# Empirical Performance Investigation of a Büchi Complementation Construction

Daniel Weibel

June 27, 2015

**Abstract**

This will be the abstract.

**Acknowledgements**

# Contents

# Chapter 1

# Büchi Complementation

## Contents

## 1.1 Büchi Automata and Other $\omega$-Automata

### 1.1.1 Büchi Automata

Büchi automata are a type of so-called $\omega$-automata ("omega"-automata). $\omega$-automata are finite state automata that process *infinite* words rather than finite words. This means that an $\omega$-automaton never "stops" reading a word, because the word it is reading has an infinite number of symbols. But still, $\omega$-automata can accept or reject the words they read. This works by the means of a special acceptance condition, which is different from the acceptance condition of automata on finite words. For the case of Büchi automata, this is the Büchi acceptance condition that we will describe below.

#### Büchi Acceptance Condition

A Büchi automaton $A$ is defined by the 5-tuple $A = (Q, \Sigma, q_0, \delta, F)$ with the following components:

- $Q$: a finite set of states
- $\Sigma$: a finite alphabet
- $q_0$: an initial state, $q_0 \in Q$
- $\delta$: a transition function, $\delta : Q \times \Sigma \to 2^Q$
- $F$: a set of accepting states, $F \in 2^Q$

We denote by $\Sigma^\omega$ the set of all words of infinite length over the alphabet $\Sigma$. A Büchi automaton runs on the elements of $\Sigma^\omega$. In the following, we define the acceptance behaviour of a Büchi automaton $A$ on a word $\alpha \in \Sigma^\omega$.

- A *run* of Büchi automaton $A$ on a word $\alpha \in \Sigma^\omega$ is a sequence of states $q_0 q_1 q_2 \ldots$ such that $q_0$ is $A$'s initial state and $\forall i \geq 0 : q_{i+1} \in \delta(q_i, \alpha_i)$
- $\inf(\rho) \in 2^Q$ is the set of states that occur infinitely often in a run $\rho$
- A run $\rho$ is accepting if an only if $\inf(\rho) \cap F \neq \varnothing$
- A Büchi automaton $A$ accepts a word $\alpha \in \Sigma^\omega$ if and only if there is an accepting run of $A$ on $\alpha$

#### Expressivity

Büchi automata are expressively equivalent to the $\omega$-regular languages. This means that every language recognised by a Büchi automaton is an $\omega$-regular language, and that for every $\omega$-regular language there exists a Büchi automaton that recognises it. This property has been proved by Büchi himself in his initial publication in 1962 [4].

However, this equivalence does only hold for *non-deterministic* Büchi automata. Deterministic Büchi automata are less expressive than non-deterministic Büchi automata. In particular, the class of languages represented by deterministic Büchi automata is a strict subset of the class of languages represented by non-deterministic Büchi automata. This property has also been proved by Büchi in his 1962 paper [4].

The lower expressivity of deterministic Büchi automata means that there exist languages that can be recognised by a non-deterministic Büchi automaton, but not by a deterministic one. A typical example is the language $L = (0+1)^* 1^\omega$. This is the language of all words consisting of 0 and 1 with a finite number of 0 and an infinite number of 1. There are famous proofs that this language $L$ can be recognised by a non-deterministic Büchi automaton, but not by a deterministic Büchi automaton [38][23]. In practice, this means that Büchi automata can, in general, not be determinised.

This differentiates Büchi automata from classical finite state automata on finite words. Non-deterministic finite state automata (NFA) are expressively equivalent to deterministic finite state automata (DFA). This means that every NFA can be turned into an equivalent DFA. This determinisation is typically done with the famous subset construction that has been introduced by Rabin and Scott in 1959 [21].

The fact that Büchi automata can in general not be determinised is the main reason that Büchi complementation is such a hard problem [19]. We will see why below.

**Complementation**

Büchi automata are closed under complementation. This means that the complement of every Büchi automaton is in turn a Büchi automaton. This result has been proved by Büchi in his seminal paper [4].

The difficulty of the concrete complementation problem does however strongly depend on whether the Büchi automaton is deterministic or non-deterministic. For deterministic Büchi automata, the complementation is "easy" and regarded as a solved problem. There is a complementation construction introduced by Kurshan in 1987 that complements a deterministic Büchi automaton in polynomial time [12]. The resulting complement is a non-deterministic Büchi automaton and has a size that is at most the double of the input automaton.

For non-deterministic Büchi automata, on the other hand, the complementation problem is much more difficult. The main reason is, as mentioned, the fact that non-deterministic Büchi automata cannot be determinised. If they could be determinised, then all non-deterministic Büchi automata could be complemented by determinising them, and complementing the resulting deterministic Büchi automaton with, for example, the efficient Kurshan construction. If the determinisation construction is efficient (that is, has polynomial complexity), then the entire complementation procedure for the non-deterministic Büchi automata would be efficient.

However, since non-deterministic Büchi automata cannot be determinised, this simple approach of complementation is not possible. Therefore, other ways of complementing non-deterministic Büchi automata have to be found. Such ways exist, but they turned out to have a very high (superpolynomial) complexity. This is the Büchi complementation problem.

### 1.1.2 Other $\omega$-Automata

Introduce acronyms NBW, DBW, NMW, etc.

- Specification

- Expressivity: - NBW: omega-regular languages (like other omega-automata) - DBW: subset of NBW –¿ Determinisation not possible

- Complementation: - Closed under complementation - DBW: complementation easy (Kurshan's construction) - NBW: since determinisation is not possible (see above), complementation is not straightforward.

- Other omega-automata

## 1.2 Run Analysis in Non-Deterministic Büchi Automata

## 1.3 Büchi Complementation Constructions

Review of proposed Büchi complementation constructions and their complexity Introduce worst-case complexity as a performance metric Mention the four basic approaches

## 1.4 Empirical Performance Investigations

## 1.5 Preliminaries

### 1.5.1 Büchi Automata

Büchi automata have been introduced in 1962 by Büchi [4] in order to show the decidability of monadic second order logic; over the successor structure of the natural numbers [3].

he had proved the decidability of the monadic-second order theory of the natural numbers with successor func- tion by translating formulas into finite automata [39] (p. 1)

Büchi needed to create a complementation construction (proof the closure under complementation of Büchi automata) in order to prove Büchi's Theorem.

Büchi's Theorem: S1S formulas and Büchi automata are expressively equivalent (there is a NBW for every S1S formula, and there is a S1S formula for every NBW).

### Definitions

Informally speaking, a Büchi automaton is a finite state automaton running on input words of infinite length. That is, once started reading a word, a Büchi automaton never stops. A word is accepted if it results in a run (sequence of states) of the Büchi automaton that includes infinitely many occurrences of at least one accepting state.

More formally, a Büchi automaton $A$ is defined by the 5-tuple $A = (Q, \Sigma, q_0, \delta, F)$ with the following components.

- $Q$: a finite set of states
- $\Sigma$: a finite alphabet
- $q_0$: an initial state, $q_0 \in Q$
- $\delta$: a transition function, $\delta : Q \times \Sigma \to 2^Q$
- $F$: a set of accepting states, $F \in 2^Q$

We denote by $\Sigma^\omega$ the set of all words of infinite length over the alphabet $\Sigma$. A Büchi automaton runs on the elements of $\Sigma^\omega$. In the following, we define the acceptance behaviour of a Büchi automaton $A$ on a word $\alpha \in \Sigma^\omega$.

- A *run* of Büchi automaton $A$ on a word $\alpha \in \Sigma^\omega$ is a sequence of states $q_0 q_1 q_2 \ldots$ such that $q_0$ is $A$'s initial state and $\forall i \geq 0 : q_{i+1} \in \delta(q_i, \alpha_i)$
- $\inf(\rho) \in 2^Q$ is the set of states that occur infinitely often in a run $\rho$
- A run $\rho$ is accepting if an only if $\inf(\rho) \cap F \neq \varnothing$
- A Büchi automaton $A$ accepts a word $\alpha \in \Sigma^\omega$ if and only if there is an accepting run of $A$ on $\alpha$

The set of all the words that are accepted by a Büchi automaton $A$ is called the *language* $L(A)$ of $A$. Thus, $L(A) \subseteq \Sigma^\omega$. On the other hand, the set of all words of $\Sigma^\omega$ that are rejected by $A$ is called the *complement language* $\overline{L(A)}$ of $A$. The complement language can be defined as $\overline{L(A)} = \Sigma^\omega \setminus L(A)$.

Büchi automata are closed under union, intersection, concatenation, and complementation [38].

Continued/discontinued runs

A deterministic Büchi automaton (DBW) is a special case of a non-determnistic Büchi automaton (NBW). A Büchi automaton is a DBW if $|\delta(q, \alpha)| = 1$, $\forall q \in Q, \forall \alpha \in \Sigma$. That is, every state has for every alphabet symbol exactly one successor state. A DBW can also be defined directly by replacing the transition function $\delta : Q \times \Sigma \to 2^Q$ with $\delta : Q \times \Sigma \to Q$ in the above definition.

### Expressiveness

It has been showed by Büchi that NBW are expressively equivalent the $\omega$-regular languages [4]. That means that every language that is recognised by a NBW is a $\omega$-regular language, and on the other hand, for every $\omega$-regular language there exists a NBW recognising it.

However, this equivalence does not hold for DBW (Büchi showed it too). There are $\omega$-regular languages that cannot be recognised by any DBW. A typical example is the language $(0 + 1)^* 1^\omega$. This is the language of all infinite words of 0 and 1 with only finitely many 0. It can be shown that this language can be recognised by a NBW (it is thus a $\omega$-regular language) but not by a DBW [38][23]. The class of

languages recognised by DBW is thus a strict subset of $\omega$-regular languages recognised by NBW. We say that DBW are less expressive than NBW.

An implication of this is that there are NBW for which no DBW recognising the same language exists. Or in other words, there are NBW that cannot be converted to DBW. Such an inequivalence is not the case, for example, for finite state automata on finite words, where every NFA can be converted to a DFA with the subset construction [7][21]. In the case of Büchi automata, this inequivalence is the main cause that Büchi complementation problem is such a hard problem [19] and until today regarded as unsolved.

### 1.5.2 Other $\omega$-Automata

After the introduction of Büchi automata in 1962, several other types of $\omega$-automata have been proposed. The best-known ones are by Muller (Muller automata, 1963) [17], Rabin (Rabin automata, 1969) [22], Streett (Streett automata, 1982) [27], and Mostowski (parity automata, 1985) [16].

All these automata differ from Büchi automata, and among each other, only in their acceptance condition, that is, the condition for accepting or rejecting a run $\rho$. We can write a general definition of $\omega$-automata that covers all of these types as $(Q, \Sigma, q_0, \delta, Acc)$. The only difference to the 5-tuple defining Büchi automata is the last element, $Acc$, which is a general acceptance condition. We list the acceptance condition of all the different $\omega$-automata types below [13]. Note that again a run $\rho$ is a sequence of states, and $\inf(\rho)$ is the set of states that occur infinitely often in run $\rho$.

| **Type** | **Definitions** | **Run $\rho$ accepted if and only if...** |
|---|---|---|
| Büchi | $F \subseteq Q$ | $\inf(\rho) \cap F \neq \varnothing$ |
| Muller | $F \subseteq 2^Q$ | $\inf(\rho) \in F$ |
| Rabin | $\{(E_1, F_1), \ldots, (E_r, F_r)\}$, $E_i, F_i \subseteq Q$ | $\exists i : \inf(\rho) \cap E_i = \varnothing \wedge \inf(\rho) \cap F_i \neq \varnothing$ |
| Streett | $\{(E_1, F_1), \ldots, (E_r, F_r)\}$, $E_i, F_i \subseteq Q$ | $\forall i : \inf(\rho) \cap E_i \neq \varnothing \vee \inf(\rho) \cap F_i = \varnothing$ |
| Parity | $c : Q \to \{1, \ldots, k\}$, $k \in \mathbb{N}$ | $\min\{c(q) \mid q \in \inf(\rho)\} \bmod 2 = 0$ |

In the Muller acceptance condition, the set of infinitely occuring states of a run ($\inf(\rho)$) must match a predefined set of states. The Rabin and Streett conditions use pairs of state sets, so-called accepting pairs. The Rabin and Streett conditions are the negations of each other. This allows for easy complementation of deterministic Rabin and Streett automata [13], which will be used for certain Büchi complementation construction, as we will see in Section1.8. The parity condition assigns a number (color) to each state and accepts a run if the smallest-numbered of the infinitely often occuring states has an even number. For all of these automata there exist non-deterministic and deterministic versions, and we will refer to them as NMW, DMW (for non-deterministic and deterministic Muller automata), and so on.

In 1966, McNaughton made an important proposition, known as *McNaughton's Theorem* [14]. Another proof given in [28]. It states that the class of languages recognised by deterministic Muller automata are the $\omega$-regular languages. This means that non-deterministic Büchi automata and deterministic Muller automata are equivalent, and consequently every NBW can be turned into a DMW. This result is the base for the determinisation-based Büchi complementation constructions, as we will see in Section 1.8.2.

It turned out that also all the other types of the just introduced $\omega$-automata, non-deterministic and determinstic, are equivalent among each other [23][10][9][13][28]. This means that all the $\omega$-automata mentioned in this thesis, with the exception of DBW, are equivalent and recognise the $\omega$-regular languages. This is illustrated in Figure 1.1

### 1.5.3 Complementation of Büchi Automata

Büchi automata are closed under complementation. This result has been proved by Büchi himself when he introduced Büchi automata in [4]. Basically, this means that for every Büchi automata $A$, there exists another Büchi automaton $B$ that recognises the complement language of $A$, that is, $L(B) = \overline{L(A)}$.

It is interesting to see that this closure does not hold for the specific case of DBW. That means that while for every DBW a complement Büchi automaton does indeed exist, following from the above closure

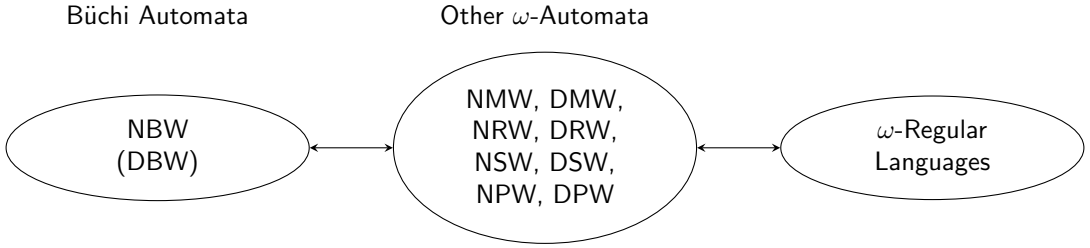



Figure 1.1: Non-deterministic Büchi automata (NBW) are expressively equivalent to Muller, Rabin, Streett, and parity automata (both deterministic and non-deterministic), and to the $\omega$-regular languages. Deterministic Büchi automata (DBW) are less expressive than NBW.

property for Büchi automata in general, this automaton is not necessarily a DBW. The complement of a DBW may be, and often is, as we will see, a NBW. This result is proved in [28] (p. 15).

The problem of Büchi complementation consists now in finding a procedure (usually called a construction) that takes as input any Büchi automaton $A$ and outputs another Büchi automaton $B$ with $L(B) = \overline{L(A)}$, as shown below.


For complementation of automata in general, construction usually differ depending on whether the input automaton $A$ is deterministic or non-deterministic. Complementation of deterministic automata is often simpler and may sometimes even provide a solution for the complementation of the non-deterministic ones.

To illustrate this, we can briefly look at the complementation of the ordinary finite state automata on finite words (FA). FA are also closed under complementation [7] (p. 133). A DFA can be complemented by simply switching its accepting and non-accepting states [7] (p. 133). Now, since NFA and DFA are equivalent [7] (p. 60), a NFA can be complemented by converting it to an equivalent DFA first, and then complement this DFA. Thus, the complementation construction for DFA provides a solution for the complementation of NFA.

Returning to Büchi automata, the case is more complicated due to the inequivalence of NBW and DBW. The complementation of DBW is indeed "easy", as was the complementation of DFA. There is a construction, introduced in 1987 by Kurshan [12], that can complement a DBW to a NBW in polynomial time. The size of the complement NBW is furthermore at most the double of the size of the input DBW.

If now for every NBW there would exist an equivalent DBW, an obvious solution to the general Büchi complementation problem would be to transform the input automaton to a DBW (if it is not already a DBW) and then apply Kurshan's construction to the DBW. However, as we have seen, this is not the case. There are NBW that cannot be turned into equivalent DBW.

Hence, for NBW, other ways of complementing them have to be found. In the next section we will review the most important of these "other ways" that have been proposed in the last 50 years since the introduction of Büchi automata. The Fribourg construction, that we present in Chapter **??**, is another alternative way of achievin this same aim.

### 1.5.4 Complexity of Büchi Complementation

Constructions for complementing NBW turned out to be very complex. Especially the blow-up in number of states from the input automaton to the output automaton is significant. For example, the original complementation construction proposed by Büchi [4] involved a doubly exponential blow-up. That is, if the input automaton has $n$ states, then for some constant $c$ the output automaton has, in the worst case, $c^{c^n}$ states [26]. If we set $c$ to 2, then an input automaton with six states would result in a complement automaton with about 18 quintillion ($18 \times 10^{18}$) states.

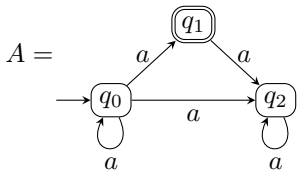



Figure 1.2: Example NBW $A$ that will be used in different places throughout this thesis. The alphabet of $A$ consists of the single symbol $a$, consequently, $A$ can only process the single $\omega$-word $a^\omega$. This word is rejected by $A$, so the automaton is empty.

Generally, state blow-up functions, like the $c^{c^n}$ above, mean the absolute worst cases. It is the maximum number of states a construction *can* produce. For by far most input automata of size $n$ a construction will produce much fewer states. Nevertheless, worst case state blow-ups are an important (the most important?) performance measure for Büchi complementation constructions. A main goal in the development of new constructions is to bring this number down.

A question that arises is, how much this number can be brought down? Researchers have investigated this question by trying to establish so called lower bounds. A lower bound is a function for which it is proven that no state blow-up of any construction can be less than it. The first lower bound for Büchi complementation has been established by Michel in 1988 at $n!$ [15]. This means that the state blow-up of any Büchi complementation construction can never be less than $n!$.

There are other notations that are often used for state blow-ups. One has the form $(xn)^n$, where $x$ is a constant. Michel's bound of $n!$ would be about $(0.36n)^n$ in this case [41]. We will often use this notation, as it is convenient for comparisons. Another form has 2 as the base and a big-O term in the exponent. In this case, Michel's $n!$ would be $2^{O(n \log n)}$ [41].

Michel's lower bound remained valid for almost two decades until in 2006 Yan showed a new lower bound of $(0.76n)^n$ [41]. This does not mean that Michel was wrong with his lower bound, but just too reserved. The best possible blow-up of a construction can now be only $(0.76n)^n$ and not $(0.36n)^n$ as believed before. In 2009, Schewe proposed a construction with a blow-up of exactly $(0.76n)^n$ (modulo a polynomial factor) [25]. He provided thus an upper bound that matches Yan's lower bound. The lower bound of $(0.76n)^n$ can thus not rise any further and seems to be definitve.

Maybe mention note on exponential complexity in [38] p. 8.

## 1.6 Run Analysis

A deterministic automaton has exactly one run on every word. A non-deterministic automaton, on the other hand, may have multiple runs on a given word. The analysis of all runs of a word, in some form or another, an integral part of Büchi complemenation constructions. Remember that a non-deterministic automaton accepts a word if there is *at least one* accepting run. Consequently, a word is rejected if only if *all* the runs are rejecting. That is, if $B$ is the complement Büchi automaton of $A$, then $B$ has to accept a word $w$ if and only if *all* the runs of $A$ on $w$ are rejecting. For constructing the complement $B$, we have thus to consider all the possible runs of $A$ on every word.

There are two main data structures that are used for analysing the runs of a non-deterministic automaton on a word. These are trees and DAGs (directed acyclic graphs) [40]. In this section, we present both of them. We put however emphasis on trees, as they are used by the subset-tuple construction presented in Chapter **??**.

### 1.6.1 From Run Trees to Split Trees

The one tree data-structure that truly represents *all* the runs of an automaton on a word are run trees. The other variants of trees that we present in this section are basically derivations of run trees that sacrifice information about individual runs, by merging or discarding some of them, at the benefit of

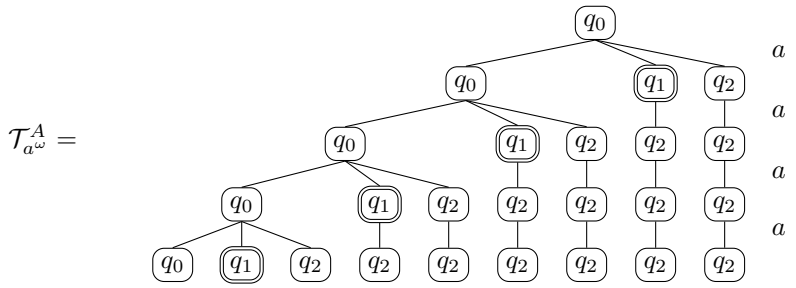



Figure 1.3: Automaton $A$ and the first five levels of the run tree of the runs of $A$ on the word $a^\omega$.

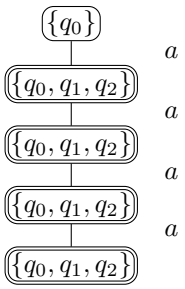



Figure 1.4: Automaton $A$ and the first five levels of the run tree of the runs of $A$ on the word $a^\omega$.

becoming more concise. Figure 1.8 shows the first few levels of the run tree of the example automaton $A$ from Figure 1.2 on the word $a^\omega$.

In a run tree, every vertex represents a single state and has a descendant for every $a$-successor of this state, if $a$ is the current symbol of the word. A run is thus represented as a branch of the run tree. In particular, there is a one-to-one mapping between branches of a run tree and runs of the automaton on the given word.

We mentioned that the other tree variants that we talk about in this section, split trees and reduced split trees, make run trees more compact by not keeping information about individual runs anymore. They thereby relinquish the one-to-one mapping between branches of the tree and runs. Let us look at one extreme of this aggregation of runs which is done by the subset construction. This will motivate the definition of split trees, and at the same time shows why the subset construction fails for determinising NBW [1].

For determinising an automaton $A$, the subset construction in effect merges all the diverse runs of $A$ on word $w$ to one single run by merging all the states on a level of the corresponding run tree to one single state. This state will be a state of the output automaton $B$, and is labelled with the set of $A$-states it includes. Figure 1.4 shows this effect with our example automaton from Figure 1.2 and the word $a^\omega$.

Clearly, this form of tree created by the subset construction is the most concise form a run tree can be brought to. However, almost all information about individual runs in $A$ has been lost. All that can be said by looking at the structure in Figure 1.4 is that there must be at least one continued $A$-run on $a^\omega$(all the other runs visiting the other $A$-states on each level, might be discontinued). But which states a possible continued run visits cannot be deduced.

This lack of identification $A$-runs is the reason why the subset construction fails for determinising Büchi automata. Note that a $B$-state of the subset construction is accepting if the set of $A$-states it represents contains at least one accepting $A$-state. For our example, this means that the state $\{q_0, q_1, q_2\}$ is accepting (this is also indicated in Figure 1.4). This state is visited infinitely often by the unified run on $a^\omega$. Hence, the DBW $B$, resulting from applying the subset construction to the NBW $A$, accepts $a^\omega$while $A$ does not accept it.

By looking closer at the trees in Figure 1.4 and 1.8, the reason for this problem becomes apparent. If we look for example at the second level of the subset-construction tree we can deduce that there must

[1] The NBW that *can* be turned into DBW.

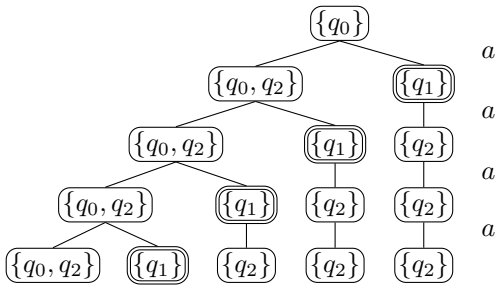



Figure 1.5: Automaton $A$ and the first five levels of the split tree of the runs of $A$ on the word $a^\omega$.

be an $A$-run that visits the accepting $A$-state $q_1$. Let us call this run $r_{q_1}$. However, at the third level, we cannot say anything about $r_{q_1}$ anymore, whether it visits one of the non-accepting states or again $q_1$ on the third level, or whether it even ended at the second level. In turn, what we know on the third level in our example is that there is again an $A$-run, $r'_{q_1}$, that visits $q_1$. However, whether $r'_{q_1}$ is $r_{q_1}$, and in turn the future of $r'_{q_1}$ cannot be deduced. In our example we end up with the situation that there are infintiely many visits to $q_1$ in the unifed $B$-run, but we don't know if the reaon for this are one or more $A$-runs that visit $q_1$ infinitely often, or infinitely man $A$-runs where each one visits $q_1$ only finitely often (the way it is in our example). In the first case, it would be correct ot accept the $B$-run, in the second case however it would be wrong as the input automaton $A$ does not accept the word. The subset construction does not distinguish these two cases and hence the determinised automaton $B$ may accept words that the input automaton $A$ rejects. In general, the language of an output DBW of the subset construction is a superset of the language of the input NBW.

This raises the question how the subset construction can be minimally modified such that the output automaton is equivalent to the input automaton. One solution is to not mix accepting and non-accepting $A$-states in the $B$-states. That is, instead of creating one $B$-state that contains all the $A$-states, as in the subset construction, one creates two $B$-states where one contains the accepting $A$-states and the other the non-accepting $A$-states. Such a construction has been formalised in [37]. The output automaton $B$ is then not deterministic, but it is equivalent to $A$. The type of run analysis trees that correspond to this refined subset construction are split trees. Figure 1.9 shows the first five levels of the split tree of our example automaton $A$ on the word $a^\omega$.

Let us see why the splitted subset construction produces output automata that are equivalent to the input automata. For this equivalence to hold, a branch of a reduced split tree must include infinitely many accepting vertices if and only if there is an $A$-run that visits at least one accepting $A$-state infinitely often. For an infinite branch of a split tree, there must be at least one continued $A$-run. If this infinite branch includes infinitely many accepting vertices, then this $A$-run must infinitely many times go trough an accepting $A$-state. This is certain, because an accepting vertex in a split tree contains *only* accepting $A$-states. Since there are only finitely many accepting $A$-states, the $A$-run must visit at least one of them infinitely often. On the other hand, if an $A$-run includes infinite visits to an accepting state, then this results in a branch of the split tree with infinitely many accepting vertices, since every $A$-run must be "contained" in a branch of the split tree.

Split trees can be seen as run trees where some of the branches are contracted to unified branches. In particular, a split tree unifies as many branches as possible, such that the resulting tree still correctly represents the Büchi acceptance of all the runs included in a unified branch. This can form the basis for constructions that transform an NBW to another equivalent NBW. Split trees are for example the basis for Muller-Schupp trees in Muller and Schupp's Büchi determinisation construction [18], cf. [2].

### 1.6.2 Reduced Split Trees

It turns out that split trees can be compacted even more. The resulting kind of tree is called reduced split tree. In a reduced split tree, each $A$-state occurs at most once on every level. Figure 1.6 shows the reduced split tree corresponding to the split tree in Figure 1.9. As can be seen, only one occurrence of each $A$-state on each level is kept, the other are discarded. To allow this, however, the order of the

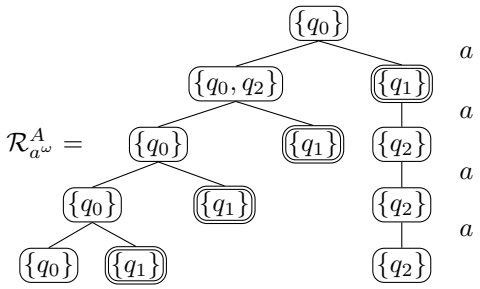



Figure 1.6: Automaton $A$ and the first five levels of the reduced split tree of the runs of $A$ on the word $a^\omega$.

accepting and non-accepting siblings in the tree matters. Either the accepting child is always put to the right of the non-accepting child (as in our example in Figure 1.6, or vice versa. We call the former variant a right-to-left reduced split tree, and the latter a left-to-right reduced split tree. In this thesis, we will mainly adopt the right-to-left version.

A reduced split tree is constructed like a split tree, with the following restrictions.

- For determining the vertices on level $n+1$, the parent vertices on level $n$ have to be processed from right to left
- From every child vertex on level $n+1$, subtract the $A$-states that occur in some vertex to the right of it on level $n+1$
- Put the accepting child to the right of the non-accepting child on level $n+1$

A very important property of reduced split trees is that they have a fixed width. The width of a tree is the maximal number of vertices on a level. For reduced split trees, this is the number of states of the input automaton $A$. As we will see, the subset-tuple construction (like other slice-based constructions) uses levels of a reduced split tree as states othe output automaton, and the limited size of these levels ensures an upper bound on the number of states these constructions can create.

By deleting $A$-states from a level of a reduced split tree, we actually delete $A$-runs that reach the same $A$-state on the same substring of the input word. For example, in the split tree in Figure 1.9 we see that there are at least for $A$-runs on the string $aaaa$ from the inital state $q_0$ to $q_2$. The reduced split tre in Figure 1.6, however, contains only one run on $aaaa$ from $q_0$ to $q_2$, namely the rightmost branch of the tree. The information about all the other runs is lost. This single run that is kept is very special and, as we will see shorty, it represents the deleted runs. We will call this run the *greedy run*. The reason for calling it greedy is that it visits an accepting state earlier than any of the deleted runs. In a right-to-left reduced split tree, the greedy run is always the rightmost of the runs from the root to a certain $A$-state on a certain level. In left-to-right reduced split tree, the greedy run would in turn be the leftmost of these runs.

We mentioned that the greedy run somehow represents the deleted runs. More precisely, the relation is as follows and has been proved in [39]: if any of the deleted runs is a prefix of a run that is Büchi-accepted (that is, an infinite run visiting infinitely many accepting $A$-states), then the greedy run is so too. That means that if the greedy cannot be expanded to a Büchi-accepting run, then none of the deleted runs could be either. Conversely, if any of the deleted runs could become Büchi-accepting, then the greedy run can so too. So, the greedy run is sufficient to indicate the existence or non-existence of a Büchi-accepting run with this prefix, and it is safe to delete all the other runs.

### 1.6.3 Run DAGs

DAGs (directed acyclic graphs) are, after trees, the second form for analysing the runs of a non-deterministic automaton on a given word. A run DAG has the form of a matrix with one column for each $A$-state and a row for each position in the word. The directed edges go from the vertices on one row to the vertices on the next row (drawn below) according to the transitions in the automaton on the

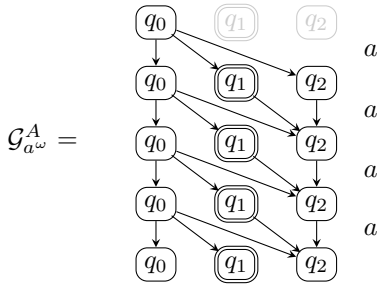



Figure 1.7: Automaton $A$ and the first five levels of the run DAG of the runs of $A$ on the word $a^\omega$.

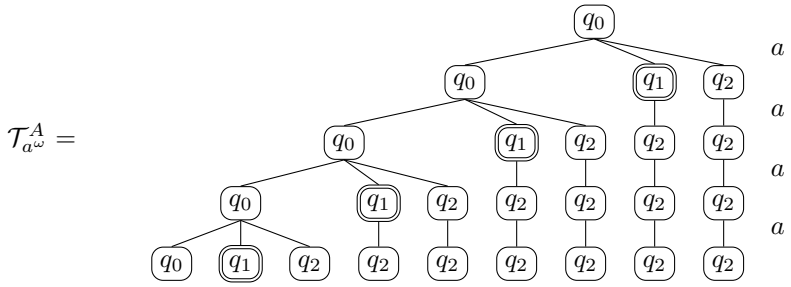



Figure 1.8: Automaton $A$ and the first five levels of the run tree of the runs of $A$ on the word $a^\omega$.

current input symbol. Figure 1.12 shows the first five rows of the run DAG of the example automaton in Figure 1.2 on the word $a^\omega$.

Like run trees, run DAGs represent all the runs of an automaton on a given word. However, run DAGs are more compact than run trees. The rank-based complementation constructions are based on run DAGs.

## 1.7 Run Analysis

In a deterministic automaton every word has exactly one run. In a non-deterministic automaton, however, a given word may have multiple runs. The analysis of the different runs of a given word on an automaton plays an important role in the complementation of Büchi automata. There are several techniques for analysing the runs of a word that we present in this section.

### 1.7.1 Run Trees

The simplest of run analyis technique is the run tree. A run tree is a direct unfolding of all the possible runs of an automaton $A$ on a word $w$. Each vertex $v$ in the tree represents a state of $A$ that we denote by $\sigma(v)$. The descendants of a vetex $v$ on level $i$ are vertices representing the successor states of $\sigma(n)$ on the symbol $w(i+1)$ in $A$. In this way, every branch of the run tree originating in the root represents a possible run of automaton $A$ on word $w$.

Figure 1.8 shows an example automaton $A$ and the first five levels of the run tree for the word $w = a^\omega$ (infinite repetitions of the symbol $a$). Each branch from the root to one of the leaves represents a possible way for reading the first four positions of $w$. On the right, as a label for all the edges on the corresponding level, is the symbol that causes the depicted transitions.

($A$ does not accept any word, it is empty. The only word it could accept is $a^\omega$ which it does not accept.)

We define by the width of a tree the maximum number of vertices occurring at any level [18]. Clearly, for $\omega$-words the width of a run tree may become infinite, because there may be an infinite number of levels and each level may have more vertices than the previous one.

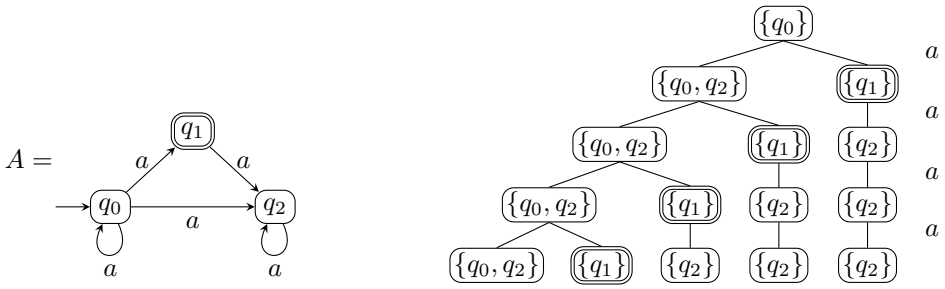



Figure 1.9: Automaton $A$ and the first five levels of the split tree of the runs of $A$ on the word $a^\omega$.

### 1.7.2 Failure of the Subset-Construction for Büchi Automata

Run trees allow to conveniently reveal the cause why the subset construction does not work for determinising Büchi automata, which in turn motivates the basic idea of the next run analysis technique, split trees.

Applying the subset construction to the same NBW $A$ used in the previous example, we get the automaton $A'$ shown in Figure **??**. Automaton $A'$ is indeed a DBW but it accepts the word $a^\omega$ which $A$ does not accept. If we look at the run tree of $A$ on word $a^\omega$, the subset construction merges the individual states occuring at level $i$ of the tree to one single state $s_i$, which is accepting if at least one of its components is accepting. Equally, the individual transitions leading to and leaving from the individual components of $s_i$ are merged to a unified transition. The effect of this is that we lose all the information about these indiviudal transitions. This fact is depicted in Figure **??**. For the NFA acceptance condition this does not matter, but for NBW it is crucial because the acceptance condition depends on the history of specific runs. In the example in Figure **??**, a run $\rho$ of $A$ visiting the accepting state $q_1$ can never visit an accepting state anymore even though the unified run of which $\rho$ is part visits $q_1$ infinitely often. But the latter is achieved by infinitely many different runs each visiting $q_1$ just once.

It turns out that enough information about individual runs to ensure the Büchi acceptance condition could be kept, if accepting and non-accepting state are not mixed in the subset construction. Such a constructio has bee proposed in [37]. Generally, the idea of treating accepting and non-accepting states separately is important in the run analyis of Büchi automata.

### 1.7.3 Split Trees

Split trees can be seen as run trees where the accepting and non-accepting descendants of a node $n$ are aggregated in two nodes. We will call the former the *accepting child* and the latter the *non-accepting child* of $n$. Thus in a split tree, every node has at most two descendants (if either the accepting or the non-accepting child is empty, it is not added to the tree), and the nodes represent sets of states rather than individual states. Figure 1.9 shows the first five levels of the split tree of automaton $A$ on the word $a^\omega$.

The order in which the accepting and non-accepting child are

The notion of split trees (and reduced split trees, see next section) has been introduced by Kähler and Wilke in 2008 for their slice-based complementation construction [8], cf. [5]. However, the idea of separating accepting from non-accepting states has already been used earlier, for example in Muller and Schupp's determinisation-based complementation construction from 1995 [18]. Formal definitions os split trees can be found in [8][5].

### 1.7.4 Reduced Split Trees

The width of a split tree can still become infinitely large. A reduced split tree limits this width to a finite number with the restriction that on any level a given state may occur at most once. This is in effect the

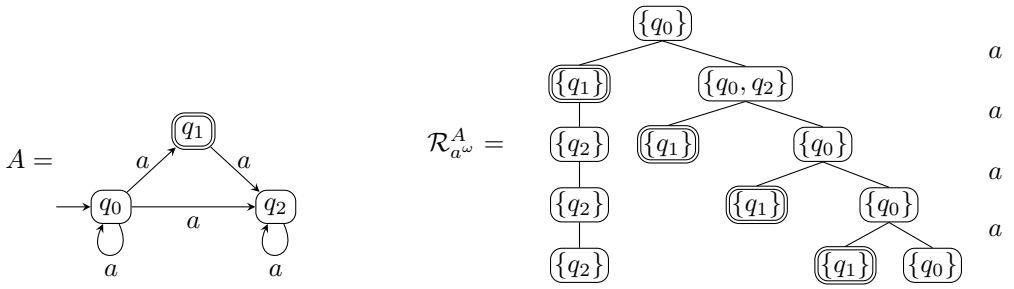



Figure 1.10: Automaton $A$ and the first five levels of the left-to-right reduced split tree of the runs of $A$ on the word $a^\omega$.

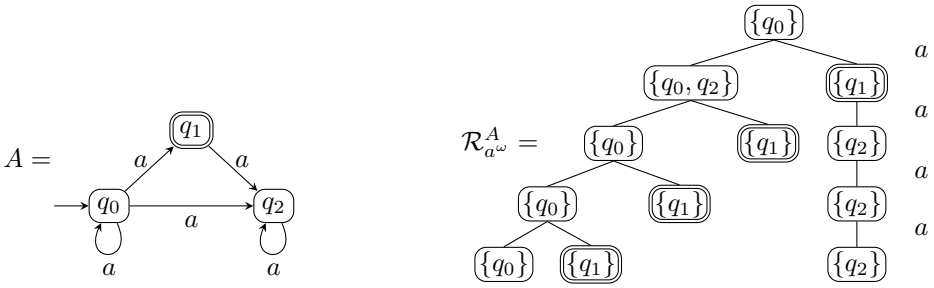



Figure 1.11: Automaton $A$ and the first five levels of the left-to-right reduced split tree of the runs of $A$ on the word $a^\omega$.

same as saying that if in a split tree there are multiple ways of going from the root to state $q$, then we keep only one of them.

### 1.7.5 Run DAGs

A run DAG (DAG stands for directed acyclic graph) can be seen as a graph in matrix form with one column for every state of $A$ and one row for every position of word $w$. The edges are defined similarly than in run trees. Figure 1.12 shows the run DAG of automaton $A$ on the word $w = a^\omega$.

## 1.8 Review of Büchi Complementation Constructions

### 1.8.1 Ramsey-Based Approaches

The method is called Ramsey-based because its correctness relies on a combinatorial result by Ramsey to obtain a periodic decomposition of the possible behaviors of a Büchi automaton on an infinite word [3].

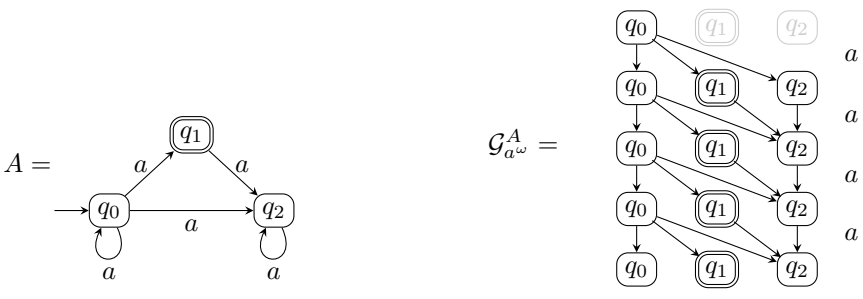



Figure 1.12: Automaton $A$ and the first five levels of the run DAG of the runs of $A$ on the word $a^\omega$.

### 1.8.2 Determinisation-Based Approaches

### 1.8.3 Rank-Based Approaches

### 1.8.4 Slice-Based Approaches

## 1.9 Empirical Performance Investigations

# Chapter 2

# The Fribourg Construction

## Contents

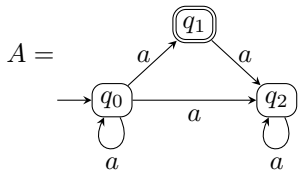



Figure 2.1: Example automaton $A$

The Fribourg construction draws from several ideas: the subset construction, run analysis based on reduced split trees, and Kurshan's construction [12] for complementing DBW. Following the classification we used in Section 1.8, it is a slice-based construction. Some of its formalisations are similar to the slice-based construction by Vardi and Wilke [39], however, the Fribourg construction has been developed independently. Furthermore, as we will see in Chapter **??**, the empirical performance of Vardi and Wilke's construction and the Fribourg construction differ considerably, in favour of the latter.

Basically, the Fribourg construction proceeds in two stages. First it constructs the so-called upper part of the complement automaton, and then adds to it its so-called lower part. These terms stem from the fact that it is often convenient to draw the lower part below the previously drawn upper part. The partitioning in these two parts is inspired by Kurshan's complementation construction for DBW. The upper part of the Fribourg construction contains no accepting states and is intended to model the finite "start phase" of a run. At every state of the upper part, a run has the non-deterministic choice to either stay in the upper part or to move to the lower part. Once in the lower part, a run must stay there forever (or until it ends if it is discontinued). That is, the lower part models the infinite "after-start phase" of a run. The lower part now includes accepting states in a sophisticated way so that at least one run on word $w$ will be accepted if and only if all the runs of the input NBW on $w$ are rejected.

As it may be apparent from this short summary, the construction of the lower part is much more involved than the construction of the upper part.

## 2.1 First Stage: Constructing the Upper Part

The first stage of the subset-tuple construction takes as input an NBW $A$ and outputs a deterministic automaton $B'$. This $B'$is the upper part of the final complement automaton $B$ of $A$. The construction of $B'$can be seen as a modified subset construction. The difference to the normal subset construction lies in the inner structure of the constructed states. While in the subset construction a state consists of a subset of the states of the input automaton, a $B'$-state in the subset-tuple construction consists of a *tuple of subsets* of $A$-states. The subsets in a tuple are pairwise disjoint, that is, every $A$-state occurs at most once in a $B'$-state. The $A$-states occurring in a $B'$-state are the same that would result from the classic subset construction. As an example, if applying the subset construction to a state $\{q_0\}$ results in the state $\{q_0, q_1, q_2\}$, the subset-tuple construction might yield the state $(\{q_0, q_2\}, \{q_1\})$ instead.

The structure of $B'$-states is determined by levels of corresponding reduced split trees. Vardi, Kähler, and Wilke refer to these levels as *slices* in their constructions [39, 8]. Hence the name slice-based approach. In the following, we will use the terms levels and slices interchangebly. A slice-based construction can work with either left-to-right or right-to-left reduced split trees. Vardi, Kähler, and Wilke use the left-to-right version in their above cited publications. In this thesis, in contrast, we will use right-to-left reduced split trees, which were also used from the beginning by the authors of the subset-tuple construction.

Figure 2.2 shows how levels of a right-to-left reduced split tree map to states of the subset-tuple construction. In essence, each node of a level is represented as a set in the state, and the order of the nodes determines the order of the sets in the tuple. [INFORMATION ABOUT ACC AND NON-ACC IS NEEDED IN THE LOWER PART BUT IMPLICIT IN THE STATES OF A]. To determine the successor of a state, say $(\{q_0, q_2\}, \{q_1\})$, one can regard this state as level of a reduced split tree, determine the next level and map this new level to a state. In the example of Figure 2.2, the successor of $(\{q_0, q_2\}, \{q_1\})$ is determined in this way to $(\{q_0\}, \{q_1\}, \{q_2\})$.

Apart from this special way of determining successor states, the construction of $B'$proceeds similarly as

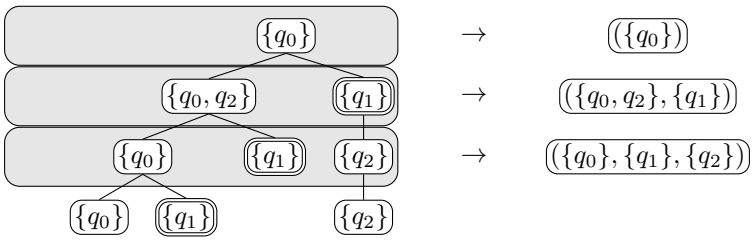



Figure 2.2: Mapping from levels of a reduced split tree, to states of the subset-tuple construction.

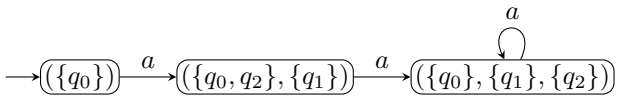



Figure 2.3: Upper part $B'$of example automaton $A$.

the subset construction. One small further difference is that if at the end of determining a successor for every state in $B'$, the automaton is not complete, it must be made complete with an *accepting* sink state. The steps for constructing $B'$from $A$ can be summarised as follows.

- Start with the state $(\{q_0\})$ if $q_0$ is the initial state of $A$
- Determine for each state in $B'$a successor for every input symbol
- It at the end $B'$is not complete, make it complete with an accepting sink state

For the example automaton $A$ in Figure 2.1, we would start with $(\{q_0\})$, determine $(\{q_0, q_2\}, \{q_1\})$ as its $a$-successor, whose $a$-successor in turn we determine a $(\{q_0\}, \{q_1\}, \{q_2\})$. The $a$-successor of $(\{q_0\}, \{q_1\}, \{q_2\})$ is $(\{q_0\}, \{q_1\}, \{q_2\})$ again what results in a loop. Figure 2.3 shows the final upper part $B'$of $A$.

## 2.2 Second Stage: Adding the Lower Part

The second stage of the subset-tuple construction adds the lower part to the upper part $B'$. The two parts together form the final complement automaton $B$. The lower part is constructed by again applying a modified subset construction to the states of the upper part $B'$. This modified subset construction is an extension of the construction for the upper part. The addition is that each set gets decorated with a colour. These colours later determine which states of the lower part are accepting states.

We divide our discussion of the lower part in two sections. In the following one (2.2.1), we explain the "mechanical" construction of the lower part, the steps that have to be done to arrive at the final complement automaton $B$. In the next section (2.3) we give the idea and intuition behind the construction and explain why it works.

### 2.2.1 Construction

As mentioned, every set of the states of the lower part gets a colour. There are three colours and we call them 0, 1, and 2. In the end we have to be able to disinguish the states of the upper part from the states of the lower part. This can be achieved by preliminarily assigning the special colour -1 to every set of the states of the upper part. After that the extended modified subset construction is applied, taking the states of the upper part (except a possible sink state) as the pre-existing states.

At first, the extended modified subset construction determines the successor tuple (without the colours) of an existing state in the same way as the construction of the upper part. We will refer to the state being created as $p$ and to the existing state as $p_{pred}$. Then, one of the colours 0, 1, or 2 is determined for each set $s$ of $p$. We denote the colour of $s$ as $c(s)$. The choice of $c(s)$ depends on three factors.

- Whether $p_{pred}$ has a set with colour 2 or not

| $p_{pred}$ has no sets with colour 2 | $s$ non-accepting | $s$ accepting |
|---|---|---|
| $c(s_{pred}) = -1$ | 0 | 2 |
| $c(s_{pred}) = 0$ | 0 | 2 |
| $c(s_{pred}) = 1$ | 2 | 2 |

| $p_{pred}$ has set(s) with colour 2 | $s$ non-accepting | $s$ accepting |
|---|---|---|
| $c(s_{pred}) = 0$ | 0 | 1 |
| $c(s_{pred}) = 1$ | 1 | 1 |
| $c(s_{pred}) = 2$ | 2 | 2 |

Figure 2.4: Colour rules.

- The colour of the predecessor set $s_{pred}$ of $s$
- Whether $s$ is an accepting or non-accepting set

The predecessor set $s_{pred}$ is the set of $p_{pred}$ that in the corresponding reduced split tree is the parent node of the node corresponding to $s$. Figure 2.4 shows the values of $c(s)$ for all possible situations as two matrices. There is one matrix for the two cases of factor 1 above ($p_{pred}$ has colour 2 or not) and the other two factors are laid out along the rows and columns of either matrix. Note that $c(s_{pred}) = -1$ is only present in the upper matrix, because in this case $p_{pred}$ is a state of the upper part and cannot contain colour 2.

We will use the following notation to denote the colour of $s$: $\widehat{s}$ if $c(s) = -1$, $s$ if $c(s) = 0$, $\overline{s}$ if $c(s) = 1$, and $\overline{\overline{s}}$ if $c(s) = 2$. Let us look now at a concrete example of this construction. We will add the lower part to the upper part $B'$in Figure 2.3, and thereby complete the complementation of the example automaton $A$ in Figure 2.1.

First of all, we assign colour $-1$ all the sets of the states of $B'$. We might then start processing the state $(\widehat{\{q_0\}})$, let us call it $p_{pred}$. The resulting successor tuple, without the colours, of $p_{pred}$ is, as in the upper part, $(\{q_0, q_2\}, \{q_1\})$. We now have to determine the colours of the sets $\{q_0, q_2\}$ and $\{q_1\}$. Since $p_{pred}$ does not contain any 2-coloured sets, we need only to consult the upper matrix in Figure 2.4. For $\{q_1\}$, the predecessor set is $\widehat{\{q_1\}}$ with colour $-1$. Furthermore $\{q_1\}$ is accepting. So, the colour of $\{q_1\}$ is 2, because we end up in the first-row, second-column cell of the upper matrix ($M_1(1, 2)$). The other set, $\{q_0, q_2\}$, in turn is non-accepting, so its colour is 0 ($M_1(1, 1)$). The successor state of $(\widehat{\{q_0\}})$ is thus $(\{q_0, q_2\}, \overline{\overline{\{q_1\}}})$.

We can then continue the construction right with this new state $(\{q_0, q_2\}, \overline{\overline{\{q_1\}}})$, and call it $p_{pred}$ in turn. The successing tuple without the colours of $p_{pred}$ is $(\{q_0\}, \{q_1\}, \{q_2\})$. Since $p_{pred}$ contains a set with colour 2, we have to consult the lower matrix of Figure 2.4 to determine the colours of $\{q_0\}$, $\{q_1\}$, and $\{q_2\}$. For $\{q_2\}$, we end up with colour 2 ($M_2(3, 1)$), because its predecessor set, which is $\overline{\overline{\{q_1\}}}$, has colour 2. $\{q_1\}$ gets colour 1 as it is accepting and its predecessor set, $\{q_0, q_2\}$, has colour 0 ($M_2(1, 2)$). $\{q_0\}$, which has the same predecessor set, gets colour 0, because it is non-accepting ($M_2(1, 1)$). The successor state of $(\{q_0, q_2\}, \overline{\overline{\{q_1\}}})$ is thus $(\{q_0\}, \overline{\{q_1\}}, \overline{\overline{\{q_2\}}})$.

The construction continues in this way until every state has been processed. The resulting automaton is shown in Figure 2.5. The last thing that has to be done is to make every state of the lower part that does not contain colour 2 accepting. In our example, this is only one state. The NBW $B$ in Figure 2.5 is the complement of the NBW $A$ in Figure 2.1, such that $L(B) = \overline{L(A)}$. This can be easily verified, since $A$ is empty and $B$ is universal (with regard to the single $\omega$-word $a^\omega$).

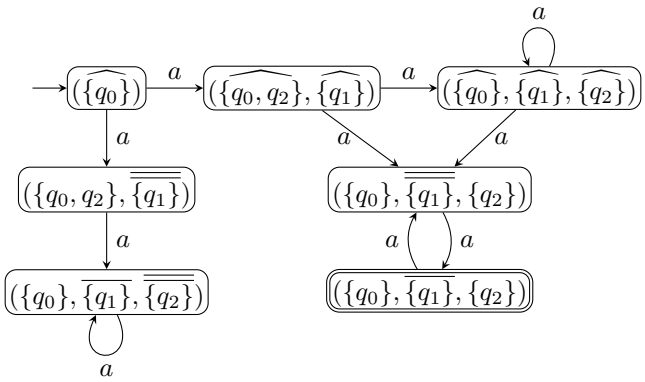



Figure 2.5: The final complement automaton $B$.

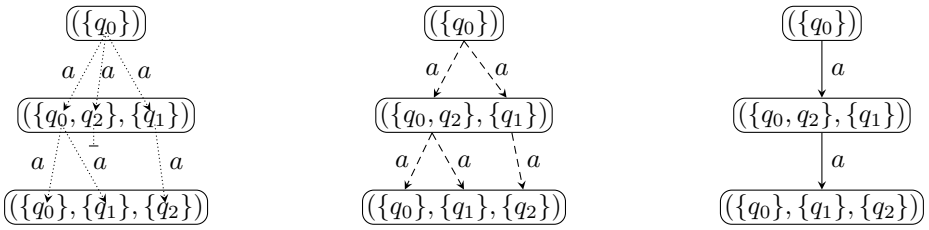



Figure 2.6: Different notions of runs.

### 2.2.2 Meaning and Function of the Colours

## 2.3 Intuition for Correctness

The general relation between a non-deterministic automaton $A$ and its complement $B$ is that a word $w$ is accepted by $B$, if and only if all the runs of $A$ on $w$ are rejecting. Of course for the subset-tuple construction, as we have just described it above, this is also true. A formal proof can be found in [1]. In this section, in contrast, we try to give an intuitive way to understand this correctness. One one hand, there is the question, if there is an accepting run of $B$ on $w$, how can we conclude that all the runs of $A$ on $w$ are rejected?

- If there is an accepting run of $B$ on $w$, how can we conclude that all the runs of $A$ on $w$ are rejected?
- If all the runs of $A$ on $w$ are rejected, how can we conclude that there must be an accepting run of $B$ on $w$?

Since this condition is on *all* runs of $A$, the construction somehow has to keep track of them.

## 2.4 Optimisations

### 2.4.1 Removal of Non-Accepting States (R2C)

### 2.4.2 Merging of Adjacent Sets (M1)

### 2.4.3 Reduction of 2-Coloured Sets (M2)

# Chapter 3

# Performance Investigation of the Fribourg Construction

## Contents



In this chapter we come to the core of this thesis, namely the empirical performance investigation of the Fribourg construction. We are interested in two things. First, how the different versions of the Fribourg construction compare to each other. That is, how do different combinations of optimisations influence the performance of the construction. Second, we want to know how the Fribourg construction performs compared to existing complementation constructions. Our main measure for the performance of a construction is the number of states of the produced complement. Throughout this thesis, we will refer to the first question as the *internal* tests, and to the second question as the *external* tests.

To do an empirical performance investigation we need an implementation of the Fribourg construction. We decided to create this implementation in the framework of an existing tool called GOAL. This is a Java tool with a graphical user interface for manipulating $\omega$-automata, and it contains implementations of various Büchi complementation constructions. In this way we can easily compare the Fribourg construction to these other construction (see external tests).

The next thing we need for an empirical performance investigation is test data. These are specific sets of automata on which all the tested construction are run. We defined two test sets. The first one, called the GOAL test set, contains a large number of randomly generated automata. The second one, called the Michel test set, contains just a small number of automata that have a special property.

Having an implementation and test data, the experiments need to be executed. Our chosen implementation approach and test data results in heavy computation tasks, that require a lot of computation power

and time. We therefore decided to execute the experiments in a professional high-performance computing (HPC) environment. This environment is the Linux-based HPC computing cluster, called UBELIX, at the University of Bern.[1]

In this chapter, we describe each of these points in a separate section. Section 3.3 also includes our experimental setup, that is, a listing of the concrete construction versions that we tested, the allocated computing resources, and so on. The results of the experiments will finally be presented in Chapter 4.

## 3.1 Implementation

As mentioned, we implemented the Fribourg construction as part of the GOAL tool. This is possible thanks to the extensible plugin architecture of GOAL which allows to write plugins that contain additional functionality for GOAL. Our implementation of the Fribourg construction has therefore the form of a GOAL-plugin.

In this section, we first present the GOAL tool in a general way (Section 3.1.1). In Section 3.1.2, we give some more details about the plugin architecture of GOAL, and describe some properties of our implementation. Finally, in Section 3.1.3, we describe how we verified the correctness of our implementation.

### 3.1.1 GOAL

GOAL stands for *Graphical Tool for Omega-Automata and Logics* and is being developed since 2007 by the Department of Information Management at the National Taiwan University (NTU)[2]. The tool has been presented in various scientific publications [33][34][35][32]. It is a Java program, and it is freely available on http://goal.im.ntu.edu.tw.

GOAL is an $\omega$-automata manipulation tool. It provides a large number of operations that can be applied to different types of $\omega$-automata. These operations range from input testing, conversions to other types of automata, to union and intersection. Figure 3.1 shows a screenshot of GOAL's graphical user interface with an open menu showing the breadth of operations that GOAL provides. Of course, complementation is also part of these operations, and GOAL includes a some of well-known Büchi complementation constructions, that we will describe below.

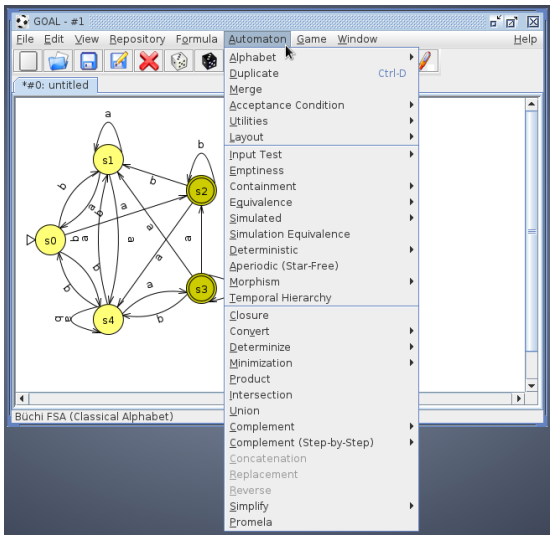



Figure 3.1: The graphical user interface of GOAL (version 2014–11–17). The open menu item gives an idea about the different types of manipulations that GOAL provides for $\omega$-automata.

[1] http://ubelix.unibe.ch
[2] http://exp.management.ntu.edu.tw/en/IM

Automata can be imported to and exported from GOAL in different formats. The default format is the GOAL File Format (GFF), and files of this type have conventionally the extension ".gff".

The main interface of GOAL is a graphical one, as shown in Figure 3.1. However, almost the entire functionality of GOAL is also available through a command line interface. Complementing an automaton can then for example be done with the command `gc complement -m safra -o out.gff in.gff`. This would complement the automaton in the file `in.gff` with the Safra construction, and write the complement in the GOAL File Format to the file `out.gff`. The command `gc` is the GOAL executable for the command line interface. The command line interface is a very important feature that makes GOAL usable for automated batch processing.

GOAL is versioned by version names of the form YYYY–MM–DD that specify the release date. The latest version at the time of this writing is version 2014–11–17. This is the version that our description is based on, and that we used for all our experiments.

What we are most interested in, in the context of this thesis, are of course GOAL's Büchi complementation constructions. The the 2014–11–17 version of GOAL, contains implementations of 10 Büchi complementation constructions that are well-known from the literature. Table 3.1 lists these constructions together with their authors and reference to the literature.

| # | Identifier | Name/description | Authors (year) | Ref. |
|---|---|---|---|---|
| 1 | Ramsey | Ramsey-based construction | Sistla, Vardi, Wolper (1987) | [26] |
| 2 | Safra | Safra construction | Safra (1988) | [24] |
| 3 | ModifiedSafra | Modification by Althoff | Althoff (2006) | [2] |
| 4 | Piterman | Safra-Piterman construction | Piterman (2007) | [20] |
| 5 | MS | Muller-Schupp construction | Muller, Schupp (1995) | [18] |
| 6 | Rank | Rank-based construction | Schewe (2009) | [25] |
| 7 | WAPA | Via weak alternating parity automata | Thomas (1999) | [30] |
| 8 | WAA | Via weak alternating automata | Kupferman, Vardi (2001) | [11] |
| 9 | Slice+P | Slice-based construction (earlier) | Vardi, Wilke (2007) | [39] |
| 10 | Slice | Slice-based construction (later) | Kähler, Wilke (2008) | [8] |

Table 3.1: The pre-implemented NBW complementation constructions in GOAL (version 2014-11-17).

We sorted the construction in Table 3.1 according to the four fundamental complementation approaches, Ramsey-based, determinization-based, rank-based, and slice-based. The first construction, Ramsey, is the only construction belonging to the Ramsey-based complementation approach. The following four constructions, Safra, ModifiedSafra, Piterman, and MS, belong to the determinization-based approach. Rank, WAPA, and WAA belong to the rank-based approach. Finally, Slice and Slice+P belong to the slice-based approach. Throughout the rest of this thesis, when we refer to one of GOAL's Büchi complementation constructions, we will use the identifiers as defined in Table 3.1.

Slice and Slice+P are actually combined in a single construction in GOAL. However, one of the two constructions can be selected by the means of the option P. With the P option, the construction by Vardi and Wilke is used, and without the P option, the one by Kähler and Wilke is used. For our study, we will use Vardi and Wilke's construction (with the P option), however, we will usually still refer to this construction as simply Slice.

At this point, it is worth pointing at a related project of the same research group, called the Büchi Store. This is an online repository of classified and tagged $\omega$-automata that can be downloaded in different formats (including GFF). The Büchi Store is located on http://buchi.im.ntu.edu.tw/ and has also been described in a scientific publication [36]. Furthermore, there is a binding in GOAL to the Büchi Store, so that the contents of the store can be directly accessed from GOAL. For our project we did not make use the Büchi Store, but it is might be an interesting option for related projects.

### 3.1.2 Implementation of the Construction

#### The GOAL Plugin

GOAL has been designed from the ground up to be modular and extensible. To this end, it has been created with the Java Plugin Framework (JPF)[3]. This framework allows to build applications whose functionality can be easily and seamlessly extended by writing additional plugins for it. These plugins can be installed in the main application without the need to recompile the whole application. Rather, the plugin is compiled separately and the resulting bytecode files are copied to the directory tree of the main application. It is not even necessary to know the source code of the main application in order to write a plugin. The interfaces of JPF itself, and the documentations of the relevant classes of the main application are all that a plugin writer needs to know.

In some more detail, JPF requires an application to define so called *extension points*. For any extension point, multiple *extensions* can be provided. These extensions contain the actual functionality of the application. A JPF application basically consists of extensions that are plugged into their corresponding extension points. A plugin is an arbitrary bundle of extensions and extension points. It is the basic unit of organisation in the Java Plugin Framework.

One of the extension points of GOAL is called ComplementConstruction. The extensions to ComplementConstruction contain the actual complementation constructions that GOAL provides. For adding a new complementation construction to GOAL, one has thus to create a new extension to ComplementConstruction. This extension can then be wrapped in a plugin, and the plugin can be compiled and installed in the main application, what makes it an integral part of it. This means that once the plugin is installed, the new construction is included in GOAL in the same way as all the other constructions.

This is how we added the Fribourg construction to GOAL. The name of our plugin is ch.unifr.goal.complement[4]. It is publicly available and can be installed by anybody in their GOAL application. We give instructions on how to get, install, and use the plugin in Appendix A.

In reality, there is more than just the extension point ComplementConstruction that can be extended to add a new complementation construction to GOAL. There are separate extension points for, for example, the command line binding, menu inclusion, or step-by-step execution support. We created extensions to all these extension points as well and included them in our plugin. Our aim was to make the integration of the Fribourg construction in GOAL as complete as possible so that it provides the same facilities as the pre-implemented constructions.

#### The Fribourg Construction Options

In our implementation of the Fribourg construction we also included the three optimisations, R2C, M1, and M2, described in Section 2.4. We implemented these optimisation as user-selectable complementation construction options. In the GUI, these options are presented to the user as a list of checkboxes immediately before the start of each complementation task. In the command line mode, there is a command line flag for each option that can be set or not set by the user.

In addition to the three optimisations, we added further options to our construction. Table 3.2 lists all the available options for the Fribourg construction. Each option has an identifier consisting of upper-case letters that we will use throughout the rest of this thesis to refer to the corresponding options.

The first three options in Table 3.2 represent the three optimisations from Section 2.4. The R2C optimisation is implemented so that it applies only to input automata that are complete. That is, selecting R2C for the complementation of an automaton that is not complete has no effect, and the result is the same as if R2C would not have been selected. The options M1 and M2 implement the M1 and M2 optimisations. Since M2 is dependent on M1, it is not possible to select M2 without also selecting M1. This restriction is enforced in both the GUI and the command line interface.

[3] http://jpf.sourceforge.net/

[4] By convention, JPF plugins are named after the base package name of their implementation files.

| Option | Description |
|---|---|
| R2C | Apply R2C optimisation if input automaton is complete |
| M1 | Apply M1 optimisation (component merging) |
| M2 | Apply M2 optimisation (colour 2 reduction) |
| C | Make input automaton complete before start of construction |
| R | Remove unreachable and dead states from output automaton |
| RR | Remove unreachable and dead states from input automaton |
| MACC | Maximise accepting states of input automaton |
| B | Use the "bracket notation" for state labels |

Table 3.2: The options of the Fribourg construction in GOAL.

The C option is one of the options that modifies the input automaton before the actual complementation starts. This option first checks if the input automaton is complete[5], and if this is not the case, makes it complete by adding a sink state. This means that an additional non-accepting state, the sink state, is added to the automaton, and from every incomplete state the "missing" transitions are added from this state to the sink state. The sink state itself has loop transitions for all symbols of the alphabet.

The purpose of the C option is to be used in conjunction with the R2C optimisation. By making an automaton complete before the start of the construction, we can ensure that the R2C optimisation will be applied. The question then arises whether, in terms of performance, it is worth to do is. Because for making an automaton complete, we have to add an additional state to the automaton what generally increases the complexity of the complementation. This question has been investigated in previous work about the Fribourg construction by Göttel [6]. In this thesis will re-investigate this point in an extended form.

The R option modifies the output automaton at the end of the construction. In particular, it removes all the so called unreachable and dead states from the complement. Unreachable states are states that cannot be reached from the initial state. Dead states are states from which it is not possible to reach an accepting state. These states can be removed from any automaton without changing the language of the automaton. The pre-implemented complementation constructions Ramsey, Piterman, Rank, and Slice also contain a similar R option.

One usage case of the R option is to determine the number of unreachable and dead state that a complementation construction produces. Complementing the same automaton with and without the R option, and taking the difference of the complement sizes will yield this number. Investigations in this direction have been done with GOAL by Tsai et al. [31]. In our own investigations we will use the R option to determine the number of unreachable and dead states the plain Fribourg construction produces.

The RR option is similar to the R option, except that it removes the unreachable and dead states from the input automaton rather than from the output automaton. This option is a custom creation by us, and the pre-implemented complementation constructions in GOAL do not contain a similar option. We are not using the RR option in our investigations.

The MACC option again modifies the input automaton before the start of the construction. Namely, it applies the technique of the "maximisation of the accepting set". This means that as many states as possible are made accepting without changing the language of the automaton. The larger number of accepting state should then simplify the complementation task. This technique has been introduced and empirically investigated by Tsai et al. in [31]. The pre-implemented complementation constructions Ramsey, Piterman, Rank, and Slice contain a similar MACC option. In our own investigations we will however not use the MACC option.

Finally, the B option is a pure display option and does not alter the automata or the construction itself. Its effect is to use the bracket notation for state labels, that indicates component colours by different types of brackets, instead of the default notation that uses numbers to specify the component colours.

[5] An automaton is complete if every state has at least one outgoing transition for every symbol of the alphabet.

### 3.1.3 Verification of the Implementation

Having an implementation, it is important to verify that it is correct. With correct we mean in this section that the construction produces a valid complement for a given input automaton, that is, that the language of the output automaton is indeed the complement of the language of the input automaton. We verified this point of our implementation as described below. In this sense, we know that our implementation is a valid construction. This is not the same as the verification that our implementation faithfully represents the Fribourg construction as it has been devised by its creators. Rather, this latter point has been informally verified during the whole development period of the implementation, which was possible thanks to the close collaboration with the construction creators.

In order to verify that our implementation produces correct results, we performed so called complementation-equivalence tests in GOAL against one of the pre-implemented construction. This works as follows. Take an automaton $A$ and complement it with one of the GOAL constructions. This yields automaton $A'$. Then, complement the same automaton $A$ with our implementation of the Fribourg construction. This yields automaton $A''$. Now, by the means of GOAL's equivalence operation then test whether $A'$ and $A''$ accept the same language.

This approach makes a couple of assumptions. First, it relies on the correctness of the pre-implemented complementation constructions, and on the equivalence operation of GOAL (which is also based on complementation). Second, with this empirical approach it is of course not possible to conclusively verify the correctness of our implementation. Every passed complementation-equivalence test just further confirms the hypothesis that our implementation is correct, but it can never be proved. Despite these points, we think that this approach is the best we can do, and that, with a sufficient number of test cases, it can confirm the correctness of our implementation with a high probability.

We tested the Fribourg construction in different versions with all the options that we described in the last section (except the B option as it does not influence the construction). The tested versions are the following.

- Fribourg
- Fribourg+R2C+C
- Fribourg+M1
- Fribourg+M1+M2
- Fribourg+R
- Fribourg+RR
- Fribourg+MACC

We tested each version with 1,000 randomly generated automata. The automata had a size of 4 and an alphabet size between 2 and 4. The pre-implemented construction we tested against, was the Piterman construction. The computations were executed on the UBELIX computer cluster that is described in Section 3.3.3. The result was that all the tests were successful, that is, there was not a single counterexample.

With a size of 4, the automata used for the tests are rather small, and it would be interesting to do the tests with bigger automata. However, our limited computing and time resources prevented us from doing so. The equivalence test of GOAL is implemented as reciprocal containment tests, which include complementation. That is, the overall test includes the complementation of the complements of the test automata. By using bigger test automata, their complements might be already so big that a further complementation is practically infeasible with our available resources. Nevertheless, we think that the high number of performed tests confirms the correctness of our construction with a high probability.

## 3.2 Test Data

Test data is the sample data that is given to an algorithm as input in order to measure and evaluate certain properties of the output. The test data is an important part of every empirical study and should meet several requirements. It should include enough test cases so that the results are statistically significant.

It should not be biased in favour of the tested algorithm. It should cover test scenarios that are relevant for evaluating the performance of the algorithm. Ideally, publicly available test data is used that has also been used for other empirical studies. In this way, the data is "objective" and results of different studies are comparable.

In our case the test data consists of a set of non-deterministic Büchi automata which are provided as input to the Fribourg construction. We chose two different sets of automata of which we believe that together they form a meaningful test set for our study.

The first set of test automata, which we call GOAL test set, consists of 11,000 NBW of size 15. This test set has been used by a previous empirical study with GOAL by Tsai et al. [31], and is publicly available. The second set of automata, which we call Michel test set, consists of a family of automata that show a very pronounced state growth for complementation. These automata have been introduced and described by Michel [15], hence the name. Below, we describe these two test sets with their relevant properties in separate sections.

### 3.2.1 GOAL Test Set

The GOAL test set is the larger and more complex one of the two test sets. In this section, we first introduce the GOAL test set and describe its basic structure. In the second part, we present the results of an analysis that we did in order to reveal further properties of the GOAL test set, namely the number and distribution of complete, universal, and empty automata.

#### Introduction and Basic Structure

The GOAL test set has been created by Tsai et al. for an empirical study evaluation the effects of several optimisations on existing Büchi complementation constructions [31]. This study has been executed in GOAL and the automata in the test set are available in the GOAL file format, hence the name GOAL test set.

The entire test set consists of 11,000 automata of size 15, and 11,000 similar automata of size 20. For our study we used however only the automata of size 15. Hence, when we refer to the GOAL test set in the rest of this thesis, we specifically mean the 11,000 automata of size 15. The GOAL test set is publicly available through the following link: https://fribourg.s3.amazonaws.com/testset/goal.zip[6].

All the automata in the test set have 15 states and an alphabet size of 2. The further properties of the automata, namely accepting states and transitions, are determined by the values of two parameters called *transition density* and *acceptance density*

The transition density determines the number of transitions of an automaton. In some more detail, the transition density $t$ is defined as follows. Let $n$ be the number of states of automaton $A$, and $t$ its transition density. Then $A$ contains $tn$ transitions for each symbol of the alphabet. In the case that $tn$ is not an integer, it is rounded up to the next integer. That is, if one of our automata with 15 states and the alphabet $0, 1$ has a transition density of 2, then it contains exactly 30 transitions for symbol 0 and 30 transitions for symbol 1.

The acceptance density $a$ is defined as the ratio of accepting states to non-accepting states in the automaton. It is thus a number between 0 and 1. If automaton $A$ has $n$ states and an acceptance density of $a$, then it has $an$ accepting states. In the case that $an$ is not an integer, it is rounded up to the next integer.

The GOAL test set is structured into 110 classes with different transition density/acceptance density pairs. These 110 pairs result from the Cartesian product of 11 transition densities and 10 acceptance

[6]This link is maintained by the author of the thesis. The original link of the GOAL tests set that is maintained by the authors of [31] is http://goal.im.ntu.edu.tw/wiki/lib/exe/fetch.php?media=goal:ciaa2010_automata.tar.gz. This package contains additionally the 11,000 automata of size 20. In the package of the first link, the files have been renamed, however, the content of the files has not been changed.

densities. The concrete transition densities $t'$ and acceptance densities $a'$ are the following:

$$t' = (1.0, 1.2, 1.4, 1.6, 1.8, 2.0, 2.2, 2.4, 2.6, 2.8, 3.0)$$
$$a' = (0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0)$$

Thus, there is one class whose automata have a transition density of 1.0 and an acceptance density of 0.1, another class with a transition density of 1.0 and an acceptance density of 0.2, and so on. Each of the 110 classes contains 100 automata

When Tsai et al. created the GOAL test set, they created the automata within the given constraints of any class at random. They chose this structure and parameter values in order to generate a broad range of complementation problems ranging from easy to hard [31].

#### Further Properties: Completeness, Universality, and Emptiness

We analysed the properties of completeness, universality, and emptiness[7] of the automata in the GOAL test set. To know about these properties is useful for the interpretation of the results of our study. For example, the R2C optimisation applies only to complete automata, thus it is interesting to know how many automata are complete. The smallest possible complements of universal and empty automata have a size of 1. Thus, we can see how many "superfluous" states a construction produces when complementing a universal or empty automaton.

GOAL provides a command for testing emptiness. However, it does not provide commands for testing completeness and universality. We therefore implemented these commands on our own and bundled them as a separate GOAL plugin. The plugin is called ch.unifr.goal.util and also publicly available as described in Appendix A.

With these GOAL commands we tested each of the 11,000 automata for the three properties. On one hand, we want to know how many complete, universal, and empty automata are there in total. On the other hand, we also want to know how these properties are distributed across the 110 classes of transition density/acceptance density combinations. Thus, we also determined the number of complete, universal, and empty automata for each class. The computations were executed on the UBELIX computer cluster that is described in Section 3.3.3.

The resulting overall numbers are the following:

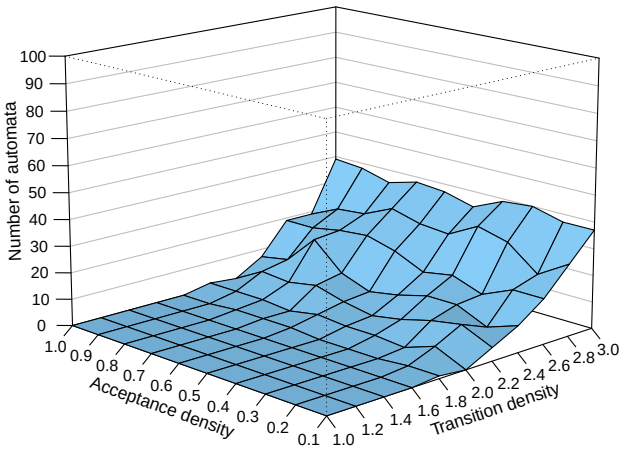
- 990 of the 11,000 automata are complete (9%)
- 6,796 of the 11,000 automata are universal (61.8%)
- 63 of the 11,000 automata are empty (0.6%)

The fact that only 9% of the automata are complete means that the R2C optimisation of the Fribourg construction affects only 9% of the test data. This is an interesting fact for the analysis of the effect the R2C optimisation has on the overall performance of the construction. A surprisingly high number of 61.8% of the automata are universal. A reason might be the small alphabet of the GOAL test set automata, which has just two symbols. With a certain number of transitions in the automata, there seems to be a high probability that the automata are universal. Conversely, the number of empty automata is very low. This can be seen as the reverse of the same effect that causes the number of universal automata to be high.

In Figure 3.2, we show the number of complete, universal, an d empty automata per class. In this figure we introduce by the way two ways for representing per-class data that we will use throughout this thesis. On the left side of Figure 3.2 the per-class data is represented as matrices. These matrices always have 11 rows and 10 columns, and the rows always represent the transition densities and the columns represent the acceptance densities. On the right side of Figure 3.2, the same data is visualised as so called perspective plots. The corner of the perspective plots that is closest to the viewer corresponds to the upper-left corner of the corresponding matrices. Thus, looking at a perspective plots is like looking

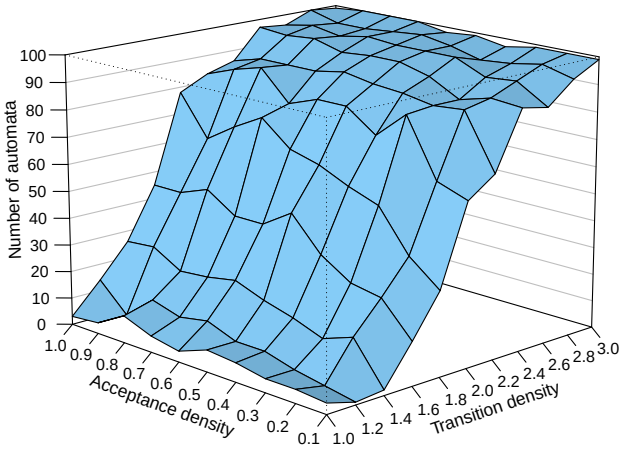
[7] An automaton is *complete* if every state has at least one outgoing transition for every symbol of the alphabet. An automaton is *universal* if it accepts every word that can be generated from its alphabet. An automaton is *empty* if it does not accept anything (except the empty word $\epsilon$).

| | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1.0 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1.0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1.2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1.4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1.6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1.8 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 2.0 | 0 | 5 | 1 | 2 | 2 | 3 | 1 | 2 | 2 | 2 |
| 2.2 | 5 | 10 | 8 | 5 | 3 | 5 | 8 | 6 | 7 | 1 |
| 2.4 | 10 | 6 | 11 | 11 | 8 | 6 | 10 | 20 | 9 | 7 |
| 2.6 | 17 | 17 | 12 | 16 | 14 | 19 | 22 | 21 | 19 | 19 |
| 2.8 | 27 | 20 | 29 | 32 | 26 | 27 | 30 | 25 | 24 | 19 |
| 3.0 | 37 | 37 | 40 | 39 | 34 | 37 | 38 | 35 | 38 | 39 |



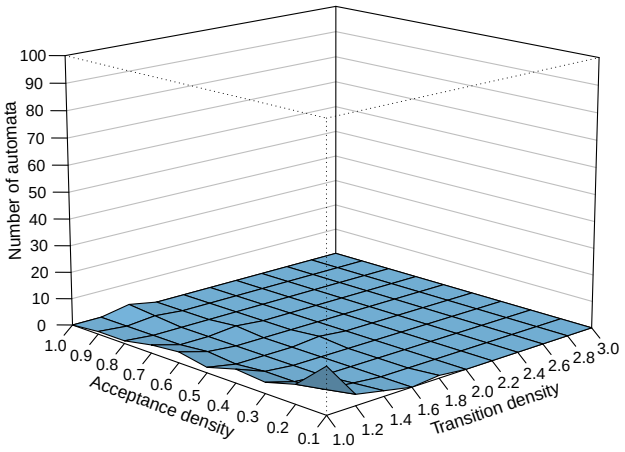

(a) Number of *complete* automata per class.

| | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1.0 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1.0 | 4 | 5 | 5 | 7 | 8 | 4 | 6 | 10 | 4 | 3 |
| 1.2 | 1 | 3 | 5 | 8 | 8 | 12 | 10 | 13 | 4 | 14 |
| 1.4 | 2 | 17 | 13 | 17 | 20 | 24 | 22 | 21 | 27 | 26 |
| 1.6 | 16 | 28 | 30 | 37 | 49 | 42 | 42 | 49 | 45 | 45 |
| 1.8 | 31 | 40 | 55 | 59 | 64 | 67 | 76 | 70 | 63 | 78 |
| 2.0 | 60 | 64 | 85 | 75 | 83 | 83 | 79 | 90 | 87 | 83 |
| 2.2 | 67 | 87 | 86 | 88 | 89 | 91 | 89 | 89 | 89 | 86 |
| 2.4 | 88 | 89 | 86 | 92 | 95 | 95 | 94 | 97 | 96 | 97 |
| 2.6 | 86 | 93 | 92 | 97 | 97 | 97 | 98 | 96 | 98 | 96 |
| 2.8 | 94 | 97 | 95 | 94 | 97 | 99 | 98 | 97 | 97 | 100 |
| 3.0 | 99 | 99 | 99 | 97 | 99 | 98 | 100 | 100 | 100 | 99 |




(b) Number of *universal* automata per class

| | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1.0 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1.0 | 17 | 7 | 4 | 5 | 2 | 4 | 3 | 1 | 1 | 0 |
| 1.2 | 4 | 2 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1.4 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 |
| 1.6 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1.8 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 2.0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2.2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2.4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2.6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2.8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3.0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |




(c) Number of *empty* automata per class

Figure 3.2: Number of complete, universal, and empty automata for each of the 110 classes of transition density/acceptance density combinations. Each class contains 100 automata.

at the corresponding matrix from the upper-left corner. The advantage of matrices is that they show all the data values explicitly. The advantage of perspective plots is that they show the patterns in the data more intuitively. When we present the results of our study in Chapter 4, we will mainly use perspective plots, however, we will give all the corresponding matrices in Appendix B.1.

Regarding the complete automata per class in Figure 3.2 (a), we can see their number increases with the transition density. Up to a transition density of 1.6 there are no complete automata at all, and then it starts to increase up to a number between 34 and 40 for the transition density of 3.0. Since each class contains exactly 100 automata, these numbers are percentages at the same time. That the number of complete automata increases with the transition density is because a higher number of transitions per alphabet symbol in the automaton increases the probability that each state has at least one outgoing transition for each alphabet symbol. For example, with a transition density of 1.0 and 15 states, the automaton contains exactly 15 transitions for each alphabet symbol. It is still possible that this automaton is complete, but the probability is very low, because there must be a one-to-one mapping of transitions and states. On the other hand, with a transition density of 3.0, there would be 45 transitions per alphabet symbol, and the probability that each state gets one of them is much higher.

The number of universal automata per class in Figure 3.2 (b) also increases with the transition density, although much stronger. While in the classes with a transition density of 1.0, there are between 3 and 10 universal automata, in the classes with a transition density of 3.0 there are between 97 and 100. As already mentioned, the small alphabet size of the GOAL test set automata and a sufficiently high number of transitions results in a high probability that an automaton accepts every possible word, and thus is universal. In Figure 3.2 (b) we can also see that low acceptance densities result by trend in slightly fewer universal automata. This is because with fewer accepting states there is less chance that a given word is accepted. As we identified the small alphabet size as a possible reason for the high number of universal automata, it would be interesting to test how many universal automata there are in similar automata with a bigger alphabet size.

Conversely to the high number of universal automata, the number of empty automata is very low. The totally 63 empty automata are mainly concentrated in the upper-left corner of the matrix in Figure 3.2 (c). That is, the automata with a low transition density and a low acceptance density have the highest probability to no accept any word, and thus being empty. The reasons for this are basically the opposite reasons for the distribution of the universal automata.

### 3.2.2 Michel Test Set

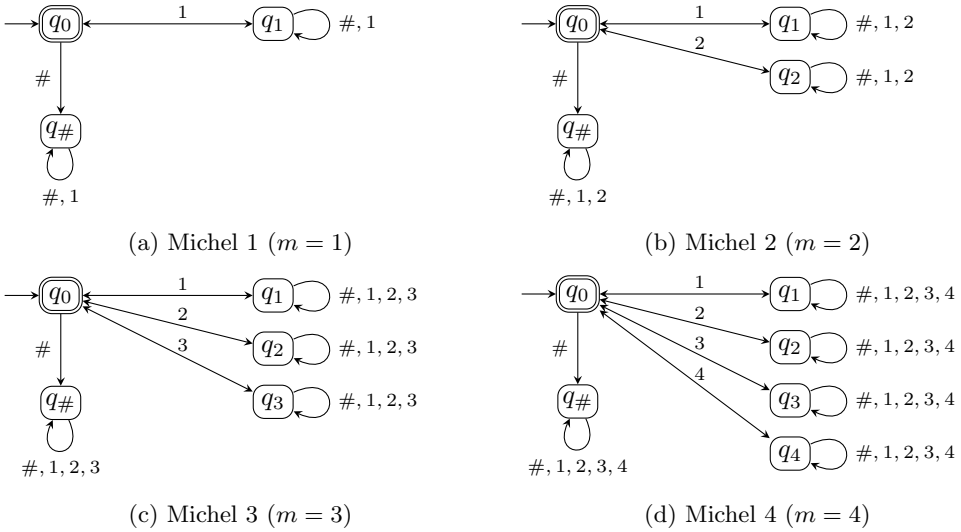
The Michel test set is very different from the GOAL test set. It consists of a family of automata, the Michel automata, which exhibit an especially heavy state growth for complementation.

Michel automata have been introduced in 1988 by Max Michel in order to prove a lower bound for the state growth of Büchi complementation of $(n-2)!$, where $n$ is the number of states of the input automaton [15][29]. Michel constructed a family of automata, characterised by the parameter $m$, that have $m+1$ alphabet symbols, and $m+2$ states. He proved that the complements of these automata cannot have less than $m!$ states. Since the number of states of the input automata is $n = m+2$, the state growth in terms of input and output states is $(n-2)!$, which is around $(0.36n)^n$.

The state growth of Michel automata is so heavy that for practical reasons we are restricted to include only the first four Michel automata, that is the ones with $m = \{1, \ldots, 4\}$, in our test set. For the Michel automata with $m \geq 5$, the required time and computing power for complementing them with our implementation would by far exceed our available resources. We present some extrapolations in this direction in Section 4.1.2. Despite the small number of automata in the test set, we can still obtain very interesting results from them, as we will see in Chapter 4.

The four Michel automata in our test set are shown in Figure 3.3. We will call them Michel 1, Michel 2, Michel 3, and Michel 4, respectively. As mentioned, Michel automata have $m+2$ states and an alphabet size of $m+1$. Furthermore, they all have a single accepting state. Our Michel automata 1 to 4 have thus 3, 4, 5, and 6 states, and alphabet sizes of 2, 3, 4, and 5, respectively.

The interesting thing about the high complementation state-complexity of Michel automata is that they allow to "elicit" large number of states from the complementation constructions that we investigate in our study. In the theoretical approach to Büchi complementation, the main performance metric is the worst-case state complexity. This is the maximum number of states that a construction can produce, in function of the number of states of the input automaton. If for example a construction has a worst-case




(a) Michel 1 ($m = 1$)

(b) Michel 2 ($m = 2$)

(c) Michel 3 ($m = 3$)

(d) Michel 4 ($m = 4$)

Figure 3.3: The Michel automata with $m = \{1, \ldots, 4\}$, an alphabet size of $m + 1$, and $m + 2$ states.

complexity of $(0.76n)^n$, then, if the input automaton has size $n$, the maximum size of the complement is $(0.76n)^n$.

Thus, if with the complementation of a Michel automata we measure a certain state growth for one of the constructions, say $(0.99n)^n$, then we can deduce that the worst-case complexity of this construction must be greater than or equal to $(0.99n)^n$. In this way, we can get an idea about the lower bounds for the worst-case complexities of our investigated constructions.

This concludes the presentation of the test data for our study. In the next section we are describe the experimental setup in which this test data is used.

## 3.3 Experimental Setup

In this section we describe the concrete experiments that we executed, including the allocated resources and imposed constraints. As mentioned, the experiments are divided into the internal tests and external tests. In the internal tests we compare different versions of the Fribourg construction with each other. In the external tests, we compare one (the most performant) version of the Fribourg construction with other well-known complementation constructions.

The internal and external test, are done with both, the GOAL and the Michel test set. Thus, there are four groups of experiments: internal–GOAL, internal–Michel, external–GOAL, and external–Michel.

In Section 3.3.1, we present the versions of the Fribourg construction used for the internal tests. These versions differ for the GOALand the Michel test set, and we present them separately. In Section **??**, we present the version of the Fribourg construction used for the external tests (which is also different for the GOALand the Michel test set), and the concrete versions of the third-party construction against which we compare the Fribourg construction. In Section 3.3.3, we describe the computing environment in which the experiments were executed. Finally, in Section 3.3.4, we present the time and memory limits that were imposed on the experiments.

### 3.3.1 Constructions for the Internal Tests

The versions of the Fribourg construction used for the internal tests consist of combinations of the three optimisations R2C, M1, and M2, and of the additional options C and R (see list of options for the Fribourg construction in Table 3.2). The sets of versions are different for the two test sets. Our aim in

choosing specific versions was to find the most performant version of the Fribourg construction for each test set.

### GOAL Test Set

For the internal test with GOAL test set, we use the following eight versions of the Fribourg construction:

1. Fribourg
2. Fribourg+R2C
3. Fribourg+R2C+C
4. Fribourg+M1
5. Fribourg+M1+M2
6. Fribourg+M1+R2C
7. Fribourg+M1+R2C+C
8. Fribourg+R

Version 1 is the plain Fribourg construction without any optimisations or options. Version 2 and 3 aim at investigating the R2C optimisation. In Version 2, the R2C optimisation is applied only to complete input automata, and as we have seen in Section 3.2.1 these are just 9% of the automata. Version 3, on the other hand, makes all input automata complete so that the R2C optimisation can be applied to all automata. The question is whether it is worth to increase the size of an automaton by one (for adding the sink state) and then being able to apply the R2C optimisation, or not.

A very similar question has been investigated in previous work about the Fribourg construction by Göttel [6]. In terms of our above listing, he compared Version 1 with Version 3, by also using the GOAL test set as the test data. His result was that the overall mean complement size of Version 3 is higher than for Version 1. By looking closely at his results we suppose however that the median complement size (which was not recorded by Göttel) might be lower for Version 3 than for Version 1. This would be an interesting relation, and therefore, we decided to reinvestigate this question.

Versions 4 and 5 aim at investigating the M1 and M2 optimisations. As M2 can only be applied together with M1, there are only these two possible combinations. As we will see in Chapter 4, Version 4 shows a better performance for the GOAL test set than Version 5. Therefore, we do not further investigate any versions containing Fribourg+M1+M2. However, in Version 6, we further improve Version 4 by adding R2C to Fribourg+M1. In Version 7, we replace R2C by the R2C+C variant. Finally, Version 8 is the same as Version 1, but all unreachable and dead states are removed from the output automaton. This allows to determine the number of unreachable an dead states that have been produced by Version 1.

Versions 6 and 7 then enhance the "better" one of Version 4 and 5 with R2C and its alternative R2C+C. As we will see in Chapter 4, the better one of Version 4 and 5 in terms of median complement sizes is Version 4. That is, the application of M2 results in a decline, rather than a gain, in performance compared to the application of M1 alone. We have to note at this point that such results are always specific to the used the test set, and not universally valid. With a different test set, Version 5 might indeed be better than Version 4. As we will see in the next section, this is the case for our alternative test set consisting of the first four Michel automata.

Version 8, finally, is again the plain Fribourg construction, but this time the output automata are reduced by removing their unreachable and dead states. Comparing the results of Version 8 with Version 1 gives an idea of how many unreachable and dead states the Fribourg construction produces. This is inspired by the paper of the GOAL authors [31] in which the number of unreachable and dead states is one of the main metrics for assessing the performance of a construction.

### Michel Test Set

For the internal tests with the Michel test set, we use the following six versions of the Fribourg construction:

1. Fribourg
2. Fribourg+R2C
3. Fribourg+M1
4. Fribourg+M1+M2
5. Fribourg+M1+M2+R2C
6. Fribourg+R

The reasons for selecting these versions is basically the same as for the GOAL test set. However, there are the following differences. First, the Michel automata are complete, thus there is no need to include the C option. Second, for the Michel test set, Fribourg+M1+M2 is more performant than Fribourg+M1. For the GOAL test set, the contrary is the case. This is why in Version 5 we add R2C to Fribourg+M1+M2 rather than to Fribourg+M1, because, as mentioned, our aim is to identify the most performant version for each test set.

### 3.3.2 Constructions for the External Tests

The constructions used for the external tests consist of the most performant version of the Fribourg construction for each test set, and a fixed set of third-party constructions that are implemented in GOAL.

Regarding the third-party constructions, theoretically all the constructions listed in Table **??** could be used. However, practical reasons prevent us from doing so. In preliminary tests we observed that most of these constructions are very inefficient, or inefficiently implemented, for the automata in the GOAL test set. Using these constructions for our external tests would cause the required memory and time resources to be prohibitively high. According to our tests, this excludes all but the Piterman, Slice, Rank, and Safra constructions from being used. A similar experience has been made by Tsai et al. in their own empirical study with GOAL [31]. They observed that the Ramsey construction could not complete the complementation of any automata in the GOAL test set within the time limit of 10 minutes and memory limit of 1 GB.

Considering these restrictions, we decided to include only the Piterman, Slice, and Rank construction in our external tests. These constructions are furthermore the main representative of three of the four main complementation approaches, determinization-based, rank-based, and slice-based. The fourth approach would be Ramsey-based, but as mentioned, the Ramsey construction in GOAL is not efficient enough. It would have been possible to also include the Safra construction, but as it belongs to the determinization-based approach and we already have the Piterman construction, we decided to not include it.

For the Slice construction, we chose the Slice+P version (see Table **??**) by Vardi and Wilke [39]. According to Tsai et al. [31] this version has a lower worst-case complexity than the alternative Slice version by Kähler and Wilke [8].

The Piterman, Rank, and Slice constructions also have a bunch of options in GOAL(for a complete list of their options it is best to consult the help page for the complement command in the command line interface of GOAL[8]). For each construction we included those options that are set by default in the GOAL GUI, except the MACC and R options. The reason to exclude these options is that they are not part of the actual construction, but they just modify the input and output automata, respectively.

We made an exception for the Piterman construction where we also excluded the SIM option. The reason for this is that the SIM option simplifies the intermediate NPW of the Piterman construction, which can also be seen as a modification of an (intermediate) output automaton.

Altogether, this gives the following three constructions that we used for the external tests:

1. Piterman+EQ+RO
2. Slice+P+RO+MADJ+EG
3. Rank+TR+RO

[8]Type gc help complement.

Regarding the Fribourg construction, we chose the most performant version for each test set. These versions are:

1. Fribourg+M1+R2C for the GOAL test set
2. Fribourg+M1+M2+R2C for the Michel test set

### 3.3.3 Execution Environment

As mentioned, we ran all the experiments on the high performance computing (HPC) computer cluster UBELIX of the University of Bern[9]. UBELIX consists of different types of computers (called *nodes*) on which the tasks (called *jobs*) of the cluster users are run.

We ensured that all our experiments run on similar nodes. These nodes have the following specifications:

- Processor: Intel Xeon E5-2665 2.40GHz
- Architecture: 64 bit
- CPUs (cores): 16
- Memory (RAM): 64 GB or 256 GB
- Operating System: Red Hat Enterprise Linux 6.6
- Java platform: OpenJDK Java 6u34
- Shell: GNU Bash 4.1.2

The experiments on the GOAL test set were run on nodes with 64 GB RAM, the experiments on the Michel test set were run on special high-memory nodes with 256 GB RAM. Apart from that, the specifications of these nodes are identical. The use of the high-memory nodes for the Michel test set was required, because the maximally allocatable memory per CPU core of the nodes with 64 GB memory is 4 GB, and this was not enough to complement the Michel automata. With the high-memory nodes, on the other hand, a total of 16 GB can be allocated per CPU core which was sufficient to complement all the Michel automata.

Regarding multicore usage, the behaviour of our experiments depends on GOAL, and thus ultimately on Java. GOAL is programmed multi-threaded and thus uses multiple CPUs. Theoretically, our tasks can use up to the total number of 16 CPUs of a node. However, we observed that our tasks typically used 2–4 CPUs[10].

We also measured the execution time of each complementation task as CPU time and real time (also known as wallclock time). These measurements were done with the `time` reserved word of Bash. The CPU time is the time a process is actually executed by the CPU. The real time is the time that passes from the start of a process until its termination, and thus includes the time the process is not executed by the CPU (idle time). If a process runs on multiple CPUs, the CPU time is counted on each CPU separately and finally summed up. This means that for multicore execution (as in our case), the CPU time may be higher than the real time. For single-core execution this is not possible, and the CPU time can only be equal to or lower than the real time. In the analysis of our results, we sometimes present statistics of the execution times. These times are always *CPU times*.

The complementation tasks are executed sequentially via the command line interface of GOAL. For each complementation task the GOAL application is started separately, which includes the loading of the Java Virtual Machine (JVM). The JVM startup time is thus included in the measured execution times. According to our observations, this JVM startup time is a constant of approximately two CPU time seconds.

The cluster itself is managed by Oracle Grid Engine (formerly known as Sun Grid Engine) version 6.2[11]. This is a load scheduler that automatically dispatches incoming jobs from cluster users to nodes that have enough free resources and capacity.

[9] http://ubelix.unibe.ch
[10] We can just indirectly guess this number by comparing the measured CPU times and real times.
[11] http://www.oracle.com/us/products/tools/oracle-grid-engine-075549.html

A computer cluster is a multi-user environment and a node can be used by multiple users at the same time. Thus, the total load of a node may vary, depending on number and intensity of other users' jobs. Our tasks were also subject to varying load nodes. We do not know whether this has an influence on our experiments, especially on the measurement of the execution times. We observed variations in the measured execution time (CPU time) for similar tasks. This would also influence the time limit that we describe in the next section. For the moment, we leave further investigations on this topic for future work.[12]

### 3.3.4 Time and Memory Limits

We imposed a time and memory limit on each complementation task of the GOAL test set. For the Michel test set, we did not set any limits. These limits are inspired by the ones that have been used by Tsai and colleagues for their own complementation experiments with GOAL [31]. The time limit is 600 seconds CPU time, and the memory limit is 1 GB Java heap. This means that if the complementation of an automaton is not finished after 600 seconds CPU time, or uses more than 1 GB Java heap, then the task is aborted.

These limits are necessary because of our limited time and computing resources. Of course, the ideal case would be to let every complementation task run to completion, no matter how long it takes and how much memory it uses. However, because of the extreme complexity of the complementation of *some* Büchi automata[13], some few extreme cases may cause practical problems in the experiments. The study is for example ultimately limited by the physically available memory on the nodes, and the maximum running time of a job. Wit these limits we can thus cut off such extreme cases and keep the required resources for the study in affordable bounds.

We implemented the time limit by the means of the ulimit Bash builtin, which allows to set a maximum running times for processes. After this time limit, processes are aborted by the operating system.

The memory limit, as mentioned, defines the maximum size of the Java heap. The heap is the main memory area of the Java process. It is where all the objects that are created by the Java program are stored. Concretely, this means that the states of the complements that are computed by our constructions are stored on the heap. We set the maximum Java heap size with the Xmx option to the Java Virtual Machine[14]. We even set the initial size of the Java heap to 1 GB by the means of the Xms option, so that the heap does not need to be enlarged for any task.

after which running processes are killed. The memory limit was implemented by setting the maximum size of the Java heap, which can be done by the -Xmx option to the Java Virtual Machine (JVM). The heap is the main memory area of Java and the place where all the objects reside. Note that since our memory limit defines actually the size of the Java heap, the total amount of memory used by the process is higher than our limit, as Java has some other memory areas, for example for the JVM itself. However, this is a rather constant amount of memory and independent from the current automaton, so it does not disturb the relative comparisons of the results.

The presence of time and memory limits, and thus aborted complementation tasks, require the introduction of the so called *effective samples* in the result analysis, as introduced in the experiment paper of the GOAL authors. The effective samples are those automata which have been successfully completed by *all* constructions that are to be compared to each other. Imagine two constructions $A$ and $B$ where $A$ is successful complementing all the automata, whereas $B$ has timeouts or memory excesses at 100 of the automata. If we would now take, for example, the median complement sizes of the two result sets without first extracting the effective samples, then $B$ is likely be assessed as too good relative to $A$, because $B$'s results do not include the 100 automata at which it failed, and which are thus likely to have large complement sizes with $B$. The same 100 automata would however be included in the results

[12] Theoretically, each job has the requested CPUs of a node for itself alone, what would mean that the used CPUs are not under varying loads. However, jobs are not prevented from using more than the requested number of CPUs on the same node, what means that we have no guarantee that there are no other job-processes running on the CPUs we are using for our own job.

[13] As we will see in Chapter 4, the distribution of the complexity of the tested Büchi automata is extremely right-skewed, that is, most are easy, and very few are hard.

[14] Usage: java -Xmx1G

of $A$. Therefore, all the result analysis of the experiments with the GOAL test sets, that we present in Chapter 4, are based on the effective samples of the result sets.

This concludes the present chapter that described the setup of our empirical performance investigation of the Fribourg construction. First, we covered how we implemented the Fribourg construction as a part of the existing $\omega$-automata tool GOAL. Then, we presented the test data, consisting of the GOAL test set and the Michel test set, that we use to test the Fribourg construction. Finally, with the experimental setup, we defined which construction versions we plan to run with which test data, and under which constraints. The next chapter is entirely dedicated to the presentation and discussion of the results of these experiments.

# Chapter 4

# Results and Discussion

## Contents



In this chapter we present and discuss the results of our empirical performance study of the Fribourg construction. The presentation of the results is structured along the two sub-studies of the internal tests and the external tests.

Section 4.1 presents the results of the internal tests, and Section 4.2 presents the results of the external tests. Both sections have two subsections. The first one for the results of the GOAL test set, and the second one for the results of the Michel test set.

In Section 4.3 we summarise and discuss the most important results and insights gained from the study. Finally, in Section 4.4, we identify the limitations of our study.

## 4.1 Internal Tests

For the internal tests we tested different versions of the Fribourg construction with both, the GOAL test set and the Michel test set.

For the GOAL test set, the tested versions are (see Section 3.3.1):

1. Fribourg
2. Fribourg+R2C
3. Fribourg+R2C+C
4. Fribourg+M1
5. Fribourg+M1+M2
6. Fribourg+M1+R2C
7. Fribourg+M1+R2C+C
8. Fribourg+R

For the Michel test set, the tested versions are (again, see Section 3.3.1):

1. Fribourg
2. Fribourg+R2C
3. Fribourg+M1
4. Fribourg+M1+M2
5. Fribourg+M1+M2+R2C
6. Fribourg+R

Below, we present the results for the GOAL and the Michel test set in separate sections.

### 4.1.1 GOAL Test Set

#### Overall

Before analysing the actual results, let us see how many aborted complementations tasks there are, so that we can determine the effective samples, which is the set of results that we will actually analyse. Table 4.1 shows the number of timeouts and memory excesses for each of the eight tested versions of the Fribourg construction.

| Construction | Timeouts | Memory excesses |
|---|---|---|
| Fribourg | 48 | 0 |
| Fribourg+R2C | 30 | 0 |
| Fribourg+R2C+C | 54 | 0 |
| Fribourg+M1 | 2 | 0 |
| Fribourg+M1+M2 | 1 | 0 |
| Fribourg+M1+R2C | 1 | 0 |
| Fribourg+M1+R2C+C | 8 | 0 |
| Fribourg+R | 48 | 0 |

Table 4.1: Number of timeouts and memory excesses in the internal tests with the GOAL test set.

As we can see in Table 4.1, there are no memory excesses at all. That is, none of the 11,000 complementation tasks needed more than 1 GB Java heap memory. However, there is quite a number of timeouts. The versions Fribourg, Fribourg+R2C, Fribourg+R2C+C, and Fribourg+R all have 30 or more timeouts. All the other versions, which are the ones containing the M1 optimisation, have only 8 or less timeouts.

If we determine the effective samples from these results, we get a number of 10,939 automata. That is, 61 automata (0.55%) are excluded from the effective samples, because their complementation has been aborted for at least one of the versions.

The entire remaining result analysis in this section will be based on these 10,939 effective samples. Our main interest are the sizes of the complements of these 10,939 automata. To get a first impression, we plot all these complement sizes as a stripchart in Figure 4.1. Each strip contains a dot for each of the 10,939 automata that indicates its complement size. Thus, each strip in Figure 4.1 contains exactly 10,939 dots.

One thing to note is that the distribution of complement sizes is right-skewed (also known as positive-skewed). That means, there is a long tail towards the right along the $x$-axis. The peak seems to be close to the left end of the $x$-axis. This means that most of the complements are small and there are fewer and fewer complements with bigger sizes. A right-skewed distribution implies that the mean is generally higher than the median. This is because the mean is "dragged" to the right by the few large values.

The most interesting thing in Figure 4.1 is however to compare the distributions of the different versions with each other, especially the tail sizes. Going from top to bottom, the distributions of Fribourg and Fribourg+R2C have similarly long tails. Fribourg+R2C+C, however, has a considerably longer tail. This shows us that the C option has a significant effect on the complement sizes, because it adds an additional state to the automata which are not complete. As we have seen in Section 3.2.1, only 9% of the automata

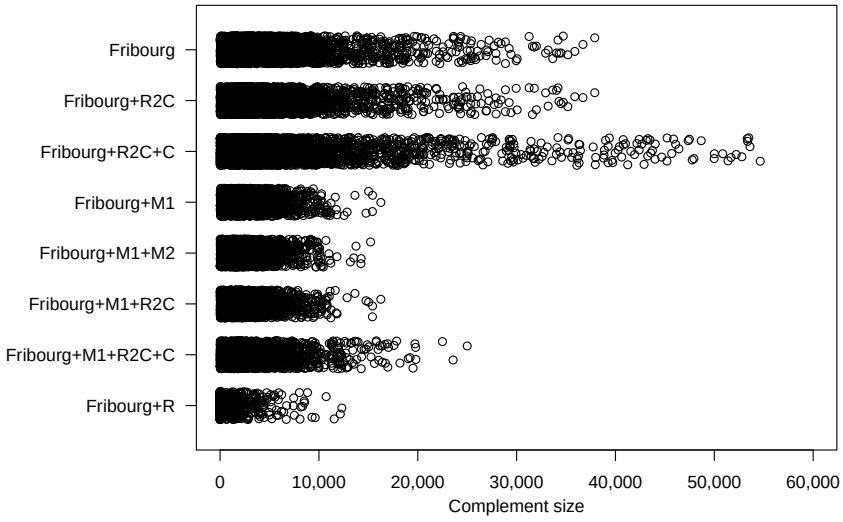



Figure 4.1: Stripchart with the complement sizes of the 10,939 effective samples of the GOAL test set.

of the GOAL test set are complete, thus 91% of the automata are enlarged in this way. This might be the cause for the bigger number of larger complements.

Next, Fribourg+M1, Fribourg+M1+M2, and Fribourg+M1+R2C all have similarly long tails. However, these tails are significantly shorter than the ones of the previous three versions. This indicates that the M1 optimisation is very effective in reducing the complement sizes. The distribution of Fribourg+M1+R2C+C again has a longer tail than the corresponding version without the C option, as we just discussed above.

Finally, the distribution of Fribourg+R has a very short tail. The Fribourg+R version is a special case, because it is essentially the Fribourg version where all the unreachable and dead states are removed from the produced complements. Thus, if we compare the results of Fribourg and Fribourg+R, then we get an idea of how man unreachable and dead states are produced by the Fribourg version.

The stripchart in Figure 4.1 gave us a first impression about the resulting complement sizes. However, for further analysis, we need statistics. In Table 4.2 we show such statistics about the complement sizes. They consist of the mean together with the classic five-number summary, consisting of minimum value, 25th percentile, median, 75th percentile and maximum value.

| Construction | Mean | Min. | P25 | Median | P75 | Max. |
|---|---|---|---|---|---|---|
| Fribourg | 2,004.6 | 2 | 222.0 | 761.0 | 2,175.0 | 37,904 |
| Fribourg+R2C | 1,955.9 | 2 | 180.0 | 689.0 | 2,127.5 | 37,904 |
| Fribourg+R2C+C | 2,424.6 | 2 | 85.0 | 451.0 | 2,329.0 | 54,648 |
| Fribourg+M1 | 963.2 | 2 | 177.0 | 482.0 | 1,138.0 | 16,260 |
| Fribourg+M1+M2 | 958.0 | 2 | 181.0 | 496.0 | 1,156.5 | 15,223 |
| Fribourg+M1+R2C | 937.7 | 2 | 152.0 | 447.0 | 1,118.0 | 16,260 |
| Fribourg+M1+R2C+C | 1,062.6 | 2 | 83.0 | 331.0 | 1,208.5 | 25,002 |
| Fribourg+R | 136.3 | 1 | 1.0 | 1.0 | 21.0 | 12,312 |

Table 4.2: Statistics of the complement sizes of the 10,939 effective samples of the GOAL test set.

As we can see, the mean is indeed throughout higher than the median, which is typical for right-skewed distributions. Regarding the characteristics of the median and the mean, the median is generally the more "robust" statistics, because it is not affected by the actual values of the data points at both sides of the median point. The mean, on the other hand, is a function of all the values in the distribution and thus may be affected by, for example, extraordinarily high values of outliers (as in our case). For our analysis, we will therefore mostly use the median. However, we will sometimes refer to the mean too, as

well as for the other statistics in Table 4.2.

If we go through the median values in Table 4.2 we encounter some surprises. To begin with, as expected, there is a decrease from Fribourg (761) to Fribourg+R2C (689). Then, however, there is a significant drop to 451 with Fribourg+R2C+C. This is a surprise insofar as by looking at Figure 4.1, Fribourg+R2C+C seems to have the worst performance at a first glance. Indeed it also has the highest mean, which is due to the group of extremely large complements. The median, however, is very low, even lower than the one of Fribourg+M1 with its significantly shorter tail in Figure 4.1. Also the 25th percentile of Fribourg+R2C+C is with 85 one of the lowest. Going to the other side of the median, however, the 75th percentile (2,329) is the largest of all versions. A possible characterisation of this phenomenon is that the C option (together with R2C) makes small complements smaller, and large complements larger. The diminishment of small complements is far-reaching enough that the median is affected by it and decreased significantly.

The next thing we see in Table 4.2 is that the median of Fribourg+M1 (482) is slightly lower than the median of Fribourg+M1+M2 (496). The same applies to the 25th and 75th percentile. This means that the additional application of the M2 optimisation to Fribourg+M1 *decreases* the performance of the construction on the GOAL test set. The difference is rather small (the median increase from Fribourg+M1 to Fribourg+M1+M2 is 2.9%). However, it is still enough for us to consider Fribourg+M1 as the more performant of the two versions.

Fribourg+M1+R2C brings down the median from 482 to 447, with respect to Fribourg+M1. Also the 25th and 75th percentile are decreased. Adding the C option to Fribourg+M1+R2C, again causes the median to drop dramatically, from 447 to 331. The 25th percentile decreases from 152 to 83. The 75th percentile however increases from 1,118 to 1,208.5. Here we have again the same picture of the effect of adding the C option that we had before. Namely that small complements are made smaller, and large complements are made larger.

Finally, the last row in Table 4.2 with Fribourg+R shows the extent of unreachable and dead states that the Fribourg version produces. The median is 1, and a further analysis reveals that also the 61st percentile is 1. Only from the 62nd percentile onwards the complement sizes start to increase. This means that 61% of the complements have a size of 1, if we remove all the unreachable and dead states This is not so surprising, because we know from Section 3.2.1 that 61.8% of the automata in the GOAL test set are universal, which means that their complements may contain only a single state.

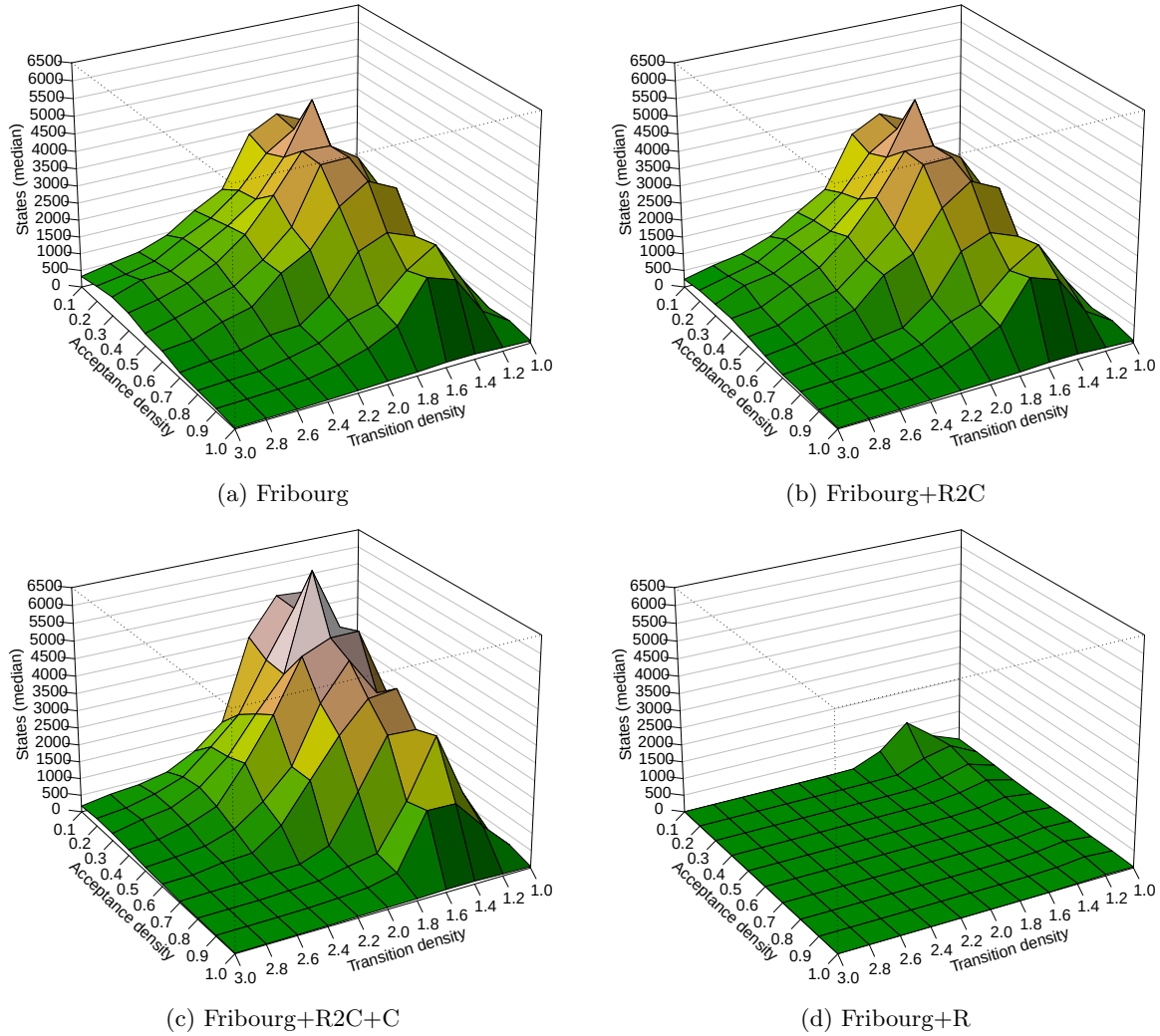
#### Per-Class

Up to now, we only looked at statistics that are aggregated over the entire test set. This mixes together all the automata of the 110 transition density/acceptance density classes of the GOAL test set with their very different characteristics (see the description of the GOAL test set in Section 3.2.1). However, it would be interesting know more about the complement sizes of specific classes, so that we can for example identify easy and hard automata. Therefore, in this part of the analysis, we look at statistics on a per-class basis. This means that for each construction version we will have not just one value per statistics (for example, one median), but 110 values, namely one for each class (for example, 110 medians). Given this complexity, we will restrict ourselves to the median statistics, which we already identified as the most significant and robust statistics. Thus, in the following, we will analyse the median complement sizes of the 110 classes of the GOAL test set.

These per-class statistics result in a similar type of per-class data that we had when we analysed the number of complete, universal, and empty automata in the 110 classes of the GOAL test set in Section 3.2.1. There, we presented this data in two forms, as matrices and as perspective plots. The advantage of matrices is that they show all values unambiguously, the advantage of perspective plots is that they show the relative differences between classes and the overall pattern more intuitively. In this chapter we will only use perspective plots to present this data. This is mainly for space reasons. However, we present the corresponding matrices of all the perspective plots of this chapter in Appendix B.1.

Figures 4.2 and 4.3 show the perspective plots for the eight tested Fribourg construction versions. The spatial orientation of these perspective plots corresponds to looking at a matrix from the lower-right

corner. That is, the corner of the perspective plots that is closest to the viewer corresponds to the lower-right corner of the corresponding $11 \times 10$ matrices. This orientation will be the same for all the remaining perspective plots in this thesis. The surface colours of the perspective plots are in function of the vertical "height" of the surface. They are chosen to draw an analogy with physical terrain. In this sense, we will often talk of "mountains", "hills", and "flatland" in the perspective plots.




(a) Fribourg

(b) Fribourg+R2C

(c) Fribourg+R2C+C

(d) Fribourg+R

Figure 4.2: Median complement sizes of the 10,939 effective samples from the GOAL test set for each of the 110 classes.

The most apparent information that the perspective plots convey is that there are indeed large differences in the complement sizes across the 110 classes. In all the construction versions there is a mountain, or at least a hill, very roughly in the area between transition densities of 1.2 and 2.4, and acceptance densities of 0.1 and 0.9. The mountain is oblong, and its ridge runs across the entire spectrum of the acceptance densities. The top of the ridge is roughly at a transition density of 1.6. On the higher end of the acceptance density spectrum (acceptance density 1.0) the mountain is flattened to a height close to zero. On the other end of the acceptance density spectrum (acceptance density 0.1), however, the mountain stays high.

Considering these median complement sizes in the perspective plots, apparently the automata of, for example, the class with a transition density of 1.6 and an acceptance density of 0.3 result in much larger complements than the automata of, for example, the class with a transition density of 3.0 and an acceptance density of 1.0. We could say that the automata of the first class are harder than the automata of the second class. Later in this section, we will try to identify hard, medium, and easy classes. For now, we will however focus on the relative differences between the different versions of the Fribourg

construction.

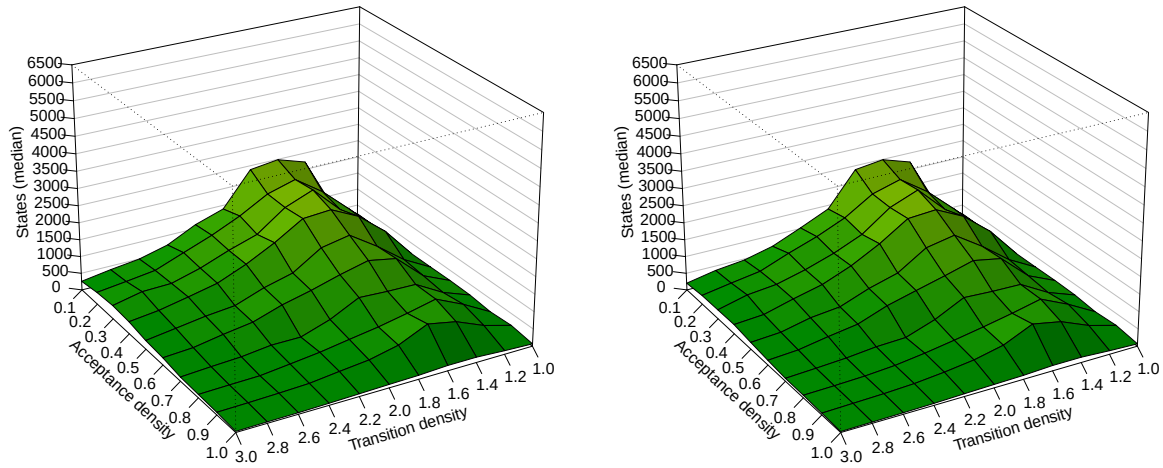
Looking at the perspective plots in Figure 4.2, the plots for Fribourg and Fribourg+R2C are rather similar. The top of the mountain ridge is between 3,500 and 4,000 states with a single peak of around 4900 states in the class with transition density 1.6 and acceptance density 0.3. From Table 4.2 we can learn that the overall median complement size is 761 for Fribourg and 689 for Fribourg+R2C. These low values might surprise at first as the mountain, which is much higher, seems to dominate. However, by taking a closer look, it becomes apparent that around half of the classes are in rather low terrain (less than 1,000 states). Furthermore, the heights of the mountain peak do not allow to deduce anything about the overall median, because the median is not affected by the actual values of the data points which are greater than the median. The overall mean complement sizes of Fribourg and Fribourg+R2C in turn are 2,004.6 and 1955.9, respectively.

Fribourg+R2C+C in Figure 4.2 (c) has an even higher mountain than the Fribourg and Fribourg+R2C. The top of the ridge is at around 5,000 states and the peak at the class 1.6/3.0 has close to 6,500 states. As already in the stripchart in Figure 4.1, Fribourg+R2C+C seems much worse than Fribourg+R2C at a first glance. However, as we have seen in Table 4.2, the median of Fribourg+R2C+C is 34.5% lower than the median of Fribourg+R2C (689 to 451). By taking a closer look at the perspective plots of Fribourg+R2C and Fribourg+R2C+C, the reason for this can be seen. The low areas of Fribourg+R2C+C are slightly lower than the low areas of Fribourg+R2C. This is apparently enough to decrease the overall median. The much higher mountain peaks of Fribourg+R2C+C, on the other hand, do not influence the median. However, they show their effect in the overall mean which for Fribourg+R2C+C is 24% higher than for Fribourg+R2C (2,424.6 to 1,955.9).

Comparing the fourth plot in Figure 4.2, Fribourg+R, to the plots of Fribourg, Fribourg+R2C, and Fribourg+R2C+C is like comparing a Dutch polder to the Swiss Alps. The mountain shrinks to a small hillock and the rest of the terrain is low and flat. This is because so many complements of the Fribourg construction can be reduced to very small sizes by removing their unreachable and dead states. The corresponding matrix in Appendix B.1 reveals that 68 of the 110 classes have a median complement size of 1. If we further compare this matrix to the matrix with the number of universal automata in Figure 3.2 (b) in Section 3.2.1, we see that all the classes with a median of 1 contain more than 50 universal automata, and the classes with a median greater than 1 contain less than 50 universal automata. There is a total of 100 automata per class. This makes sense as the complements of universal automata are empty automata, and every empty automaton can be reduced to an automaton with a single non-accepting state. Looking at the classes with a median greater than 1, we see that their values are still considerably lower than the ones of the plain Fribourg construction.

Figure 4.3 shows the perspective plots of the remaining four versions of the Fribourg construction, all of which include the M1 optimisation. Most apparent in these plots is that the mountain that we described for the plots of Fribourg, Fribourg+R2C, and Fribourg+R2C+C is still there, but it is rather a hill than a mountain. For Fribourg+M1, and Fribourg+M1+R2C, the height of the ridge is around 2,500 states. This is reflected by the overall means of these two versions compared to their counterparts without the M1 optimisation, Fribourg, and Fribourg+R2C. The decrease of the overall mean from Fribourg to Fribourg+M1 is by 52% (from 2004.6 to 963.2) and from Fribourg+R2C to Fribourg+M1+R2C by 52.1% (from 1955.9 to 937.7). The decreases of the overall medians are by 36.6% (from 761 to 482), and 35.1% (from 689 to 447) for the same two pairs of versions. With this we can confirm that the M1 optimisation brings a significant performance gain for the automata in the GOAL test set.

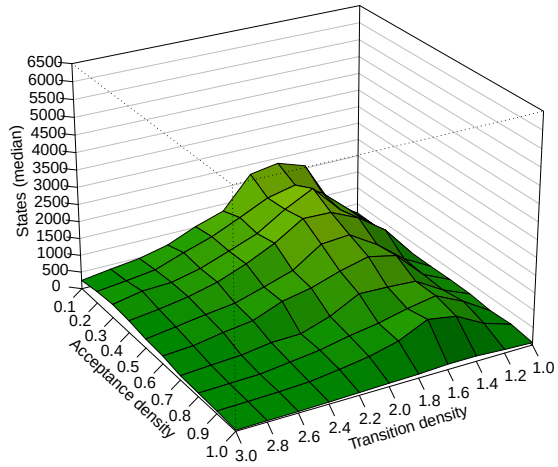
Regarding the M2 optimisation, we can see that the mountain ridge in the Fribourg+M1+M2 perspective plot is slightly lower than the one in the Fribourg+M1 perspective plot. The flatland regions, however, seem to not change much. This is reflected by the overall mean of Fribourg+M1+M2 which is slightly lower than in Fribourg+M1 (958.9 opposed to 963.2). The overall median, on the other hand, is higher for Fribourg+M1+M2 than for Fribourg+M1 (496 opposed to 482). An interpretation of this behaviour is that the application of the M2 optimisation results in smaller complements for *some* input automata. Better analysis: Fribourg+M1+M2 is better for almost all classes with an acceptance density up to 0.4, and worse for most of the classes with a n acceptance density between 0.5 and 0.9. The results are exactly identical for all the classes with an acceptance density of 1.0!. These automata are especially the hard ones that produce large complements. This positive effect of M2 does however not affect enough input automata, especially not the easy automata, as to improve the overall performance of the construction in



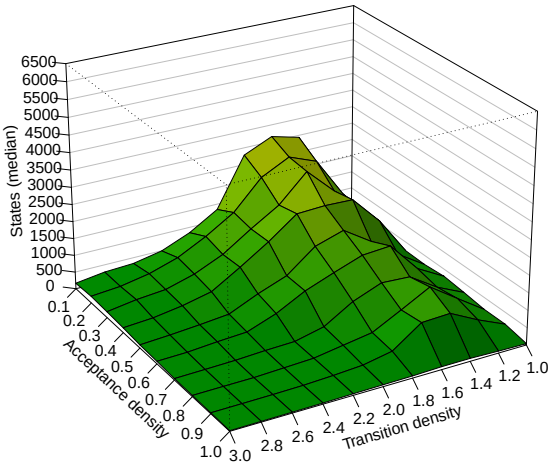

(a) Fribourg+M1




(b) Fribourg+M1+R2C




(c) Fribourg+M1+M2




(d) Fribourg+M1+R2C+C

Figure 4.3: Median complement sizes of the 10,939 effective samples from the GOAL test set for each of the 110 classes.

terms of the median complement sizes. As already stated previously, we consider therefore Fribourg+M1 as the better construction on the GOAL test set than Fribourg+M1+M2.

Finally,Fribourg+M1+R2C+C differs from Fribourg+M1+R2C in a similar way that Fribourg+R2C+C differs from Fribourg+R2C. The higher regions get higher and the lower regions get lower, that is, a performance decline on hard automata, but a performance gain on easy automata. The performance gain on the easy automata is however effective enough to decrease the overall median from 447 to 331, which is minus 26%.

With 331 states, Fribourg+M1+R2C+C has the lowest median of all the versions (apart from the special case Fribourg+R). However, we still declare Fribourg+M1+R2C as the winner on the GOAL test set, mainly for two reasons. First, while Fribourg+M1+R2C+C has a lower median, the mean is still higher (1062.6 to 937.7 which is a plus of 13.3%). This results from the complements of the hard automata, which are larger than with Fribourg+M1+R2C. From a practical point of view, the mean might be relevant, because it relates more directly to the required computing resources than the median. Indeed, the execution per complementation task in CPU time is 25.4% higher for Fribourg+M1+R2C+C than for Fribourg+M1+R2C (all measured execution time in CPU time are presented in Appendix C). The increase in the average execution time per automaton is from 4.44 to 5.57 seconds and in the total execution time from 48,572 seconds ($\approx$ 135 hours) to 60,919 seconds ($\approx$ 169 hours). Fribourg+M1+R2C, on the other hand, has the lowest mean of all versions. The second reason that we choose Fribourg+M1+R2C as the

winner and not Fribourg+M1+R2C+C is that the C option is not a real part of the construction. It actually modifies the input automata before the construction starts in order to make them better suited for the construction. Fribourg+M1+R2C, on the other hand, includes only construction-specific options.

### Difficulty Categories

As we have seen, there are big difference in the complement sizes across the different classes of the GOAL test set. Furthermore, there is a certain pattern throughout the results of all construction versions, namely the mountain. We attempted to categorise the classes of the GOAL test set into the three groups "easy", "medium", and "hard". To do so, we first averaged the matrices with the median complement sizes of all the eight versions of the Fribourg construction. In this way, we have a mean median complement size for each class. Then we defined two breakpoints that divide the classes into easy, medium, and hard groups. The breakpoints 500 and 1,600 result in an appropriate groups that seem to capture the reality well. The resulting categorisation is shown in Figure **??**.

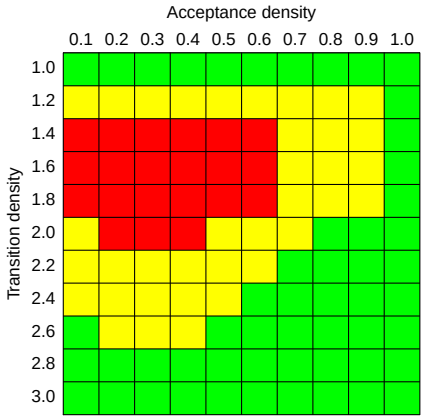



Figure 4.4: Difficulty categories of the 110 GOAL test set classes. Green: easy; yellow: medium; red: hard.

As can be seen in Figure **??**, there are 53 easy, 36 medium, and 21 hard classes. The easy classes are mainly those with extreme values. In particular, all the classes with a low or high transition density of 0.1, or 2.8 and 3.0, and a high acceptance density of 1.0 are easy. Furthermore, there is a "triangle" of easy classes between transition densities 2.0 and 2.6. and acceptance densities 0.5 and 0.9. The higher the transition density, the lower acceptance density values are tolerated for the class to be easy. The hard classes are roughly those with a transition density between 1.4 and 1.8 and an acceptance density between 0.1 and 0.6. The medium classes finally are grouped as a "belt" around the hard classes.

It is interesting that the extreme values of transition density and acceptance density result in easy automata. With a transition density of 1.0 and an alphabet size of 2, each of the 15 states has on average two outgoing and two incoming transitions. With a transition density of 3.0, each state has on average 6 outgoing and 6 incoming transitions. These low or high connectivity seems to considerably simplify the complementation task. The same applies to a high acceptance density of 1.0, which means that every state is an accepting state. Generally, we can say that automata with high acceptance densities are easier to complement than automata with lower acceptance densities. This also means that the pattern of easy automata at the extreme values of transition and acceptance density, does not apply to to the lower extreme of the acceptance density. Automata with a very low acceptance density of 0.1 are hard to complement—unless they are made easy by a low or high transition density.

Another interesting point is that the hard automata have transition densities between 1.4 and 1.8. It seems that this range of transition densities is the crucial factor in the hardness of a complementation task, and that it is only alleviated by a growing acceptance density. This explains the decline of the mountain ridge from low to high acceptance density values.

Summarising we can say that transition densities between 1.4 and 1.8 produce the hardest complementation tasks, and that to the both sides the difficulty steadily decreases with declining or growing transition

density. Furthermore, a growing acceptance density generally implies easier complementation tasks.

### 4.1.2 Michel Test Set

Our second test set consists of the four Michel automata that are listed in Figure **??** in Section 3.2.2. They have 3, 4, 5, and 6 states, respectively. The Fribourg construction versions that we tested on the Michel automata are the following.

1. Fribourg
2. Fribourg+R2C
3. Fribourg+M1
4. Fribourg+M1+M2
5. Fribourg+M1+M2+R2C
6. Fribourg+R

The resulting complement sizes are listed in Table 4.3.

| Construction | Michel 1 | Michel 2 | Michel 3 | Michel 4 | Fitted curve | Std. error |
|---|---|---|---|---|---|---|
| Fribourg | 57 | 843 | 14,535 | 287,907 | $(1.35n)^n$ | 0.01% |
| Fribourg+R2C | 33 | 467 | 8,271 | 168,291 | $(1.24n)^n$ | 0.06% |
| Fribourg+M1 | 44 | 448 | 5,506 | 81,765 | $(1.10n)^n$ | 0.07% |
| Fribourg+M1+M2 | 42 | 402 | 4,404 | 57,116 | $(1.03n)^n$ | 0.12% |
| Fribourg+M1+M2+R2C | 28 | 269 | 3,168 | 43,957 | $(0.99n)^n$ | 0.04% |
| Fribourg+R | 18 | 95 | 528 | 3,315 | $(0.64n)^n$ | 0.35% |

Table 4.3: Complement sizes of the Michel automata with $m = \{1, \ldots, 4\}$ and 3, 4, 5, and 6 states, respectively.

In the second-last column "Fitted curve" of Table 4.3, we fitted a function of the form $(an)^n$ to the measured four data points. These data points consist of the sizes of the four Michel automata (3, 4, 5, and 6) as the $x$-values ad the corresponding complement sizes as the $y$-values. The fitted function $(an)^n$ can be seen as an "averaged" state growth of these four automata ($n$ is the size of the input automaton). The last column "Std. error" contains the standard error that resulted from the fit.

We can see in Table 4.3 that the state growths are indeed very large. For example, complementing Michel 4, which has six states, with the plain Fribourg construction results in a complement of 287,907 states. However, the optimisations R2C, M1, and M2 have a large influence on the complement sizes. If we consider Michel 4, then the R2C optimisation alone reduces the complement size from 287,907 to 168,291 which is a reduction of 51.5%. The M1 optimisation has an eve larger influence as it reduces the complement size from 287,907 to 81,765 which is a reduction of 71.6%. Adding M2 to M1 further reduces the complement size of Fribourg+M1 by 30.1% (from 81,765 to 57,116). Finally, adding R2C on top of M1 and M2 brings a further reduction of of 23% (from 57,116 to 43,957). If we compare the most efficient version (Fribourg+M1+M2+R2C) with the least efficient one (Fribourg), then the complement size of the former version is only 15.3% of the complement size of the latter version.

It is interesting to see that for the Michel automata Fribourg+M1+M2 is more efficient than Fribourg+M1. For the GOAL test set, Friburg+M1+M2 had a slightly higher median than Fribourg+M1 although it had a slightly lower mean. We identified in Section 4.1.1 that the M2 optimisation has a positive effect only on some automata, and that these are mostly the hard automata (The automata with acceptance densities up to 0.4). Michel automata are very hard automata (The acceptance densities of the four tested Michel automata are 0.25 or less), and indeed the M2 optimisation has a considerably positive effect. These results support thus the observation we made in Section 4.1.1.

The special version Fribourg+R yields very small complements compared to the other versions. This tells us that the complements of the other versions contain a large number of unreachable and dead states. For example, the complement of Michel 4 of Fribourg+R (3,315 states) is 1.2% of the size of the complement

of Fribourg. This means that 98.8% of the 287,907 states of the complement of Fribourg are unreachable and dead states. This is actually not surprising, because, following the proof of Michel [15][29], the smallest possible complement of Michel 4 has 24 states. This is because Michel 4 has $m = 4$ and Michel proved that the complement has at least size $m!$. This means that that even after reducing all the unreachable and dead states from the complement of Fribourg, an even much smaller complement would still be possible.

Up to now we just looked at the specific results of Michel 4. The fitted functions of the form $(an)^n$ summarise the results of all the four Michel automata. These functions give us reference points for the worst-case state complexities of the different versions of the Fribourg construction. For example, for the plain Fribourg construction with its fitted function of $(1.35n)^n$, we have now the proof that this construction produces complements of size $(1.35n)^n$, where $n$ is the size of the input automaton. This means that the worst-case complexity cannot be lower than $(1.35n)^n$ (but it can still be higher). This bound decreases for the different versions of the Fribourg construction down to $(0.99n)^n$ for Fribourg+M1+M2+R2C.

In Table 4.4 we show the measured execution time in seconds (CPU time) for each complementation task. We can see that the difference between the least and most efficient version is bigger than for the complement sizes. For example for Michel 4, Fribourg+M1+M2+R2C is more than 43 times faster than Fribourg (2,332.6 seconds compared to 100,976 seconds). In more familiar unities, this corresponds to approximately 39 minutes for Fribourg+M1+M2+R2C against 28 hours for Fribourg. We also fitted functions of the form $(an)^n$ to the measured execution times where $n$ is the number of states of the input automaton, and the value of the function is the execution time of the task in CPU time seconds.

| Construction | Michel 1 | Michel 2 | Michel 3 | Michel 4 | Fitted curve | Std. error |
|---|---|---|---|---|---|---|
| Fribourg | 2.3 | 4.0 | 88.8 | 100,976.0 | $(1.14n)^n$ | 0.64% |
| Fribourg+R2C | 2.3 | 3.4 | 27.4 | 27,938.3 | $(0.92n)^n$ | 0.64% |
| Fribourg+M1 | 2.2 | 3.6 | 17.9 | 6,508.4 | $(0.72n)^n$ | 0.63% |
| Fribourg+M1+M2 | 2.3 | 3.5 | 13.8 | 2,707.4 | $(0.62n)^n$ | 0.62% |
| Fribourg+M1+M2+R2C | 2.5 | 3.5 | 10.8 | 2,332.6 | $(0.61n)^n$ | 0.62% |
| Fribourg+R | 2.4 | 3.7 | 86.0 | 101,809.6 | $(1.14n)^n$ | 0.64% |

Table 4.4: Execution time in seconds (CPU time) for complementing the Michel automata 1 to 4.

The fitted functions that we calculated for the measured complement sizes and execution times are based on only four data points. This is generally not enough to make reliable extrapolations. However, it is still interesting to do such an extrapolation in order to see the involved complexity and to show why we were restricted to include only the first four Michel automata in the test set. In Table 4.5, we show extrapolated values for the complement sizes and execution times for the plain Fribourg construction (the least efficient one), based on the corresponding fitted functions. The table includes the extrapolated values for the Michel automata 5 to 8, which have 7 to 10 states.

| Automaton | States ($n$) | Compl. size $(1.35n)^n$ | Exec. time $(1.14n)^n$ | $\approx$ days/months/years |
|---|---|---|---|---|
| Michel 5 | 7 | 6,882,980 | 2,020,385 | 23 days |
| Michel 6 | 8 | 189,905,394 | 46,789,245 | 18 months |
| Michel 7 | 9 | 5,939,189,262 | 1,228,250,634 | 39 years |
| Michel 8 | 10 | 207,621,228,081 | 36,039,825,529 | 1,142 years |

Table 4.5: Extrapolated values for the complement sizes and execution times (seconds CPU time) for the Michel automata with $m = \{5, \ldots, 10\}$ with the Fribourg version of the Fribourg construction.

According to the fitted state growth function, the complement of Michel 5 would have nearly 7 million states, and the complement of Michel 8 even more than 207 billion states. Already the computation of the 7 million states of Michel 5 would most probably exceed the available memory resources in our computing environment. Regarding execution time, the complementation of Michel 5 would take 23 days. This would by far exceed the maximal running time of a job on our computer cluster. And even if we would not have these administrative time restriction, the time to wait for the complementation of Michel 5 to 8, between 18 months and 1,142 years, is definitely too long, even for a master's thesis.

## 4.2 External Tests

In the external tests we compared the most efficient version of the Fribourg construction to three other constructions. These constructions are Piterman+EQ+RO, Slice+P+RO+MADJ+EG, and Rank+TR+RO. The most efficient version of the Fribourg construction is Fribourg+M1+R2C for the GOAL test set, and Fribourg+M1+M2+R2C for the Michel test set. We present the results from these two test sets separately in the following two sections.

### 4.2.1 GOAL Test Set

As for the internal tests on the GOAL test set, we set a time limit of 600 seconds CPU time, and a Java heap size limit of 1 GB per complementation task. Table 4.6 shows the number of timeouts and memory excesses that we observed for the four constructions.

| Construction | Timeouts | Memory excesses |
|---|---|---|
| Piterman+EQ+RO | 2 | 0 |
| Slice+P+RO+MADJ+EG | 0 | 0 |
| Rank+TR+RO | 3,713 | 83 |
| Fribourg+M1+R2C | 1 | 0 |

Table 4.6: Number of timeouts and memory excesses for the GOAL test set.

#### With Rank

Most salient in Table 4.6 is the high number of aborted tasks for the Rank construction. 3,317 of the 11,000 automata (33.8%) were aborted due to a timeout, and further 83 (0.8%) due to a memory excess. Regarding the other constructions, there are just two timeouts for the Piterman construction, and a single timeout for the Fribourg construction.

Determining the effective samples of these runs gives a number of 7,204 automata, which is 65.5% of the total number of automata. In Figure 4.5 we present the complement sizes of these 7,204 effective samples as a stripchart.

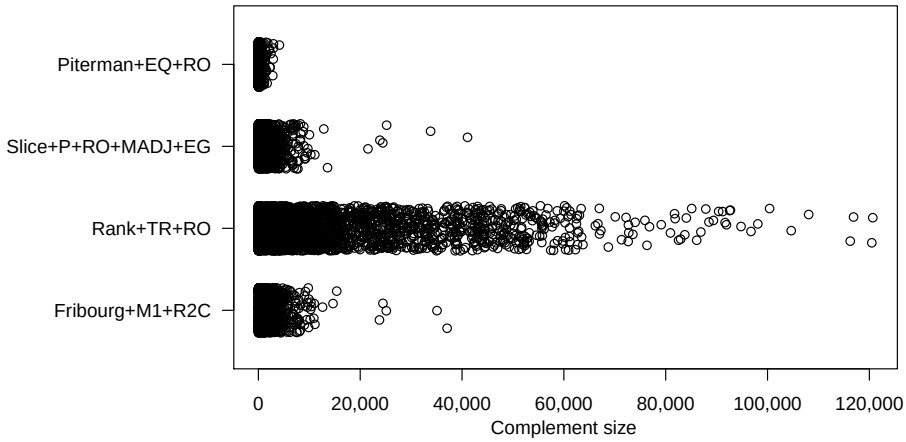



Figure 4.5: Complement sizes of the 7,204 effective samples.

The stripchart makes the reason for the high number of aborted tasks of the Rank construction apparent. Rank+TR+RO produces a high number of very large complements compared to the other constructions.

But from the stripchart in Figure 4.6 alone we cannot yet tell whether the Rank construction *generally* produces larger complements than the other constructions, or if this holds just for *some* automata. To this end we have to inspect the statistics about the distribution of the complement sizes in Table 4.7.

| Construction | Mean | Min. | P25 | Median | P75 | Max. |
|---|---|---|---|---|---|---|
| Piterman+EQ+RO | 106.0 | 1 | 29.0 | 58.0 | 121.0 | 4,126 |
| Slice+P+RO+MADJ+EG | 555.4 | 2 | 70.0 | 202.0 | 596.0 | 41,081 |
| Rank+TR+RO | 5,255.6 | 2 | 81.0 | 254.5 | 3,178.2 | 120,674 |
| Fribourg+M1+R2C | 662.9 | 2 | 101.0 | 269.0 | 754.5 | 37,068 |

Table 4.7: Statistics of complement sizes of the 7,204 effective samples

And indeed, the 25th percentile and the median of Rank are higher than for Piterman and Slice, but still lower than for our Fribourg construction. This means that the Rank construction produces more smaller complements than the Fribourg construction. However, the picture changes dramatically for the 75th percentile where the value of Rank is more than four times higher than the value for Fribourg. Also the mean of Rank is many times higher than the means of all the other constructions. A possible explanation for this is that the Rank construction has a comparable performance with the other constructions for easy automata. For harder automata, however, the performance of Rank is much worse than the other constructions. In addition, the automata that are hardest for Rank are not even included in this analysis as it includes only the 7,204 effective samples. The 3,796 automata that are excluded would probably have resulted in even larger complements with the Rank construction.

What we cannot tell is whether the automata which are hard for Rank are the same that are hard for the other constructions. However, as we will see later, we think that this is not necessarily the case.

### Without Rank

Given the large number of aborted complementation tasks of Rank we decided to do the main analysis and comparison of the results without the Rank construction. Because with the Rank construction would basically exclude more than one third of the tasks that have been successfully completed by the other constructions from the result analysis. Our main interest is however to compare the performance of the Fribourg construction to the other constructions. In this way, we would probably miss important aspects in the result analysis of the other three constructions.

Without the Rank construction there are 10,998 effective samples In Figure 4.6 we display the complement sizes of these 10,998 effective samples as a stripchart.

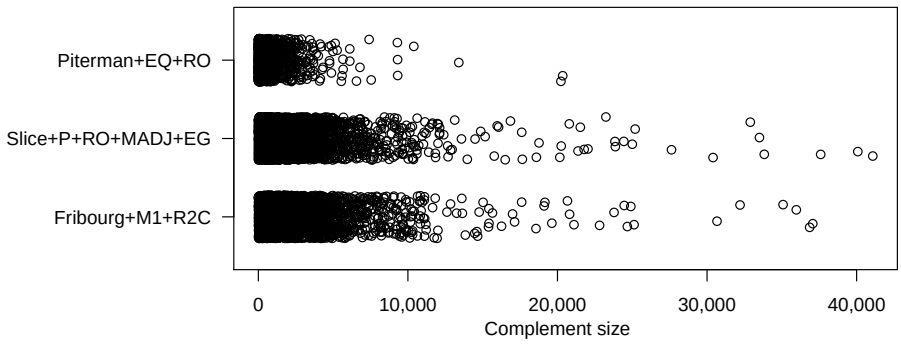



Figure 4.6: Complement sizes of the 10,998 effective samples.

From the stripchart we can see that Fribourg and Slice have a comparable distribution of complement sizes, whereas Piterman has a considerably higher concentration of small complement sizes. We can say that Piterman generally produces smaller complement than Fribourg and Slice.

We present the statistics of these distributions in Table 4.8. Indeed, for all statistics Piterman has values that are multiple times lower than the ones of Fribourg and Slice. It is interesting that for mean, 25th

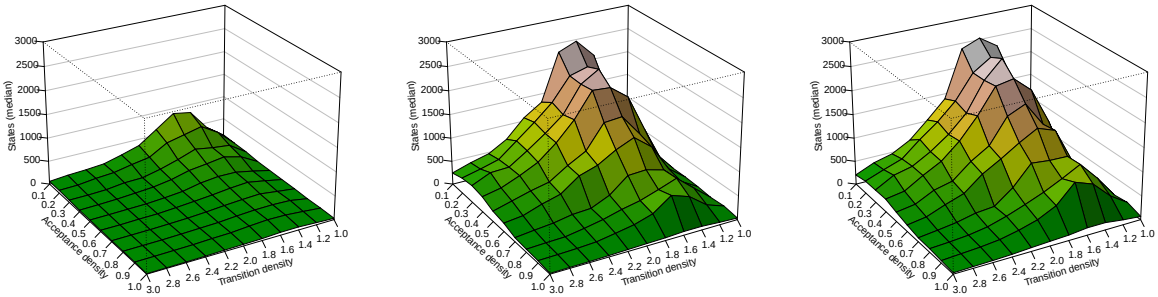
percentile, median, and 75th percentile the values of Piterman are more or less five times smaller. It seems like Piterman would produce complements that are throughout five times smaller than the complements of Fribourg and Slice.

| Construction | Mean | Min. | P25 | Median | P75 | Max. |
|---|---|---|---|---|---|---|
| Piterman+EQ+RO | 209.6 | 1 | 38.0 | 80.0 | 183.0 | 20,349 |
| Slice+P+RO+MADJ+EG | 949.4 | 2 | 120.0 | 396.0 | 1,003.0 | 41,081 |
| Fribourg+M1+R2C | 1,017.3 | 2 | 153.0 | 452.0 | 1,134.0 | 37,068 |

Table 4.8: Aggregated statistics of complement sizes of the 10,998 effective samples without Rank.

Comparing Fribourg and Slice, there is a slight advantage for Slice. Mean, 25th percentile, median, and 75th percentile are lower for Slice than for Fribourg by 6.7%, 21.6%, 12.4%, and 11.6%, respectively. We have to conclude that from an overall point of view, the Fribourg construction has the second-worst performance for the GOAL test set after Piterman and Slice, and before Rank.

Figure 4.7 shows the perspective plots with the median complement sizes for the 110 classes of the 10,998 samples. As already mentioned, the corresponding matrices can be found in Appendix B.1.




(a) Piterman+EQ+RO (b) Slice+P+RO+MADJ+EG (c) Fribourg+M1+R2C

Figure 4.7: Median complement sizes (10,998 samples)

Note that the plot of Fribourg+M1+R2C in Figure 4.7 (c) is the same as the one in Figure 4.3 (b). The only difference is the scale of the vertical axis.

In the perspective plots we can see that the pattern for Fribourg and Slice are very similar. The median complement sizes in the individual classes do not differ a lot, both relatively and absolutely. However, the medians of Fribourg seem to be throughout (with some exceptions) slightly higher than the ones of Slice. This means that Fribourg and Slice seem to have similar strengths and weaknesses, but Slice is slightly more efficient on the tested automata.

Piterman, as expected, has medians that are multiple times lower than the corresponding medians of Fribourg and Slice. The basic pattern, however, is still similar. There is a mountain ridge along the classes with a transition density of 1.6 with its top in the class with transition density 1.6 and acceptance density 0.1.

### 4.2.2 Michel Test Set

For the Michel test set we used the same three third-party construction as for the GOAL test set, namely Piterman+EQ+RO, Slice+P+RO+MADJ+EG, and Rank+TR+RO. The used Fribourg construction version is however Fribourg+M1+M2+R2C, as this is the most efficient version of the Fribourg construction on the Michel test set.

The resulting complement sizes are shown in Table 4.8. Again, we fitted a function of the form $(an)^n$ to the four measured data points of each construction and calculated the standard error of this fit.

Considering the results of the GOAL test set, the results in Table 4.8 are surprising. Rank is the most efficient construction. It produces the smallest complements for all Michel automata, and with $(0.91n)^n$

| Construction | Michel 1 | Michel 2 | Michel 3 | Michel 4 | Fitted curve | Std. error |
|---|---|---|---|---|---|---|
| Piterman+EQ+RO | 23 | 251 | 5,167 | 175,041 | $(1.25n)^n$ | 0.29% |
| Slice+P+RO+MADJ+EG | 35 | 431 | 6,786 | 123,180 | $(1.18n)^n$ | 0.02% |
| Rank+TR+RO | 23 | 181 | 1,884 | 25,985 | $(0.91n)^n$ | 0.01% |
| Fribourg+M1+M2+R2C | 28 | 269 | 3,168 | 43,957 | $(0.99n)^n$ | 0.04% |

Figure 4.8: Complement sizes of the first four Michel automata.

it has the flattest fitted curve of all constructions. This is surprising because for the GOAL test set, Rank produced by far the largest complements, and 34.5% of the test data could not even be completed within the given time and memory limits. With the Michel automata, however, the case seems to be reversed and Rank produces by far the smallest complements.

Rank is followed by the Fribourg construction, which has the second-smallest complements for Michel 3 and 4, and with $(0.99n)^n$ the second-flattest fitted curve. The complements of Michel 2, 3, and 4 of the Fribourg construction are bigger than the ones of the Rank construction by 48.6%, 68.2%, and 69.2%, respectively.

The construction with the next steeper fitted curve of $(1.18n)^n$ is the Slice construction. for Michel 1 to 3, this is actually the worst construction, but then for Michel 4, the complement is smaller than the one of Piterman what results in the flatter fitted curve. The gap to the Fribourg construction is big. The complement sizes of Michel 2 to 4 exceed the ones of Fribourg by 60.2%, 114.2%, and 180.2%, respectively. This is also a remarkable point, because for the GOAL test set, Fribourg and Slice showed a very similar performance.

The last in the ranking is Piterman with a fitted curve of $(1.25n)^n$. However, a special fact for Piterman is that it has the smallest complement for Michel 1 (together with Rank), the second-smallest for Michel 2, the third-smallest for Michel 3, and the largest for Michel 4. It is actually the large complement of Michel 4 that makes Piterman having the steepest fitted curve. However, it is still remarkable that this construction, which is by far the most efficient for the GOAL test set, produces so much worse results for the Michel automata than all the other constructions. Compared with the Rank construction, Piterman's complements of Michel 2 to 4 are 38.7%, 174.3%, and 573.6%, respectively, bigger. Compared to the Fribourg construction, Piterman produces slightly smaller complements for Michel 1 and 2, but larger ones for Michel 3 and 4. Namely, they are 63.1% and 298.2% larger than the corresponding ones of the Fribourg construction.

Summarising we can say that the ranking of the constructions for the Michel test set is exactly the reverse of the ranking for the GOAL test set. The by far worst construction for the GOAL test set (Rank) is the best one for the Michel test set, and the by far best construction for the GOAL test set (Piterman) is the worst one for the Michel test set (at least for Michel 4). For the Fribourg construction this means that it "advances" from rank 3 for the GOAL test set to rank 2 for the Michel test set.

In Table 4.9 we present the execution times per complementation task in CPU time seconds. As for the complement sizes, we fitted a function of the form $(an)^n$ to the measured execution times where $n$ is the size of the input automaton.

| Construction | Michel 1 | Michel 2 | Michel 3 | Michel 4 | Fitted curve | Std. error |
|---|---|---|---|---|---|---|
| Piterman+EQ+RO | 2.5 | 3.8 | 42.6 | 75,917.4 | $(1.08n)^n$ | 0.64% |
| Slice+P+RO+MADJ+EG | 2.3 | 3.6 | 11.4 | 159.5 | $(0.39n)^n$ | 0.38% |
| Rank+TR+RO | 2.2 | 3.0 | 6.4 | 30.0 | $(0.29n)^n$ | 0.18% |
| Fribourg+M1+M2+R2C | 2.5 | 3.5 | 10.8 | 2,332.6 | $(0.61n)^n$ | 0.62% |

Table 4.9: Execution times for the first four Michel automata.

Most interesting in Table 4.9 is the column with the times for Michel 4. The time difference between the best and the worst construction is enormous. While the Rank construction took just 30 seconds to complement Michel 4, the Piterman construction took 75,917.4 seconds which is approximately 21 hours.

This is more than 2500 times longer. Of course the Piterman construction produced a bigger automata, which naturally requires more time, however, the automaton produced by the Piterman construction is just around 6.7 times bigger than the one of the Rank construction. This means that the Piterman construction must include very inefficient processes before finally arriving at the output automaton.

Furthermore, we can see in Table 4.9 that also the Fribourg construction took relatively long to complement Michel 4 compared to Rank, namely 2,332.6 seconds which are approximately 39 minutes. This is 77.8 times longer than the 30 seconds of Rank. At the same time, Fribourg's complement has just 68.2% more states than Rank's complement. Similarly, compared to the Slice construction the Fribourg construction is slow for Michel 4. Slice's complement is 2.8 times bigger than Fribourg's complement, but with 159.5 seconds the complementation of slice was 14.6 times faster than the complementation of Fribourg.

So there seems to an inefficiency in the Fribourg construction in terms of execution time for the complementation of Michel 4. However, this inefficiency is by far not as pronounced as for Piterman. While the complement of Piterman is just 4 times bigger, the execution time of Piterman is 32.5 times longer than the one of the Fribourg construction. One could also look at it from the other side and say that not the Fribourg construction is inefficient on Michel 4, but that Rank and Slice are extremely efficient on this automaton.

Finally, these interesting differences in the execution times between the four constructions can only be observed for Michel 4. For Michel 3 there are also differences but they are by far not as pronounced as for Michel 4. If the computational resources would allow it, it would be very interesting to run the constructions on Michel 5 and beyond. One thing that stays the same for all the four Michel automata is that Rank is always the fastest and Piterman always the slowest construction.

## 4.3 Summary and Discussion of the Results

## 4.4 Limitations of the Approach

# Appendix A

# Plugin Installation and Usage

Since between the 2014–08–08 and 2014–11–17 releases of GOAL certain parts of the plugin interfaces have changed, and we adapted our plugin accordingly, the currently maintained version of the plugin works only with GOAL versions 2014-11-17 or newer. It is thus essential for any GOAL user to update to this version in order to use our plugin.

# Appendix B

# Median Complement Sizes of the GOAL Test Set

Bla bla bla

| | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1.0 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1.0 | 269 | 308 | 254 | 236 | 238 | 297 | 266 | 156 | 207 | 68 |
| 1.2 | 960 | 1,407 | 1,479 | 2,150 | 1,152 | 1,090 | 942 | 1,206 | 718 | 104 |
| 1.4 | 3,426 | 2,915 | 2,752 | 3,393 | 2,693 | 3,265 | 2,263 | 2,425 | 1,844 | 154 |
| 1.6 | 3,799 | 3,698 | 4,901 | 3,926 | 3,960 | 3,655 | 2,580 | 1,905 | 2,124 | 155 |
| 1.8 | 3,375 | 3,169 | 3,420 | 3,967 | 3,943 | 3,132 | 2,246 | 1,144 | 971 | 114 |
| 2.0 | 1,906 | 2,261 | 2,383 | 2,884 | 2,354 | 2,096 | 1,169 | 932 | 568 | 98 |
| 2.2 | 1,467 | 1,633 | 1,795 | 1,942 | 1,611 | 1,640 | 569 | 499 | 330 | 78 |
| 2.4 | 924 | 1,232 | 1,319 | 1,317 | 1,056 | 886 | 514 | 314 | 182 | 59 |
| 2.6 | 625 | 763 | 880 | 945 | 828 | 684 | 316 | 175 | 132 | 44 |
| 2.8 | 483 | 584 | 836 | 690 | 575 | 395 | 240 | 151 | 103 | 41 |
| 3.0 | 319 | 450 | 557 | 523 | 367 | 313 | 155 | 116 | 84 | 32 |

(a) Fribourg

| | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1.0 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1.0 | 269 | 308 | 254 | 236 | 238 | 297 | 266 | 156 | 207 | 68 |
| 1.2 | 960 | 1,407 | 1,479 | 2,150 | 1,152 | 1,090 | 942 | 1,206 | 718 | 104 |
| 1.4 | 3,426 | 2,915 | 2,752 | 3,393 | 2,693 | 3,265 | 2,263 | 2,425 | 1,844 | 154 |
| 1.6 | 3,799 | 3,698 | 4,901 | 3,926 | 3,960 | 3,655 | 2,580 | 1,905 | 2,124 | 155 |
| 1.8 | 3,375 | 3,169 | 3,420 | 3,967 | 3,943 | 3,093 | 2,246 | 1,144 | 971 | 114 |
| 2.0 | 1,906 | 2,184 | 2,383 | 2,818 | 2,354 | 1,989 | 1,127 | 885 | 568 | 97 |
| 2.2 | 1,410 | 1,561 | 1,639 | 1,884 | 1,609 | 1,588 | 496 | 464 | 284 | 78 |
| 2.4 | 884 | 1,200 | 1,234 | 1,184 | 939 | 806 | 373 | 256 | 165 | 55 |
| 2.6 | 575 | 731 | 815 | 860 | 751 | 575 | 246 | 162 | 114 | 43 |
| 2.8 | 431 | 530 | 672 | 466 | 371 | 274 | 174 | 120 | 85 | 36 |
| 3.0 | 232 | 325 | 344 | 360 | 269 | 169 | 91 | 85 | 53 | 27 |

(b) Fribourg+R2C

| | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1.0 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1.0 | 390 | 438 | 434 | 324 | 328 | 459 | 337 | 204 | 227 | 40 |
| 1.2 | 1,576 | 2,394 | 2,505 | 2,996 | 1,613 | 1,551 | 1,166 | 1,542 | 1,002 | 58 |
| 1.4 | 5,007 | 4,336 | 4,652 | 4,877 | 3,458 | 3,956 | 3,169 | 3,380 | 1,868 | 86 |
| 1.6 | 5,067 | 5,032 | 6,444 | 4,868 | 4,575 | 3,864 | 3,211 | 1,731 | 1,892 | 85 |
| 1.8 | 4,016 | 3,701 | 3,647 | 4,523 | 3,548 | 3,009 | 1,808 | 451 | 336 | 62 |
| 2.0 | 1,663 | 2,276 | 2,676 | 3,035 | 1,925 | 1,932 | 464 | 307 | 150 | 54 |
| 2.2 | 989 | 1,514 | 1,621 | 1,826 | 1,121 | 846 | 155 | 127 | 93 | 45 |
| 2.4 | 560 | 821 | 919 | 771 | 529 | 267 | 133 | 87 | 55 | 32 |
| 2.6 | 388 | 519 | 524 | 441 | 259 | 219 | 84 | 50 | 41 | 26 |
| 2.8 | 311 | 317 | 396 | 242 | 165 | 95 | 64 | 44 | 33 | 22 |
| 3.0 | 173 | 224 | 211 | 169 | 102 | 72 | 41 | 34 | 27 | 18 |

(c) Fribourg+R2C+C

| | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1.0 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1.0 | 225 | 223 | 195 | 181 | 187 | 199 | 189 | 124 | 161 | 68 |
| 1.2 | 731 | 971 | 946 | 1,071 | 629 | 562 | 488 | 568 | 388 | 104 |
| 1.4 | 2,228 | 1,701 | 1,543 | 1,732 | 1,241 | 1,287 | 945 | 944 | 727 | 154 |
| 1.6 | 2,489 | 2,263 | 2,331 | 2,133 | 1,777 | 1,443 | 964 | 757 | 889 | 155 |
| 1.8 | 2,381 | 2,027 | 2,009 | 2,075 | 1,618 | 1,243 | 1,005 | 592 | 515 | 114 |
| 2.0 | 1,390 | 1,569 | 1,416 | 1,573 | 1,093 | 1,008 | 594 | 464 | 330 | 98 |
| 2.2 | 1,118 | 1,197 | 1,150 | 1,151 | 879 | 809 | 317 | 330 | 241 | 78 |
| 2.4 | 712 | 885 | 836 | 809 | 580 | 535 | 316 | 231 | 145 | 59 |
| 2.6 | 498 | 569 | 601 | 627 | 497 | 412 | 217 | 137 | 113 | 44 |
| 2.8 | 391 | 455 | 578 | 456 | 374 | 263 | 173 | 119 | 90 | 41 |
| 3.0 | 258 | 350 | 392 | 354 | 253 | 208 | 119 | 97 | 74 | 32 |

(d) Fribourg+M1

| | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1.0 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1.0 | 215 | 213 | 189 | 174 | 175 | 192 | 186 | 121 | 156 | 68 |
| 1.2 | 712 | 914 | 913 | 1,075 | 619 | 563 | 526 | 620 | 416 | 104 |
| 1.4 | 2,075 | 1,620 | 1,503 | 1,650 | 1,254 | 1,339 | 1,003 | 1,006 | 848 | 154 |
| 1.6 | 2,344 | 2,062 | 2,340 | 2,016 | 1,755 | 1,520 | 1,053 | 858 | 986 | 155 |
| 1.8 | 2,205 | 1,873 | 1,920 | 2,040 | 1,689 | 1,315 | 1,080 | 664 | 598 | 114 |
| 2.0 | 1,290 | 1,485 | 1,405 | 1,522 | 1,134 | 1,044 | 652 | 531 | 392 | 98 |
| 2.2 | 1,023 | 1,119 | 1,092 | 1,127 | 868 | 875 | 376 | 359 | 262 | 78 |
| 2.4 | 674 | 849 | 790 | 807 | 617 | 544 | 355 | 251 | 156 | 59 |
| 2.6 | 478 | 549 | 594 | 597 | 510 | 431 | 231 | 147 | 116 | 44 |
| 2.8 | 370 | 439 | 559 | 455 | 382 | 283 | 182 | 124 | 93 | 41 |
| 3.0 | 249 | 341 | 388 | 348 | 260 | 225 | 123 | 101 | 77 | 32 |

(e) Fribourg+M1+M2

| | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1.0 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1.0 | 225 | 223 | 195 | 181 | 187 | 199 | 189 | 124 | 161 | 68 |
| 1.2 | 731 | 971 | 946 | 1,071 | 629 | 562 | 488 | 568 | 388 | 104 |
| 1.4 | 2,228 | 1,701 | 1,543 | 1,732 | 1,241 | 1,287 | 945 | 944 | 727 | 154 |
| 1.6 | 2,489 | 2,263 | 2,331 | 2,133 | 1,777 | 1,443 | 964 | 757 | 889 | 155 |
| 1.8 | 2,381 | 2,027 | 2,009 | 2,075 | 1,618 | 1,215 | 1,005 | 592 | 515 | 114 |
| 2.0 | 1,390 | 1,513 | 1,416 | 1,542 | 1,093 | 1,003 | 594 | 441 | 330 | 97 |
| 2.2 | 1,019 | 1,156 | 1,064 | 1,104 | 859 | 785 | 304 | 303 | 221 | 78 |
| 2.4 | 672 | 867 | 789 | 772 | 544 | 478 | 269 | 191 | 139 | 55 |
| 2.6 | 466 | 542 | 572 | 568 | 452 | 348 | 183 | 129 | 99 | 43 |
| 2.8 | 368 | 407 | 480 | 337 | 260 | 197 | 129 | 96 | 75 | 36 |
| 3.0 | 201 | 261 | 266 | 272 | 199 | 136 | 83 | 74 | 50 | 27 |

(f) Fribourg+M1+R2C

| | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1.0 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1.0 | 329 | 303 | 279 | 240 | 229 | 288 | 230 | 157 | 160 | 40 |
| 1.2 | 988 | 1,392 | 1,356 | 1,352 | 751 | 741 | 608 | 704 | 516 | 58 |
| 1.4 | 2,939 | 2,581 | 2,066 | 2,190 | 1,351 | 1,622 | 1,132 | 1,261 | 932 | 86 |
| 1.6 | 3,150 | 2,900 | 2,842 | 2,218 | 1,885 | 1,563 | 1,177 | 821 | 896 | 85 |
| 1.8 | 2,782 | 2,485 | 2,047 | 2,180 | 1,625 | 1,269 | 855 | 395 | 309 | 62 |
| 2.0 | 1,338 | 1,638 | 1,544 | 1,566 | 979 | 957 | 349 | 261 | 147 | 54 |
| 2.2 | 838 | 1,125 | 993 | 1,027 | 667 | 521 | 153 | 125 | 93 | 45 |
| 2.4 | 494 | 700 | 624 | 524 | 296 | 214 | 126 | 87 | 55 | 32 |
| 2.6 | 327 | 434 | 383 | 334 | 212 | 163 | 82 | 50 | 41 | 26 |
| 2.8 | 283 | 273 | 305 | 202 | 144 | 95 | 60 | 44 | 33 | 22 |
| 3.0 | 164 | 200 | 173 | 142 | 92 | 72 | 41 | 34 | 27 | 18 |

(g) Fribourg+M1+R2C+C

| | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1.0 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1.0 | 126 | 118 | 97 | 60 | 51 | 52 | 62 | 36 | 48 | 30 |
| 1.2 | 432 | 517 | 345 | 262 | 160 | 126 | 92 | 120 | 109 | 40 |
| 1.4 | 1,044 | 331 | 133 | 89 | 45 | 22 | 19 | 31 | 27 | 20 |
| 1.6 | 358 | 24 | 11 | 5 | 4 | 6 | 5 | 3 | 3 | 4 |
| 1.8 | 19 | 5 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2.0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2.2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2.4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2.6 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2.8 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 3.0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

(h) Fribourg+R

Figure B.1: Median complement sizes of the 10,939 effective samples of the internal tests on the GOAL test set. The rows (1.0 to 3.0) are the transition densities, and the columns (0.1 to 1.0) are the acceptance densities.

| | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1.0 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1.0 | 130 | 117 | 109 | 77 | 69 | 61 | 56 | 40 | 40 | 29 |
| 1.2 | 387 | 456 | 352 | 281 | 155 | 136 | 101 | 105 | 75 | 45 |
| 1.4 | 822 | 683 | 394 | 376 | 230 | 204 | 151 | 120 | 105 | 63 |
| 1.6 | 890 | 594 | 458 | 321 | 237 | 178 | 134 | 114 | 113 | 61 |
| 1.8 | 624 | 507 | 324 | 275 | 196 | 136 | 110 | 92 | 89 | 41 |
| 2.0 | 362 | 286 | 211 | 176 | 117 | 103 | 79 | 64 | 59 | 34 |
| 2.2 | 248 | 222 | 124 | 116 | 82 | 73 | 56 | 52 | 50 | 28 |
| 2.4 | 147 | 145 | 114 | 87 | 56 | 48 | 43 | 39 | 35 | 19 |
| 2.6 | 115 | 117 | 67 | 61 | 47 | 42 | 32 | 29 | 29 | 15 |
| 2.8 | 95 | 71 | 52 | 45 | 38 | 29 | 27 | 25 | 23 | 13 |
| 3.0 | 59 | 60 | 47 | 35 | 32 | 27 | 22 | 21 | 20 | 10 |

(a) Piterman+EQ+RO

| | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1.0 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1.0 | 171 | 174 | 166 | 124 | 118 | 117 | 100 | 67 | 84 | 35 |
| 1.2 | 622 | 833 | 803 | 877 | 529 | 398 | 320 | 372 | 215 | 53 |
| 1.4 | 2,086 | 1,618 | 1,367 | 1,676 | 1,065 | 967 | 664 | 682 | 494 | 78 |
| 1.6 | 2,465 | 2,073 | 2,182 | 1,959 | 1,518 | 1,259 | 767 | 545 | 623 | 78 |
| 1.8 | 2,310 | 1,963 | 1,950 | 1,988 | 1,485 | 1,095 | 746 | 418 | 346 | 57 |
| 2.0 | 1,318 | 1,482 | 1,393 | 1,461 | 981 | 871 | 434 | 338 | 228 | 50 |
| 2.2 | 1,068 | 1,145 | 1,085 | 1,067 | 772 | 747 | 263 | 235 | 158 | 40 |
| 2.4 | 689 | 838 | 809 | 751 | 524 | 466 | 240 | 159 | 93 | 30 |
| 2.6 | 469 | 531 | 555 | 565 | 437 | 360 | 169 | 94 | 71 | 23 |
| 2.8 | 369 | 421 | 536 | 405 | 329 | 224 | 130 | 81 | 58 | 21 |
| 3.0 | 244 | 327 | 360 | 322 | 219 | 176 | 85 | 64 | 49 | 16 |

(b) Slice+P+RO+MADJ+EG

Figure B.2: Median complement sizes of the 10,998 effective samples of the external tests without the Rank construction. The rows (1.0 to 3.0) are the transition densities, and the columns (0.1 to 1.0) are the acceptance densities.

# Appendix C

# Execution Times

| Construction | Mean | Min. | P25 | Median | P75 | Max. | Total | ≈ hours |
|---|---|---|---|---|---|---|---|---|
| Fribourg | 8.5 | 2.5 | 3.3 | 4.9 | 7.3 | 586.0 | 93,351.2 | 259 |
| Fribourg+R2C | 6.6 | 2.2 | 2.9 | 4.2 | 6.4 | 219.7 | 72,545.7 | 202 |
| Fribourg+R2C+C | 8.5 | 2.2 | 2.6 | 3.5 | 6.4 | 582.9 | 93,396.2 | 259 |
| Fribourg+M1 | 4.9 | 2.5 | 3.2 | 4.1 | 5.9 | 55.1 | 54,061.3 | 150 |
| Fribourg+M1+M2 | 4.6 | 2.2 | 2.9 | 3.8 | 5.1 | 38.4 | 49,848.0 | 138 |
| Fribourg+M1+R2C | 4.4 | 2.2 | 2.8 | 3.6 | 5.3 | 42.5 | 48,572.0 | 135 |
| Fribourg+M1+R2C+C | 5.6 | 2.5 | 3.2 | 4.0 | 6.5 | 147.4 | 60,918.9 | 169 |
| Fribourg+R | 7.5 | 2.2 | 3.0 | 3.9 | 6.3 | 470.5 | 82,387.3 | 229 |

Table C.1: Execution times in CPU time seconds for the 10,939 effectvie samples of the GOAL test set.

| Construction | Mean | Min. | P25 | Median | P75 | Max. | Total | ≈ hours |
|---|---|---|---|---|---|---|---|---|
| Piterman+EQ+RO | 3.0 | 2.2 | 2.6 | 2.8 | 3.0 | 42.9 | 21,410.6 | 59 |
| Slice+P+RO+MADJ+EG | 3.7 | 2.2 | 2.7 | 3.2 | 4.1 | 36.7 | 26,398.9 | 73 |
| Rank+TR+RO | 16.0 | 2.3 | 2.8 | 3.7 | 9.3 | 443.3 | 115,563.9 | 321 |
| Fribourg+M1+R2C | 4.0 | 2.2 | 2.7 | 3.1 | 4.4 | 410.4 | 28,970.8 | 80 |

Table C.2: Execution times in CPU time seconds for the 7,204 effectvie samples of the GOAL test set.

| Construction | Mean | Min. | P25 | Median | P75 | Max. | Total | ≈ hours |
|---|---|---|---|---|---|---|---|---|
| Piterman+EQ+RO | 3.6 | 2.2 | 2.7 | 2.9 | 3.4 | 365.7 | 39,663.4 | 110 |
| Slice+P+RO+MADJ+EG | 4.3 | 2.2 | 2.9 | 3.7 | 5.0 | 42.4 | 47,418.2 | 132 |
| Fribourg+M1+R2C | 4.7 | 2.2 | 2.8 | 3.6 | 5.3 | 410.4 | 52,149.0 | 145 |

Table C.3: Execution times in CPU time seconds for the 10,998 effectvie samples of the GOAL test set without the Rank construction.

| Construction | Michel 1 | Michel 2 | Michel 3 | Michel 4 | Fitted curve | Std. error |
|---|---|---|---|---|---|---|
| Fribourg | 2.3 | 4.0 | 88.8 | 100,976.0 | $(1.14n)^n$ | 0.64% |
| Fribourg+R2C | 2.3 | 3.4 | 27.4 | 27,938.3 | $(0.92n)^n$ | 0.64% |
| Fribourg+M1 | 2.2 | 3.6 | 17.9 | 6,508.4 | $(0.72n)^n$ | 0.63% |
| Fribourg+M1+M2 | 2.3 | 3.5 | 13.8 | 2,707.4 | $(0.62n)^n$ | 0.62% |
| Fribourg+M1+M2+R2C | 2.5 | 3.5 | 10.8 | 2,332.6 | $(0.61n)^n$ | 0.62% |
| Fribourg+R | 2.4 | 3.7 | 86.0 | 101,809.6 | $(1.14n)^n$ | 0.64% |

Table C.4: Execution times in CPU time seconds for the four Michel automata.

| Construction | Michel 1 | Michel 2 | Michel 3 | Michel 4 | Fitted curve | Std. error |
|---|---|---|---|---|---|---|
| Piterman+EQ+RO | 2.5 | 3.8 | 42.6 | 75,917.4 | $(1.08n)^n$ | 0.64% |
| Slice+P+RO+MADJ+EG | 2.3 | 3.6 | 11.4 | 159.5 | $(0.39n)^n$ | 0.38% |
| Rank+TR+RO | 2.2 | 3.0 | 6.4 | 30.0 | $(0.29n)^n$ | 0.18% |
| Fribourg+M1+M2+R2C | 2.5 | 3.5 | 10.8 | 2,332.6 | $(0.61n)^n$ | 0.62% |

Table C.5: Execution times in CPU time seconds for the four Michel automata.

# Bibliography


[1] J. Allred, U. Ultes-Nitsche. Complementing Büchi Automata with a Subset-Tuple Construction. Tech. rep.. University of Fribourg, Switzerland. 2014.

[2] C. Althoff, W. Thomas, N. Wallmeier. Observations on Determinization of Büchi Automata. In J. Farré, I. Litovsky, S. Schmitz, eds., *Implementation and Application of Automata*. vol. 3845 of *Lecture Notes in Computer Science*. pp. 262–272. Springer Berlin Heidelberg. 2006.

[3] S. Breuers, C. Löding, J. Olschewski. Improved Ramsey-Based Büchi Complementation. In L. Birkedal, ed., *Foundations of Software Science and Computational Structures*. vol. 7213 of *Lecture Notes in Computer Science*. pp. 150–164. Springer Berlin Heidelberg. 2012.

[4] J. R. Büchi. On a Decision Method in Restricted Second Order Arithmetic. In *Proc. International Congress on Logic, Method, and Philosophy of Science, 1960*. Stanford University Press. 1962.

[5] S. J. Fogarty, O. Kupferman, T. Wilke, et al. Unifying Büchi Complementation Constructions. *Logical Methods in Computer Science*. 9(1). 2013.

[6] C. Göttel. Implementation of an Algorithm for Büchi Complementation. BSc Thesis, University of Fribourg, Switzerland. November 2013.

[7] J. E. Hopcroft, R. Motwani, J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley. 2nd edition ed.. 2001.

[8] D. Kähler, T. Wilke. Complementation, Disambiguation, and Determinization of Büchi Automata Unified. In L. Aceto, I. Damgård, L. Goldberg, et al, eds., *Automata, Languages and Programming*. vol. 5125 of *Lecture Notes in Computer Science*. pp. 724–735. Springer Berlin Heidelberg. 2008.

[9] J. Klein. Linear Time Logic and Deterministic Omega-Automata. *Master's thesis, Universität Bonn*. 2005.

[10] J. Klein, C. Baier. Experiments with Deterministic $\omega$-Automata for Formulas of Linear Temporal Logic. In J. Farré, I. Litovsky, S. Schmitz, eds., *Implementation and Application of Automata*. vol. 3845 of *Lecture Notes in Computer Science*. pp. 199–212. Springer Berlin Heidelberg. 2006.

[11] O. Kupferman, M. Y. Vardi. Weak Alternating Automata Are Not that Weak. *ACM Trans. Comput. Logic*. 2(3):pp. 408–429. Jul. 2001.

[12] R. Kurshan. Complementing Deterministic Büchi Automata in Polynomial Time. *Journal of Computer and System Sciences*. 35(1):pp. 59 – 71. 1987.

[13] C. Löding. Optimal Bounds for Transformations of $\omega$-Automata. In C. Rangan, V. Raman, R. Ramanujam, eds., *Foundations of Software Technology and Theoretical Computer Science*. vol. 1738 of *Lecture Notes in Computer Science*. pp. 97–109. Springer Berlin Heidelberg. 1999.

[14] R. McNaughton. Testing and generating infinite sequences by a finite automaton. *Information and Control*. 9(5):pp. 521 – 530. 1966.

[15] M. Michel. Complementation is more difficult with automata on infinite words. *CNET, Paris*. 15. 1988.

[16] A. Mostowski. Regular expressions for infinite trees and a standard form of automata. In A. Skowron, ed., *Computation Theory*. vol. 208 of *Lecture Notes in Computer Science*. pp. 157–168. Springer Berlin Heidelberg. 1985.

[17] D. E. Muller. Infinite Sequences and Finite Machines. In *Switching Circuit Theory and Logical Design, Proceedings of the Fourth Annual Symposium on*. pp. 3–16. Oct 1963.

[18] D. E. Muller, P. E. Schupp. Simulating Alternating Tree Automata by Nondeterministic Automata: New Results and New Proofs of the Theorems of Rabin, McNaughton and Safra. *Theoretical Computer Science*. 141(1–2):pp. 69 – 107. 1995.

[19] F. Nießner, U. Nitsche, P. Ochsenschläger. Deterministic Omega-Regular Liveness Properties. In S. Bozapalidis, ed., *Preproceedings of the 3rd International Conference on Developments in Language Theory, DLT'97*. pp. 237–247. Citeseer. 1997.

[20] N. Piterman. From Nondeterministic Buchi and Streett Automata to Deterministic Parity Automata. *Logical Methods in Computer Science*. 3(5):pp. 1–21. 2007.

[21] M. Rabin, D. Scott. Finite Automata and Their Decision Problems. *IBM Journal of Research and Development*. 3(2):pp. 114–125. April 1959.

[22] M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*. 141:pp. 1–35. July 1969.

[23] M. Roggenbach. Determinization of Büchi-Automata. In E. Grädel, W. Thomas, T. Wilke, eds., *Automata Logics, and Infinite Games*. vol. 2500 of *Lecture Notes in Computer Science*. pp. 43–60. Springer Berlin Heidelberg. 2002.

[24] S. Safra. On the Complexity of Omega-Automata. In *Foundations of Computer Science, 1988., 29th Annual Symposium on*. pp. 319–327. Oct 1988.

[25] S. Schewe. Büchi Complementation Made Tight. In *26th International Symposium on Theoretical Aspects of Computer Science-STACS 2009*. pp. 661–672. 2009.

[26] A. P. Sistla, M. Y. Vardi, P. Wolper. The Complementation Problem for Büchi Automata with Applications to Temporal Logic. *Theoretical Computer Science*. 49(2–3):pp. 217 – 237. 1987.

[27] R. S. Streett. Propositional dynamic logic of looping and converse is elementarily decidable. *Information and Control*. 54(1–2):pp. 121 – 141. 1982.

[28] W. Thomas. Automata on Infinite Objects. In J. van Leeuwen, ed., *Handbook of Theoretical Computer Science (Vol. B)*. chap. Automata on Infinite Objects, pp. 133–191. MIT Press, Cambridge, MA, USA. 1990.

[29] W. Thomas. Languages, Automata, and Logic. In G. Rozenberg, A. Salomaa, eds., *Handbook of Formal Languages*. pp. 389–455. Springer Berlin Heidelberg. 1997.

[30] W. Thomas. Complementation of Büchi Automata Revisited. In J. Karhumäki, H. Maurer, G. Păun, et al, eds., *Jewels are Forever*. pp. 109–120. Springer Berlin Heidelberg. 1999.

[31] M.-H. Tsai, S. Fogarty, M. Vardi, et al. State of Büchi Complementation. In M. Domaratzki, K. Salomaa, eds., *Implementation and Application of Automata*. vol. 6482 of *Lecture Notes in Computer Science*. pp. 261–271. Springer Berlin Heidelberg. 2011.

[32] M.-H. Tsai, Y.-K. Tsay, Y.-S. Hwang. GOAL for Games, Omega-Automata, and Logics. In N. Sharygina, H. Veith, eds., *Computer Aided Verification*. vol. 8044 of *Lecture Notes in Computer Science*. pp. 883–889. Springer Berlin Heidelberg. 2013.

[33] Y.-K. Tsay, Y.-F. Chen, M.-H. Tsai, et al. Goal: A Graphical Tool for Manipulating Büchi Automata and Temporal Formulae. In O. Grumberg, M. Huth, eds., *Tools and Algorithms for the Construction and Analysis of Systems*. vol. 4424 of *Lecture Notes in Computer Science*. pp. 466–471. Springer Berlin Heidelberg. 2007.

[34] Y.-K. Tsay, Y.-F. Chen, M.-H. Tsai, et al. Goal Extended: Towards a Research Tool for Omega Automata and Temporal Logic. In C. Ramakrishnan, J. Rehof, eds., *Tools and Algorithms for the*

*Construction and Analysis of Systems*. vol. 4963 of *Lecture Notes in Computer Science*. pp. 346–350. Springer Berlin Heidelberg. 2008.

[35] Y.-K. Tsay, Y.-F. Chen, M.-H. Tsai, et al. Tool support for learning Büchi automata and linear temporal logic. *Formal Aspects of Computing*. 21(3):pp. 259–275. 2009.

[36] Y.-K. Tsay, M.-H. Tsai, J.-S. Chang, et al. Büchi Store: An Open Repository of Büchi Automata. In P. Abdulla, K. Leino, eds., *Tools and Algorithms for the Construction and Analysis of Systems*. vol. 6605 of *Lecture Notes in Computer Science*. pp. 262–266. Springer Berlin Heidelberg. 2011.

[37] U. Ultes-Nitsche. A Power-Set Construction for Reducing Büchi Automata to Non-Determinism Degree Two. *Information Processing Letters*. 101(3):pp. 107 – 111. 2007.

[38] M. Vardi. An automata-theoretic approach to linear temporal logic. In F. Moller, G. Birtwistle, eds., *Logics for Concurrency*. vol. 1043 of *Lecture Notes in Computer Science*. pp. 238–266. Springer Berlin Heidelberg. 1996.

[39] M. Y. Vardi, T. Wilke. Automata: From Logics to Algorithms. In J. Flum, E. Grädel, T. Wilke, eds., *Logic and Automata: History and Perspectives*. vol. 2 of *Texts in Logic and Games*. pp. 629–736. Amsterdam University Press. 2007.

[40] T. Wilke. $\omega$-Automata. In J.-E. Pin, ed., *Handbook of Automata Theory*. European Mathematical Society. To appear, 2015.

[41] Q. Yan. Lower Bounds for Complementation of $\omega$-Automata Via the Full Automata Technique. In M. Bugliesi, B. Preneel, V. Sassone, et al, eds., *Automata, Languages and Programming*. vol. 4052 of *Lecture Notes in Computer Science*. pp. 589–600. Springer Berlin Heidelberg. 2006.