to analyze the complementary Büchi automata generated by the new slice-based Büchi complementation algorithm of libefa, we had to store the automaton in a convenient and readable format. At the time there already existed a XML format proposed by the Vaucason group of the EPITA [4] in Villejuif, France. Unfortunately the XML Schema was too complex for our purpose. For this reason we introduced our own XML Schema FAXML (short for **F**inite **A**utomaton **XML**). The structure of the XML file for a single finite automaton looks as follows in figure 3.1.

The order of the elements is mandatory and imposed by the faxml.xsd schema. Element attributes such as name or label for example are required to store a finite automaton in memory. The schema allows to store multiple finite automata in a single file, by chaining <automaton> elements inside the <faxml> element. I am not going further into details explaining each element and its attributes as the FAXML format is kept simple and is straightforward to write.

[2] http://www.kylheku.com/~kaz/kazlib.html
[3] http://www.burtleburtle.net/bob/hash/doobs.html
[4] http://www.epita.fr

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <faxml version="0.3">
3    <automaton name="example1" class="non-deterministic"
4               minimal="true" type="buechi" >
5      <alphabet>ab</alphabet>
6      <states>
7        <state initial="true" name="0">
8          <transitions>
9            <transition label="ab" to="0" />
10           <transition label="b" to="1" />
11         </transitions>
12       </state>
13       <state accepting="true" name="1">
14         ...
15       </state>
16     </states>
17   </automaton>
18 </faxml>
```

Figure 3.1.: Example structure of finite automaton in FAXML format

3.1.3. Data structures

For a better understanding of the code, the reader should know that in libefa a state_t, sos_t and tuple_t data structure is always abstracted by a hnode_t or a rbnode_t data structure. This abstraction guarantees faster access time for hash table and red black tree data structures.

Finite Automaton

The finite automaton data structure is similar to its original data structure in libfa with a couple of changes. As mentioned earlier, the states are not organized in list structure anymore but in a hash table or red black tree. The data_structure field is a pointer to the data structure that must be cast either to a hash table hash_t or a red black tree rbtree_t.

```
1   typedef struct finite_automaton {
2     void *data_structure;
3     const char *alphabet;
4     const char *name;
5     unsigned int type : 3;
6     unsigned int class : 2;
7     unsigned int minimal : 1;
8     unsigned int trans_re : 1;
9     unsigned int data_type : 3;
10    unsigned int nb_symbols : 16;
11  } fa_t;
```

Figure 3.2.: Declaration of finite automaton data structure

After the alphabet and name field follow bit fields used for further specification of the automaton. The efa.h header file defines specifically enumerations that are assigned to these bit fields. nb_symbols hold the number of different symbols in the alphabet.

State

The state_t data structure in figure 3.3 is the most elementary data structure in libefa. The label holds a unique string to identify a state. It is followed by four bit fields: accept is set if the state is an accepting state, live is set if the state appears in a run sequence, reachable is set if the states is connected to the finite automaton by a transition leading to the state, visited is used by algorithms and set if the state was processed by the algorithm. The structure of transition_row depends on whether the finite automaton is deterministic or non-deterministic. If the finite automaton is deterministic transition_row is an array of pointers to hnode_t or rbnode_t. If it is non-deterministic it is an array of arrays of pointers to hnode_t or rbnode_t. The length of the first array corresponds to the number of symbols nb_symbols.

Set of states

The sos_t (**s**et **o**f **s**tates) data structure in figure 3.4 is very similar to the state_t data structure. The states field is a pointer of an ordered array of pointers to either hnode_t or rbnode_t, which encapsulate state_t for faster access. sos_t has two more bit fields: buechi_info holds the information added to set of states in tuples of the infinite part of a complement Büchi automaton and mixed is set if the set of states consists of accepting and non accepting states.

```
1   typedef struct state {
2     const char *label;
3     unsigned int accept : 1;
4     unsigned int live : 1;
5     unsigned int reachable : 1;
6     unsigned int visited : 1;
7     void **transition_row;
8   } state_t;
```

Figure 3.3.: Declaration of state data structure

```
1   typedef struct set_of_states {
2     const char *label;
3     void **states;
4     unsigned int accept : 1;
5     unsigned int live : 1;
6     unsigned int reachable : 1;
7     unsigned int visited : 1;
8     unsigned int buechi_info : 2;
9     unsigned int mixed : 1;
10    void **transition_row;
11  } sos_t;
```

Figure 3.4.: Declaration of set of states data structure

Tuple

The tuple_t data structure in figure 3.5 is again very similar to the state_t data structure. set_of_states is an array of pointers to hnode_t or rbnode_t encapsulating sos_t. The part bit field is used to indicate the part of a Büchi automaton to which a tuple is assigned (e.g. initial, finite and infinite).

Hash table

The hash table data structure hash_t is used to access and search for states, sets of states and tuples quickly. table holds the entire hash table that has either a power of 2 size or prime size depending on the hash function that is used. nchains stores the current size of the hash table. nodecount holds the current number of nodes and maxcount indicates the maximum number of nodes that can be stored in the hash table.

highmark and lowmark indicate the lower and upper bound of nodes in a hash

```
1   typedef struct tuple {
2     const char *label;
3     void **set_of_states;
4     unsigned int accept : 1;
5     unsigned int live : 1;
6     unsigned int reachable : 1;
7     unsigned int visited : 1;
8     unsigned int part : 2;
9     void **transition_row;
10  } tuple_t;
```

Figure 3.5.: Declaration of tuple data structure

```
1   typedef struct hash_t {
2     hnode_t **table;
3     hnode_t *initial;
4     uint32_t nchains;
5     uint32_t nodecount;
6     uint32_t maxcount;
7     uint32_t highmark;
8     uint32_t lowmark;
9     uint32_t mask;
10    hash_comp_t compare;
11    hash_fun_t function;
12    unsigned int dynamic : 1;
13  } hash_t;
```

Figure 3.6.: Declaration of 32-bit hash table data structure

table if the dynamic bit field is set. The hash table size will increase or decrease if the lower or upper bound is reached. The lowmark and highmark fields are ignored if the dynamic bit field not set. mask is a bit mask field that masks the bits of a hash according to the size of the hash table. compare and function are function pointers that point to a hash compare function and a hash function, which can both be exchanged at run time.

Hash nodes are used as carrier data structures to store a state, set of states or tuples in the hash table. The next field is a pointer to the next hash node in the hash table chain. The length of such a hash chain depends on how good your hash function is. If it is a good hash function, it will distribute the items uniformly in the hash table, meaning that all chains will have about the same length and the load factor of the table is high. If your buggy or not well elaborated and all

```
1     typedef struct hash_node {
2       struct hash_node *next;
3       const char *key;
4       uint32_t hkey;
5       void *data;
6     } hnode_t;
```

Figure 3.7.: Declaration of hash node data structure

items get mapped to the same hash, you will end up with a hash chain holding all item. Furthermore hnode_t stores the key and also the hashed key hkey for quick access. The data field is then a pointer to the actual state, set of states or tuple data structure that has to be casted.

The complexity analysis shows a slight advantage for the hash table with a possible best case time complexity of $O(1)$ over $O(lg(n))$ for the red black tree, where n is the number of nodes stored in the data structure. Supposed is we have a good hash function, that distributes all items uniformly. Then we can access an item in the hash table in constant time $O(1)$, which is considered the best case. Provided we still use a good hash function and we are dealing with a hash table that has a node count close to the high mark, we know that all chains in the hash table hold about the same number of items. Since the high mark is twice the number of chains in the hash table, we conclude that the time needed to access an item in the hash table will take at most $O(lg(n))$ time. Since this is mostly the use case, we call this the average case. In the worst case as described before, all states are mapped to the same hash and stored in a single chain. The time it takes to access the last item in this chain is of the order $O(n)$, as each item in the chain has to be traversed.

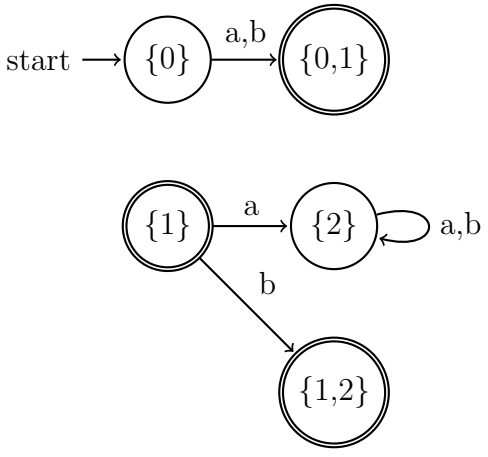
3.1.4. Implementation of the algorithm

The slice-based Büchi complementation algorithm has been implemented as described in sections 2.4 and 2.5. However, I made some adjustments to the algorithm by adding a modified subset construction algorithm 3.1.1, that constructs a temporary labeled transition system to speed up the complementation process. This additional algorithm creates for the finite part the temporary, disconnected deterministic labeled transition system $\mathcal{T}'''$ in figure 3.8, which becomes the disconnected non-deterministic labeled transition system $\mathcal{T}^{\mathcal{IV}}$ in figure 3.9 of the infinite part. By convention, I colored the mixed sets of states in the labeled transition system $\mathcal{T}^{\mathcal{IV}}$ according to the color the set of accepting states would get, based on these transition rules.

Algorithm 3.1.1: Modified subset construction

input : A Büchi automaton $\mathcal{B}$
output: A labeled transition system $\mathcal{T}$

```
/* add all states as sets of states to the transition system
   and work list                                              */
Q_T ← {q_i}, ∀q_i ∈ Q_B;
W ← {q_i}, ∀q_i ∈ Q_B;
while W ≠ ∅ do
    s ∈ W;                                  /* set of states s from W */
    W ← W \ s;                          /* mark s by removing it from W */
    /* loop through all symbols σ in the alphabet Σ                   */
    foreach σ ∈ Σ do
        t ← δ(s, σ);                /* find the successor set of states */
        if t ∉ T then
            Q_T ← Q_T ∪ t;                  /* add new set of states t */
            /* Add only non-mixed sets of states to the work
               list.  p is a state in the set of states t.            */
            if ∀p ∈ t, p ∉ F_B then
                W ← W ∪ t;
            else if ∀p ∈ t, p ∈ F_B then
                W ← W ∪ t;
            end
        end
        δ_T(s, σ) ← t;                  /* update the transition table */
    end
end
```



(3.8.1) Labeled transition system $\mathcal{T}'''$

	a	b
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow \{0\}$	$\{0,1\}$	$\{0,1\}$
$*\{1\}$	$\{2\}$	$\{1,2\}$
$\{2\}$	$\{2\}$	$\{2\}$
$*\{0,1\}$	$\emptyset$	$\emptyset$
$*\{1,2\}$	$\emptyset$	$\emptyset$

(3.8.2) Modified subset construction

Figure 3.8.: Labeled transition system $\mathcal{T}'''$ and its modified subset construction based on the NBA $\mathcal{A}$

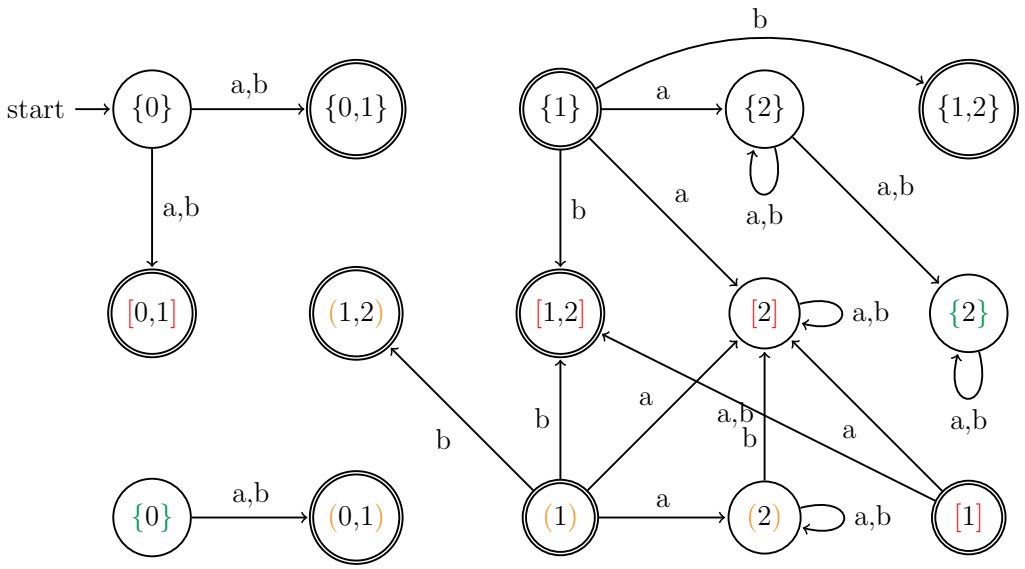


Figure 3.9.: Labeled transition system $\mathcal{T}^{\mathcal{IV}}$

	a	b
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow \{0\}$	$\{0,1\},[0,1]$	$\{0,1\},[0,1]$
$*\{1\}$	$\{2\},[2]$	$\{1,2\},[1,2]$
$\{2\}$	$\{2\},\{2\}$	$\{2\},\{2\}$
$*\{0,1\}$	$\emptyset$	$\emptyset$
$*\{1,2\}$	$\emptyset$	$\emptyset$
$\{0\}$	$(0,1)$	$(0,1)$
$*(1)$	$(2),[2]$	$(1,2),[1,2]$
$*[1]$	$[2]$	$[1,2]$
$\{2\}$	$\{2\}$	$\{2\}$
$\{2\}$	$\{2\}$	$\{2\}$
$\{2\}$	$\{2\}$	$\{2\}$
$*[0,1]$	$\emptyset$	$\emptyset$
$*[0,1]$	$\emptyset$	$\emptyset$
$*(1,2)$	$\emptyset$	$\emptyset$
$*[1,2]$	$\emptyset$	$\emptyset$

Figure 3.10.: Transition table for labeled transition system $\mathcal{T}^{\mathcal{IV}}$

3.2. Installing libefa

libefa was programmed on a Linux operating system and only supports the Linux operating system. It requires the following packages to be installed:

- gcc
- glibc, glibc-devel
- autoconf, automake, m4, make, libtool
- libxml2, libxml2-devel.
- pdflatex (texlive-latex)

The libefa packages comes already bootstrapped. If changes are made to Makefile.am or the configure.ac file, one has to bootstrap the packages with the GNU autotools chain by executing the commands in 3.11.

```
user> aclocal --force --warnings=all
user> libtoolize --force
user> aclocal --force --warnings=all
user> autoconf --force --warnings=all
user> autoheader --force --warnings=all
user> automake --force --warnings=all --add-missing
```

Figure 3.11.: Executing the GNU autotools chain to bootstrap the package

To install the package one simply executes the usual Unix build commands as in 3.12.

libefa will be automatically installed in /usr/local/lib and efatool in /usr/local/bin. Furthermore libefa comes with a couple of example files and the XML Schema for FAXML that can be found in /usr/local/share/libefa.

3.3. The efatool

The efatool is the interface between the user and libefa. It takes one or multiple finite automata stored in a single file or a regular expression and manipulates each finite automaton according to the additional arguments given to the program. efatool can be used to convert file formats. This command for example converts a finite automaton stored in FAXML format to graphviz' dot format. By further processing the dot file an image of the finite automaton can be generated.

```
(distribution build)
user> ./configure --disable-rbtree LDFALGS=-lxml2
      CFLAGS=-O2

(debug build)
user> ./configure --disable-rbtree LDFLAGS=-lxml2
      CFLAGS="-g -O0" CPPFLAGS=-DDEBUG

user> make clean
user> make
root> make install
```

Figure 3.12.: Installing libefa

```
user> efatool -i in.xml -o out.dot
```

Figure 3.13.: Command to convert finite automaton formats

Manipulation of finite automata is done with the algorithm chain. The algorithms are linked together and executed in the order given by the algorithm argument. If a user wants to do a subset construction on a NFA and compare the produced DFA to another automaton, one would write the following command:

```
user> efatool -i in.xml -a transform.subset_construction,eq
uivalence.table-filling=comp.xml
```

Figure 3.14.: Command to transform and compare finite automata

The algorithm argument structure is broken down into three levels: category, algorithm name and parameters. First comes the category of algorithm a user wants to apply. A user can obtain a list of all algorithms and their category with the argument -halgorithm or –help=algorithm. Followed by the category comes a dot sign and the algorithm name of the algorithm in question. Users can pass parameters to an algorithm only if the algorithm itself can be parametrized. Parameters are separated by a equal sign from the algorithm name followed by the list of parameters. The list of parameters is a key value pair separated by an equal sign too, whereas the elements of the list are separated by a colon sign.

The user can pass a finite automaton to the efatool by a regular expression. In addition the finite automaton can be further specified with arguments for class and type of automaton. Furthermore the user is able to choose between hash table

and red black tree in memory representation of the finite automaton. Putting it all together would be a command such as this one:

```
user> efatool -r ab* -c deterministic -t ordinary -d hashta
ble -H kazlib -o out.xml
```

Figure 3.15.: Exaple command with regular expression

Last but not least benchmarking. Each algorithm in libefa is timed and could also be benchmarked for memory usage, but its implementation is not done and was of lower priority for this thesis. A user can benchmark IO and transformation of a finite automaton with a command such as:

```
user> efatool -i in.xml -a complementation.unifr -v time -o
out.xml
```

Figure 3.16.: Command with benchmarking

4. Performance analysis of the algorithm

4.1. Environment specifications

All Büchi automata used to test the slice-based Büchi complementation algorithm were tested on two machines: one 32-bit machine and one 64-bit machine. I tried to set up a similar environment on both machines, in order to compare their results. All tests were run in text mode (# init 3) to have as much as possible memory available for the tests and as few as possible other processes running at the same time. libefa is single threaded and will thus be executed only on one core of the 64-bit machine.

	32-bit machine	64-bit machine
Processor	Intel® Pentium® 4 3.00 GHz	Intel® Core™2 Duo 2.66 GHz
Memory	1.97 GiB	3.71 GiB
Operating System	openSUSE Tumbleweed (openSUSE 12.3 i586)	openSUSE Tumbleweed (openSUSE 12.3 x86_64)
Kernel	Linux 3.11.5-32	Linux 3.11.5-32
Swap	2.01 GiB	2.00 GiB
C compiler	gcc 4.7.2	gcc 4.7.2
C standard library	glibc 2.17-4.4.1	glibc 2.17-4.4.1
XML library	libxml2 2.9.0-2.17.1	libxml2 2.9.0-2.17.1
Compiler flags	-O2	-O2

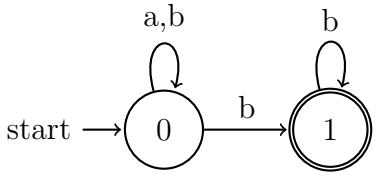
4.2. Büchi automata used for testing

This section provides a listing of all Büchi automata used for testing as well as a short explanation. Not all of their complementary Büchi automata are depicted in the appendix as they are too large to fit properly on a page.

4.2.1. Example Büchi automata

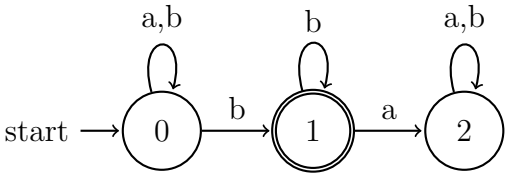
Example 1

Example 1 is a non-deterministic Büchi automaton that accepts all ω-words having a finite number of as. This automaton was chosen since it can be quickly complemented by hand. It was the first NBA used to check the correctness of the slice-based Büchi complementation algorithm 2.4.4. The complementary Büchi automaton is depicted in figure A.1.



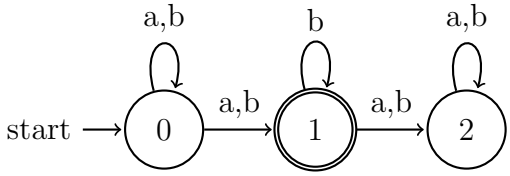
Example 2

Example 2 is the complete version of NBA example 1 4.2.1. It allows me to compare this complementary Büchi automaton A.2 to the complementary Büchi automaton obtained in example 1. Another interesting aspect of the complementary Büchi automaton of this example is the tuple construction with subset splitting of the infinite part. The whole infinite part of the complementary Büchi automaton is split into two branches that can only be accessed via the finite part. The tuple construction with subset splitting starting with the initial state ($\{0\}$) constructs one of this branches while the other branch is constructed when the finite and infinite part are connected.



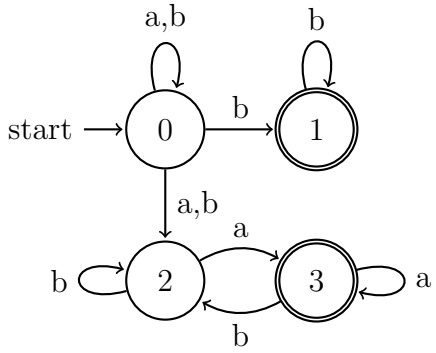
Example 3

The NBA of example 3 is the Büchi automaton Prof. Dr. Ulrich Ultes-Nitsche and Joël Allred used to test their slice-based complementation algorithm. It basically accepts the same language as the NBA in example 2 4.2.1. Again the infinite part of the complementary Büchi automaton A.3 is split into two branches that are accessed via the finite part as already seen in example 4.2.1.



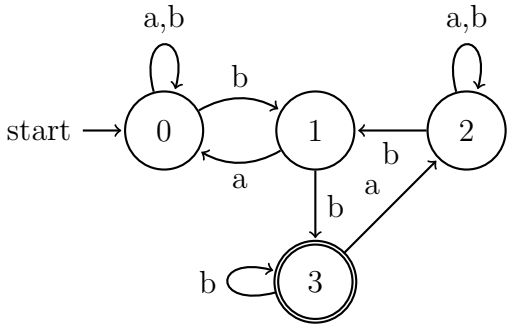
Example 4

This NBA is the first Büchi automaton in this example series that has multiple accepting states. Three kinds of ω-words are accepted: ω-words having a finite number of as, a finite number of bs or ω-words having a sequence of $(ba)^\omega$. Furthermore, the complement Büchi automaton is the first complement Büchi automaton having in the label split set of states with more than just one state.



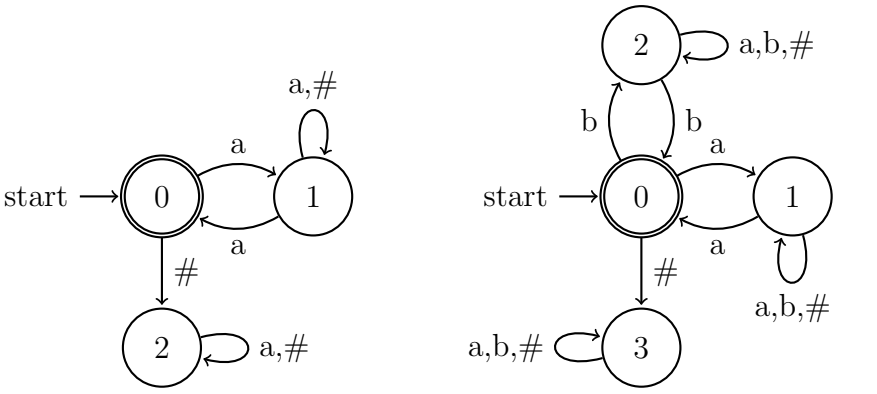
Example 5

Example 5 is a NBA accepting ω-words with a sequence of $(bb)^\omega$. The complement Büchi automaton of this example combines many of the features of the previous examples.



4.2.2. Michel's state explosion Büchi automaton

Michel used the Büchi automata in figure 4.1 1988 to prove the explosion of states when complementing Büchi automata [10, p. 423]. These automata, when complemented with our slice-based complementation algorithm, generate permutations of the sets of states in the tuple label. From combinatorics we know that the number of permutations of a list with k elements is $k!$. $n!$ is therefore the number of states generated from a tuple of size n. As the tuples of a complement Büchi automaton are of different size, we can conclude that the number of states obtained after complementation of a "Michel automaton" is $> n!$.



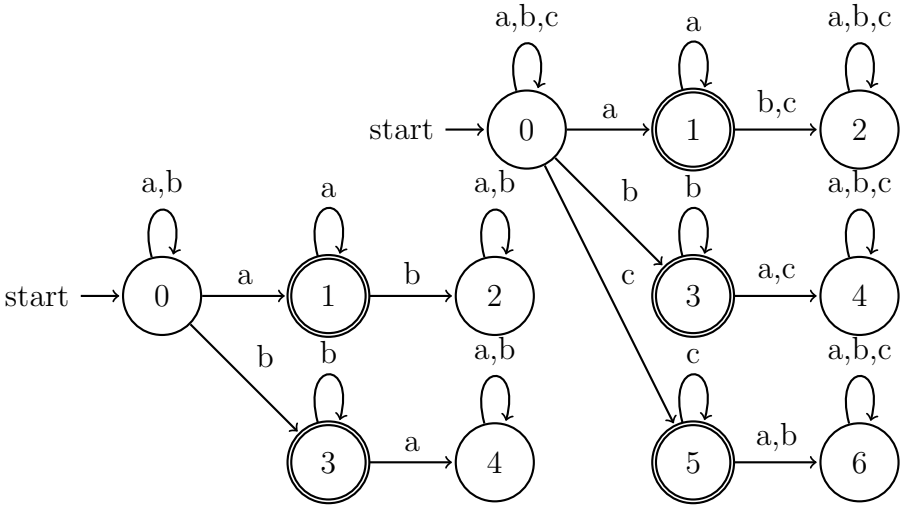
(4.1.1) Michel automaton with $n = 1$

(4.1.2) Michel automaton with $n = 2$

Figure 4.1.: Büchi automata used by Michel to prove state explosion of Büchi complementation

4.2.3. "Symbol" Büchi automaton

This kind of Büchi automaton accepts an ω-word if it ends on a infinite sequence of one symbol of the alphabet, which is why I gave it the name "symbol automaton". It is basically a combination of multiple Büchi automata as in example 4.2.1, but each accepting another symbol.



(4.2.1) Symbol automaton of size $n = 2$

(4.2.2) Symbol automaton of size $n = 3$

4.2.4. Test set

The paper "*State of Büchi Complementation*" by Vardi et all. [7] has a reference to the URL [1] from which two sets of random Büchi automata can be downloaded. One set has 11'000 Büchi automata of size 15 and the other 11'000 Büchi automata of size 20. The 11'000 Büchi automata are categorized into different classes: there are 11 transition density classes and 10 acceptance density classes. The transition density ranges from 1 to 3 and indicates the average number of transitions from a state per symbol. The acceptance density is given in percent of states of the initial Büchi automaton. Each class holds 100 random Büchi automata. Since these Büchi automata were randomly generated, they have unreachable states.

4.3. Test summary

4.3.1. Example Büchi automata

The following table shows some values obtained from applying the slice-based Büchi complementation algorithm on the example Büchi automata.

[1] `http://goal.im.ntu.edu.tw`

	init. size	c. size	c. trans.	states fin.	states infi.	acc. states
example1	2	4	12	2	2	1
example2	3	10	28	4	6	2
example3	3	11	30	4	7	1
example4	4	12	34	5	7	2
example5	4	27	72	9	18	6
michel1	3	53	128	11	42	7
michel2	4	814	2'610	56	758	99
michel3	5	14'473	44'836	305	14'168	1'453
michel4	6	289'448	1'456'670	1'886	287'562	21'925
symbol2	5	39	96	9	30	4
symbol3	7	307	1'059	46	261	36
symbol4	9	2'253	10'040	257	1'996	240
symbol5	11	17'571	95'985	1'626	15'945	1'600

The timing for complementing the example Büchi automata can be found in the following section 4.3.2 *Hash functions*.

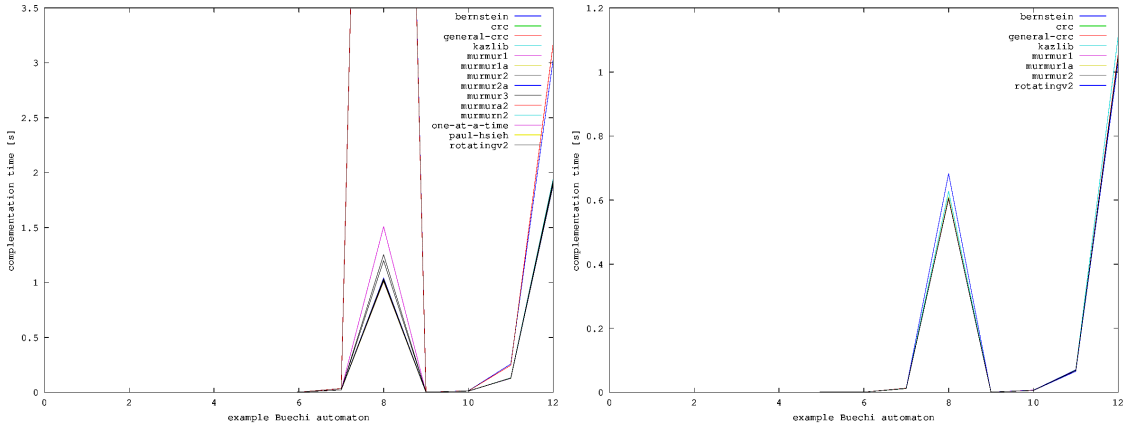
4.3.2. Hash functions

In this section I want to find out which of the implemented hash functions performs best for this slice-based Büchi complementation algorithm. I therefore complemented each of the example Büchi automata five times and computed the average complementation time. The following table describes the numbering of the hash functions and the example Büchi automata of figure 4.2.

32-bit	axis number	64-bit	axis number
bernstein	1	bernstein	1
crc	2	crc	2
general-crc	3	general-crc	3
kazlib	4	kazlib	4
murmur1	5	murmur1	5
murmur1a	6	murmur1a	6
murmur2	7	murmur2	7
murmur2a	8	rotatingv2	8
murmur3	9		
murmura2	10		
murmurn2	11		
one-at-a-time	12		
paul-hsieh	13		
rotatingv2	14		

Büchi automaton	axis number
example1	1
example2	2
example3	3
example4	4
example5	5
michel1	6
michel2	7
michel3	8
symbol2	9
symbol3	10
symbol4	11
symbol5	12

The graphs in figure 4.2 show, that the time needed to complement one of the example Büchi automata does not change much.



(4.2.3) 32-bit time (4.2.4) 64-bit time

Figure 4.2.: Recorded times for 32-bit and 64-bit machines

The following ranking has been made from the time measurement of the *symbol5* automaton, since it is the largest complement Büchi automaton in the example Büchi automaton set. Hash collisions are not taken into account for this ranking.

Rank	32-bit hash function	Rank	64-bit hash function
1.	bernstein	1.	bernstein
2.	paul-hsieh	2.	crc
3.	murmur3	3.	murmur2
4.	crc	4.	murmur1a
5.	murmur2	5.	rotatingv2
6.	murmur1	6.	murmur1
7.	rotatingv2	7.	general-crc
8.	one-at-a-time	8.	kazlib
9.	murmur1a		
10.	general-crc		
11.	murmurn2		
12.	kazlib		
13.	murmur2a		
14.	murmura2		

One can see that the time needed for complementing the Büchi automata on the 64-bit machine is about half as long as on the 32-bit machine. Based on the ranking I recommend using *Bernstein* hash to complement Büchi automata with the slice-based Büchi complementation algorithm.

4.3.3. Test set

I applied the slice-based Büchi complementation algorithm to the test set in order to compare it with the results obtained by Vardi et all. in their paper "*State of Büchi complementation*" [7]. I complemented the test set with three different versions of the slice-based Büchi complementation algorithm:

0: complement the Büchi automaton

1: complete the Büchi automaton before complementing it

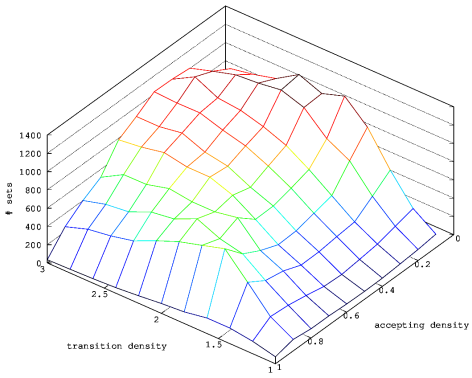
2: complete the Büchi automaton, complement it and apply the optimization

Unfortunately it will not be possible to directly compare the results of the new slice-based complementation algorithm to the results in "*State of Büchi complementation* because I do not know the effective samples.

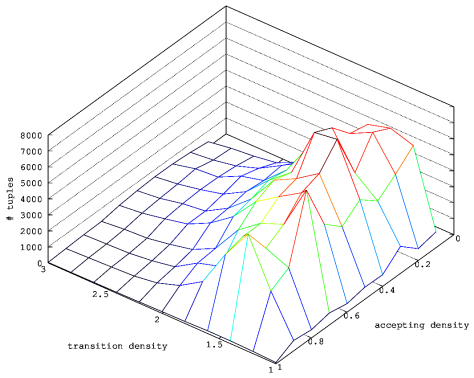
Size / Memory

The following plots show the number of sets of state that are generated while complementing the Büchi automata of the test set and the complement size of the Büchi automata. The number in the caption indicates which version of the slice-based Büchi complementation algorithm was applied.

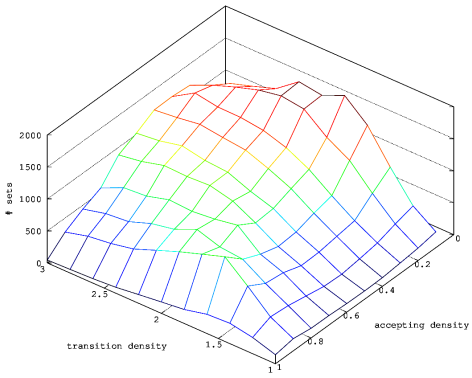
Initial Büchi automaton size 15:



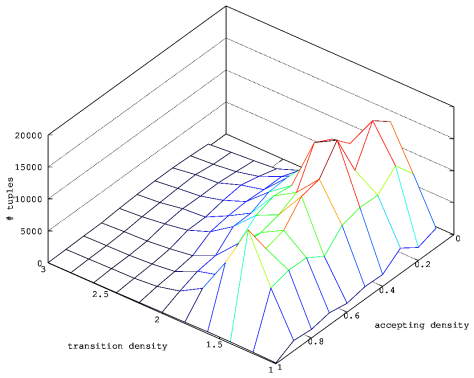
(4.3.1) 0: Sets of states



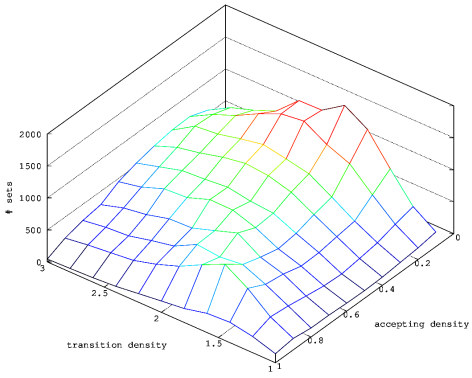
(4.3.2) 0: Tuples



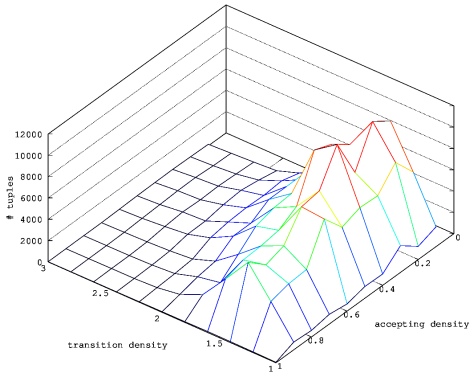
(4.3.3) 1: Sets of states



(4.3.4) 1: Tuples

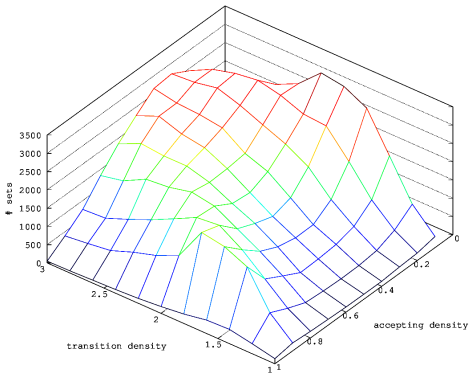


(4.3.5) 2: Sets of states

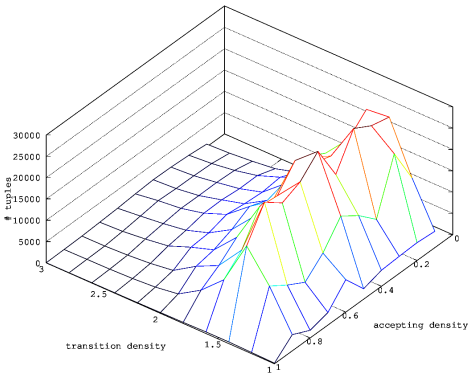


(4.3.6) 2: Tuples

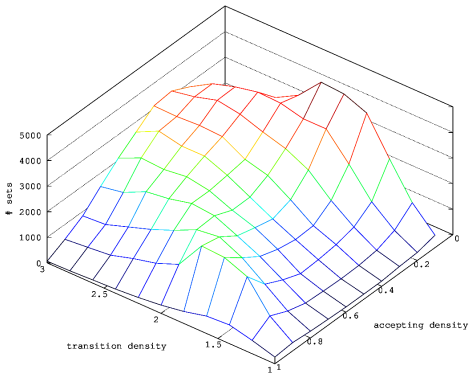
Initial Büchi automaton size 20:



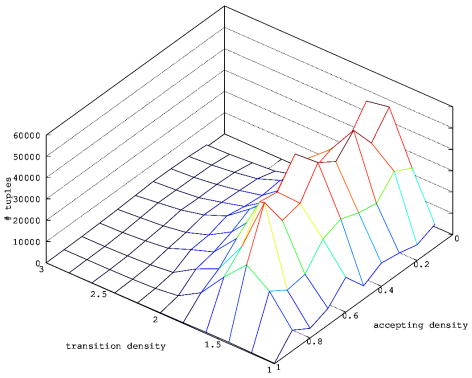
(4.3.7) 0: Sets of states



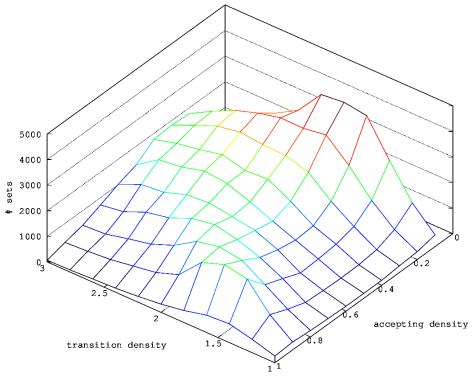
(4.3.8) 0: Tuples constructed



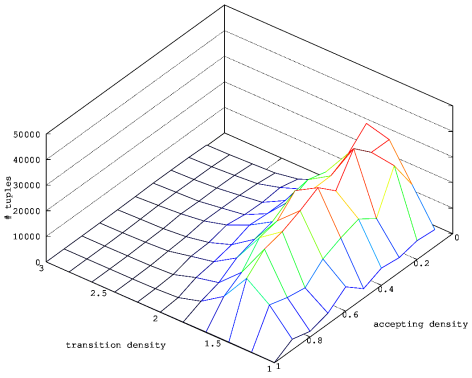
(4.3.9) 1: Sets of states



(4.3.10) 1: Tuples



(4.3.11) 2: Sets of states



(4.3.12) 2: Tuples

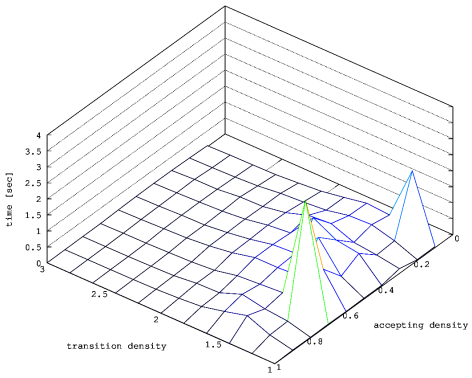
We can see, that there are always around 1'400+ set of states generated for the size 15 test set and around 3'000+ set of states for the size 20 test set. The reason this number does not change much is due to the implementation of the algorithm. The number of sets of states is higher for version 1 of the algorithm than for version 0, which is because of the additional state to make the Büchi automata complete. Version 2 of the algorithm generates less sets of states, since the algorithm removes the states for the optimization already in the construction process. We therefore do not need to generate as many sets of states as we do for version 0 and 1 of the algorithm.

For all versions of the algorithm and initial sizes of the test sets we see an interesting evolution of the complement size (tuples) of the Büchi automata. The algorithm seems to have some characteristic that lets the complement size of the Büchi automaton explode if the initial Büchi automaton has a transition density of around 1.5. If we look at the initial Büchi automata, we can see that the states in Büchi automata with a transition density below 1.5 often have a deterministic choice for a transition of a given symbol. Around a transition density of 1.5 the states of the initial Büchi automata have almost always a non-deterministic choice to transition with a given symbol. On average at a given state we can reach at least another state with any symbol. As the transition density further increases the complement size decreases. In this situation we have on average more than one state that is reachable from any state in the initial Büchi automaton with any symbol. The determinization step in the algorithm can then group these states together and thus reduces the complement size.

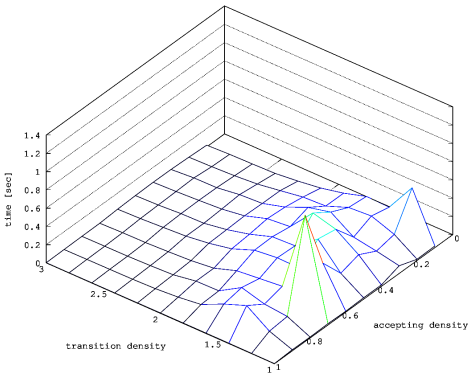
Time

The following plots show the time needed to complement the test sets with all versions of the slice-based Büchi complementation algorithm.

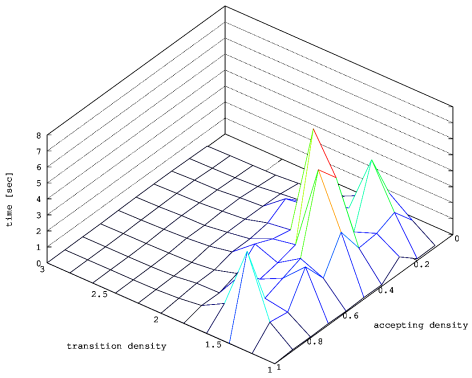
Initial Büchi automaton size 15:



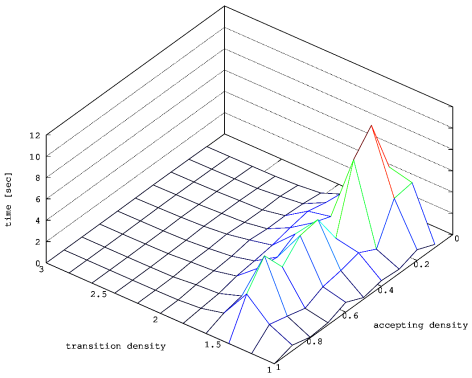
(4.3.13) 0: 32-bit time



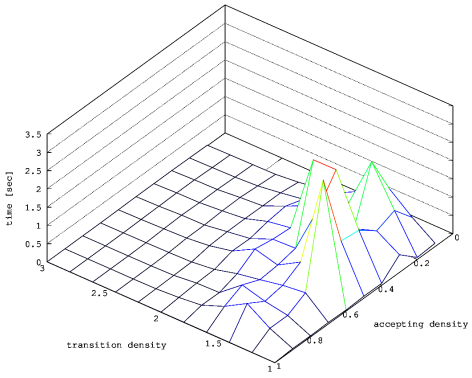
(4.3.14) 0: 64-bit time



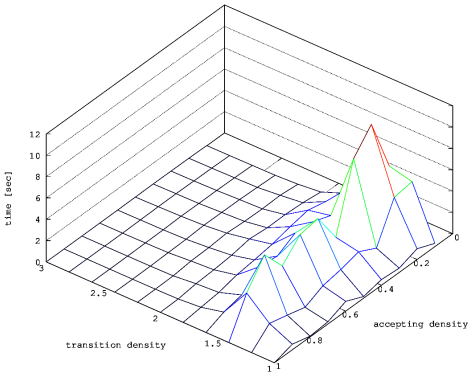
(4.3.15) 1: 32-bit time



(4.3.16) 1: 64-bit time

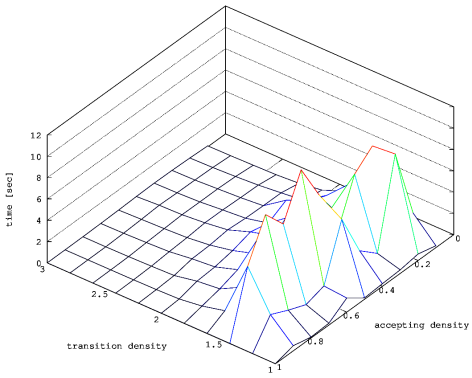


(4.3.17) 2: 32-bit time

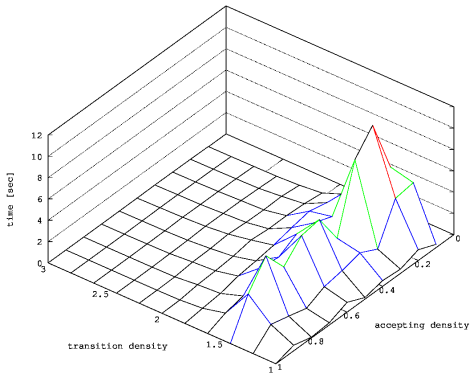


(4.3.18) 2: 64-bit time

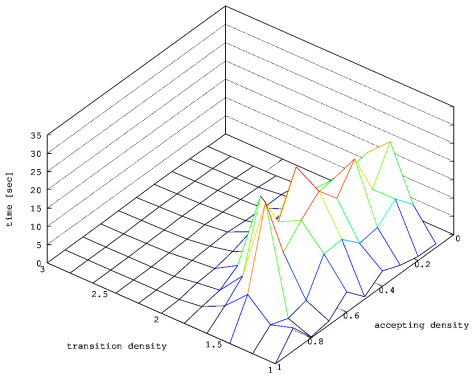
Initial Büchi automaton size 20:



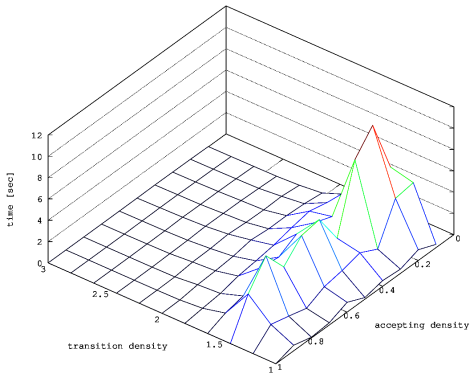
(4.3.19) 0: 32-bit time



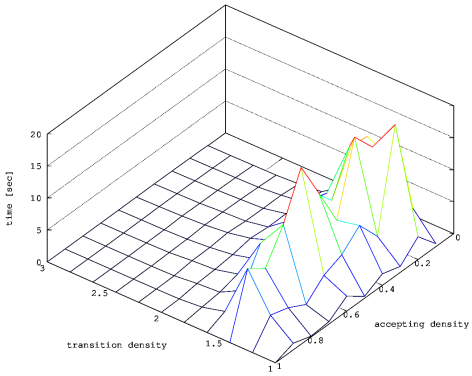
(4.3.20) 0: 64-bit time



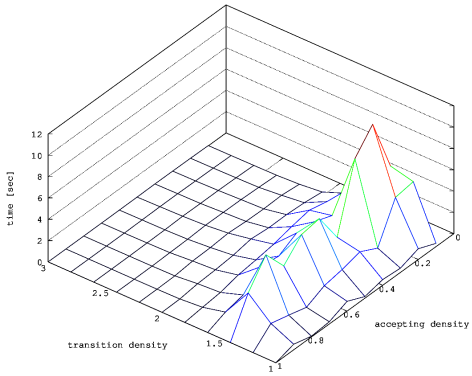
(4.3.21) 1: 32-bit time



(4.3.22) 1: 64-bit time



(4.3.23) 2: 32-bit time



(4.3.24) 2: 64-bit time

The results show that there are time peaks around a transition density of 1.5, which is due to the large number of states that is generated for the complement size. However, complementation times of smaller Büchi automata take longer on the 64-bit machine than on the 32-bit machine as can be seen for versions 1 and 2 of the algorithm for initial Büchi automata of size 15. Probably this is due to the architecture of the CPUs. The 32-bit CPU has a higher frequency than the 64-bit CPU, but the 64-bit CPU has a larger cache.

4.3.4. Complexity

To see the evolution of the complement size of Michel's Büchi automata I plotted their initial size against their complement size and fitted the points in the graph with `gnuplot` using a function of the form $(an)^n$.

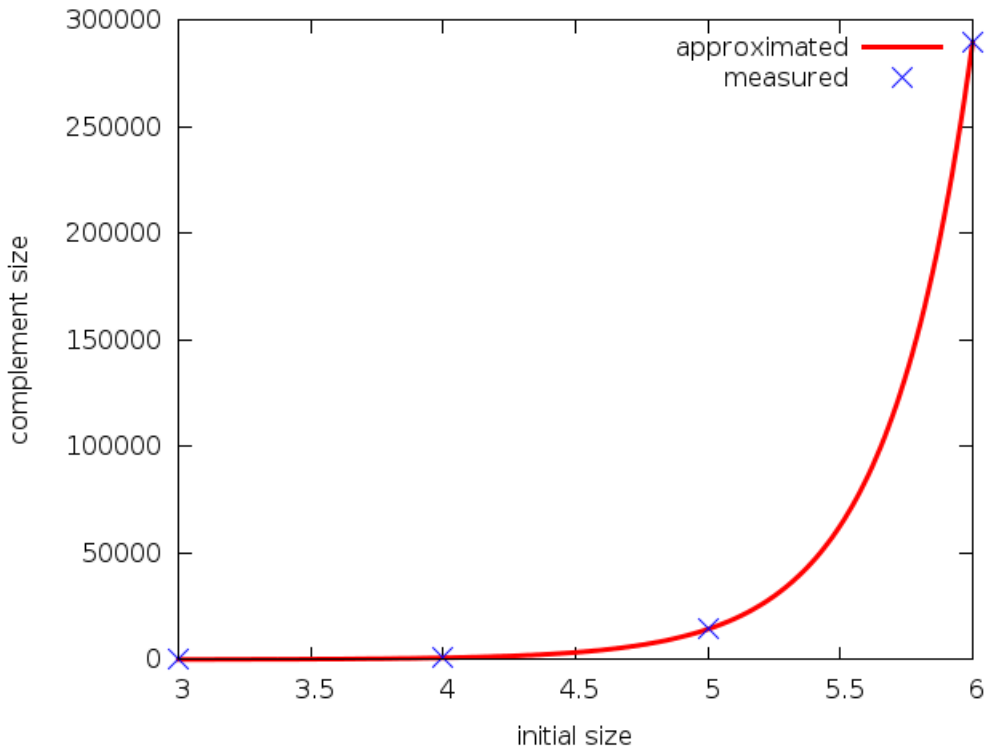


Figure 4.3.: State explosion of Michel's automata 1-4 fitted by $(an)^n$ function

I found that the parameter a has a value of 1.36 with an asymptotic standard error of 0.01% leading to the function $c(n) = (1.36n)^n$ where $c(n)$ is the complement size of Michel's automaton with n initial states.

5. Conclusion

Slice-based complementation algorithms tend to generate a lot more reachable states than Safra-Piterman-based algorithms or rank-based algorithms. I made the same experience for the new slice-based complementation algorithm as can be seen in the test summary 4.3. However, applying the new slice-based complementation algorithm directly on a Büchi automaton without completing the Büchi automaton first produces less states than the new slice-based complementation algorithm with optimization. Thus further optimization need to be made for the new slice-based complementation algorithm in order to reduce the complement size of Büchi automata.

My thesis demonstrated that the slice-based Büchi complementation algorithm developed by Prof. Dr. Ulrich Ultes-Nitsche and Joël Allred can be implemented and seems to perform well compared to other Büchi complementation algorithms. From the objective point of view it is easier to apply the new slice-based algorithm than the slice-based algorithm described by the authors of *"State of Büchi Complementation (Full Version)"* [8, p. 12]. The approach of using tuples with colored sets of states as label for the states contributes to the simplification of the slice-based algorithm.

At this point `libefa` is a small C library for finite automata that has not a lot of functionality. It would therefore be good if the library could be further improved, while adding new optimization to the new slice-based complementation algorithm. There are many places in the library that need to be implemented such as the red-black-tree module, basic algorithms for finite automata as well as for Büchi automata and monitoring memory usage. Porting the library to other platforms and adding a user interface for the `efatool` would round off the project.

Bibliography

[1] J. Richard Büchi. Weak second-order arithmetic and finite automata. Technical report, University of Michigan, September 1959.

[2] J. Richard Büchi. On a decision method in restricted second order arithmetic. In *Logic, Methodology and Philosophy of Science (Proceedings of the 1960 International Congress)*, pages 1–11. Stanford University Press, Stanford, California, 1962.

[3] Ehud Friedgut, Orna Kupferman, and Moshe Y. Vardi. Büchi complementation made tighter. *International Journal of Foundations of Computer Science*, 2004.

[4] Rajeev Motwani John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Pearson Addison-Wesley, Upper Saddle River, NJ, 3 edition, 2007.

[5] D. Scott M. O. Rabin. Finite Automata and Their Descision Problems. *IBM Journal*, pages 114–125, April 1959.

[6] M. Michel. Complementation is more difficult with automata on infinite words. Manuscript, 1988.

[7] Moshe Y. Vardi Ming-Hsien Tsai, Seth Fogarty and Yih-Kuen Tsay. State of Büchi Complementation. In Michael Domaratzki and Kai Salomaa, editors, *Implementation and Application of Automata*, volume 6482 of *Lecture Notes in Computer Science*, pages 261–271. Springer Berlin Heidelberg, 2011.

[8] Moshe Y. Vardi Ming-Hsien Tsai, Seth Fogarty and Yih-Kuen Tsay. State of Büchi Complementation (Full Version). pages 1–16, 2011.

[9] Sven Schewe. Büchi Complementation Made Tight. In *Proceedings of the 26th International Symposium on Theoretical Aspects of Computer Science*, pages 661–672. IBFI Schloss Dagstuhl, Freiburg, Germany, 2009.

[10] Wolfgang Thomas. Lecture notes in computer science. In Grzegorz Rozenberg and Arto Salomaa, editors, *Languages, Automata, and Logic*, Handbook of Formal Languages, pages 389–455. Springer Berlin Heidelberg, 1997.

[11] Q. Yan. Lower bounds for complementation of omega-automata via the full automata technique. *Logical Methods in Computer Science*, 4:1–20, 2008.

Index

A. Complementary Büchi automata

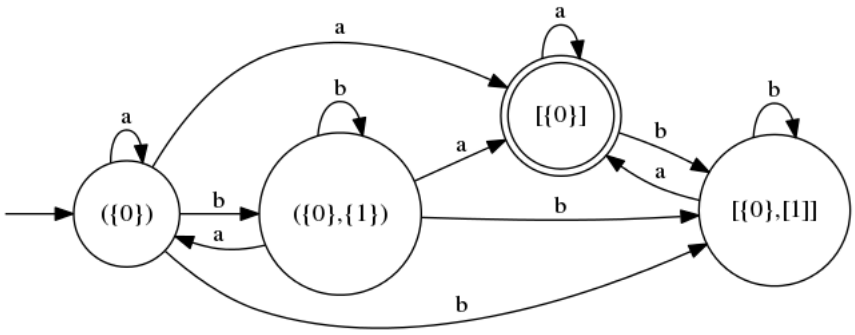


Figure A.1.: Complementary Büchi automaton of example 1

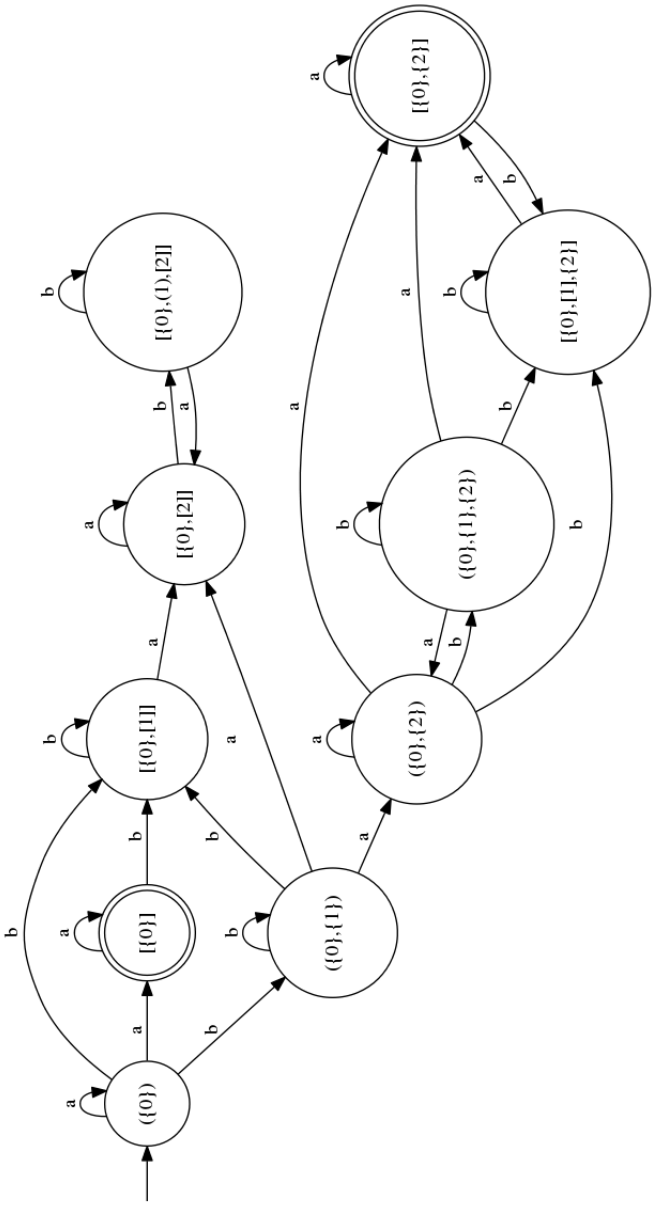


Figure A.2.: Complementary Büchi automaton of example 2

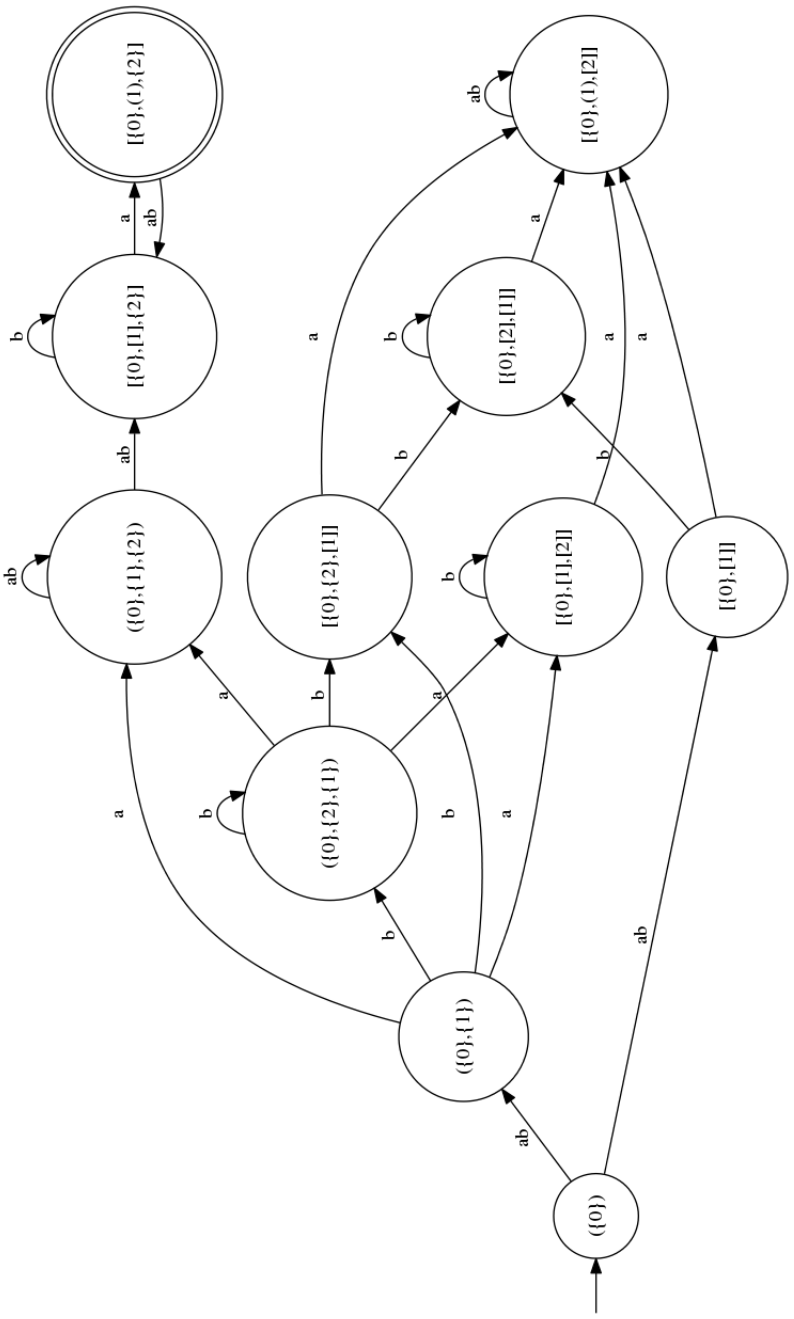


Figure A.3.: Complementary Büchi automaton of example 3