Empirical Performance Investigation of a Büchi Complementation Construction

Daniel Weibel

August 6, 2015

Abstract

This will be the abstract.

Acknowledgements

Contents

Chapter 1

Introduction

At the beginning of the 1960's, a Swiss logician named Julius Richard Büchi was looking for a way to decide the satisfiability of formulas of the monadic second order logic with one successor (S1S). In his quest, Büchi observed that an S1S formula can be represented by a certain type of finite state automaton that runs on infinite words, such that this automaton accepts a word if and only if the corresponding interpretation satisfies the formula. The proof of this equivalence between S1S formulas and this type of autoamtaon, which is known as *Büchi's Theorem*, led Büchi to his desired decision procedure: to test whether an S1S formula φ is satisfiable, translate it to an equivalent automaton A, and test whether A is empty (that is, accepts no words at all). If A is empty, then φ is unsatisfiable, if A is non-empty, then φ is satsifiable. [8]

The type of automaton that Büchi used for solving this logical problem is called *Büchi automaton.* The application of Büchi automata to logic, that was established by Büchi, had a large impact on other fields, especially model checking, which is a technique of formal verification. In particular, Büchi automata allow to solve the model checking question automata-theoretically, which has many advantages [60].

However, there is one operation on Büchi automata that is giving a "headache" to the research community since the introduction of Büchi automata more than 50 years ago, namely the problem of *complementation.* Algorithms for carrying out this operation, although possible[1], turn out to be very complex, in many cases too complex for practical application. Yet, Büchi complementation has a practical application in the automata-theoretic approach to model checking. This discrepancy led to an ongoing quest for finding more efficient *Büchi complementation constructions*, and generally better understanding the complexity of Büchi complementation. The work in this thesis is situated in this area of research.

In this introductory chapter, we will first

1.1 Context

1.1.1 Büchi Automata and Büchi Complementation

Büchi automata are finite state automata that process words of infinite length, so called ω-words. If Σ is the alphabet of a Büchi automaton, then the set of all the possible ω-words that can be generated from this alphabet is denoted by Σ^ω. A word $\alpha \in \Sigma^\omega$ is accepted by a Büchi automaton if it results in at least one run that contain infinitely many occurrences of at least one accepting state. A run of a Büchi automaton on a word is an infinite sequence of states. Deterministic Büchi automata have at most one run for each word in Σ^ω, whereas non-deterministic Büchi automata may have multiple runs for a word.

The complement of a Büchi automaton A is another Büchi automaton[2] denoted by $\overline{A}$. Both, A and $\overline{A}$, share the same alphabet Σ. Regarding a word $\alpha \in \Sigma^\omega$, the relation between an automaton and its complement is as follows:

$$\alpha \text{ accepted by } A \Longleftrightarrow \alpha \text{ not accepted by } \overline{A}$$

That is, all the words of Σ^ω that are *accepted* by an automaton are *rejected* by its complement, and all the words of Σ^ω that are *rejected* by an automaton are *accepted* by its complement. In other words, there is no single word of Σ^ω that is accepted or rejected by *both* of an automaton and its complement.

A Büchi complementation construction is an algorithm that, given a Büchi automaton, creates the complement of this Büchi automaton. The difficulty of this operation depends on whether the input automaton is determinstic or non-deterministic. The complementation of deterministic Büchi automata is "easy" and can be done in polynomial time and linear space [26]. The complementation of non-deterministic Büchi automata, however, is very complex. The understanding and reduction of its complexity is a domain of active research and lies at the centre of this thesis.

Consequently, when in the following we talk about "Büchi complementation", we specifically mean the complementation of *non-determinstic* Büchi automata. The main problem with the complexity of Büchi complementation is the so-called state growth or state complexity (sometimes also called state blow-up or

[1] Büchi himself has proved that Büchi automata are closed under complementation [8].

[2] Büchi automata are closed under complementation. This has been proved by Büchi [8], who, to this end, described the first Büchi complementation construction in history.

state explosion). This is the number of states of the output automaton in relation to the number of states of the input automaton. In simple words, Büchi complementation constructions produce complements that may be very, very large.

This inhibits the practial applicaion of Büchi complementation, because in this case the limited computing and time resources may not be high enough to accommodate for this high complexity. In the following subsections we highlight an important application that Büchi complementation has in practice, and thereby motivate the research on Büchi complementation and of this thesis.

1.1.2 Applications of Büchi Complementation

Language Containment of ω-Regular Languages

Büchi complementation is used for testing language containment of ω-regular languages. The ω-regular languages are the class of formal languages that is equivalent to non-deterministic Büchi automata[3]. At this point, we briefly describe the language containment in general, before in turn describing an application of the language containment problem in the next subsection.

Given two ω-regular languages L_1 and L_2 over alphabet Σ^ω the language containment problem consists in testing whether $L_1 \subseteq L_2$. This is true if all the words of L_1 are also in L_2, and false if L_1 contains at least one word that is not in L_2. The way this problem is algorithmically solved is by testing $L_1 \cap \overline{L_2} = \varnothing$. Here, $\overline{L_2}$ denotes the complement language of L_2, which means $\overline{L_2}$ contains all the words of Σ^ω that are *not* in L_2. The steps for testing $L_1 \cap \overline{L_2} = \varnothing$ are the following:

- Create the complement language $\overline{L_2}$ of L_2
- Create the intersection language $L_{1,\overline{2}}$ of L_1 and $\overline{L_2}$
- Test whether $L_{1,\overline{2}}$ is empty (that is, contains no words at all)

Thus, the language containment problem is reduced to three operations on languages, *complementation*, *intersection*, and *emptiness testing*. The common way to work with formal languages is not to handle the languages themselves, but more compact structures that represent them, such as automata. In the case of ω-regular languages, these are non-deterministic Büchi automata.

For solving $L_1 \subseteq L_2$, one thus works with two Büchi automata A_1 and A_2 that represent L_1 and L_2, respectively. The problem then becomes $L(A_1) \subseteq L(A_2)$, and equivalently, $L(A_1) \cap \overline{L(A_2)} = \varnothing$. This is automata-theoretically solved as $\mathsf{empty}(A_1 \cap \overline{A_2})$, which includes the three following steps:

- Construct the complement automaton $\overline{A_2}$ of A_2
- Construct the intersection automaton $A_{1,\overline{2}}$ of A_1 and A_2
- Test whether $A_{1,\overline{2}}$ is empty (that is, accepts no words at all)

If the final emptiness test on automaton $A_{1,\overline{2}}$ is true, then $L_1 \subseteq L_2$ is true, and if the emptiness test is false, then $L_1 \subseteq L_2$ is false. In this way, the language containment problem of ω-regular languages is reduced to three operations of *complementation*, *intersection*, and *emptiness testing* of non-deterministic Büchi automata. Thus, Büchi complementation is an intergral part of language contaiment of ω-regular languages.

Automata-Theoretic Model Checking via Language Containment

In the last subsection, we have seen that Büchi complementation is used for testing language containment of ω-regular languages. In this subsection, we will see what in turn language containment of ω-regular languages is used for. To this end, we describe one important application of it, namely the language containment approach to automata-theoretic model checking. In the following, we first describe basic working of this technique in general, and then point out the significance that Büchi complementation bears for it.

[3]Note that deterministic Büchi automata have a lower expressivity than non-deterministic Büchi automata, and are equivalent to only a subset of the ω-regular languages.

Basics

The language containment approach to automata-theoretic model checking is an approach to automata-theoretic model checking, which is an approach to model checking, which in turn is an approch to formal verification [61]. Figure 1.1 shows the branch of the family of formal verification techniques that contains the language containment approach to automata-theoretic model checking.

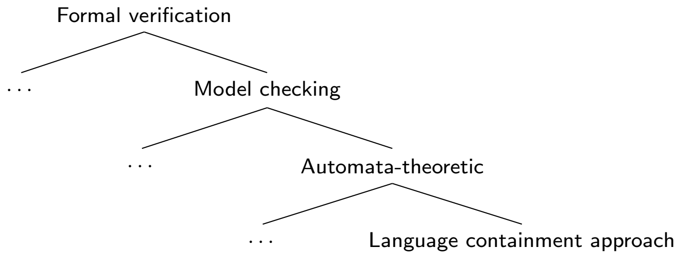


Figure 1.1: Branch of the family of formal verification techniques that contains the language containment approach to automata-theoretic model checking.

Formal verification is the use of mathematical techniques for proving the correctness of a system (software of hardware) with respect to a specified property [61]. A typical example is to verify that a program is deadlock-free (in which case the property would be "deadlock-freeness"). In general, formal verification techniques consist of the following three parts [19]:

1. A framework for modelling the system to verify
2. A framework for specifying the property to be verified
3. A verification method for testing whether the system satisfies the property

For the language containment approach to automata-theoretic model checking, the frameworks for points 1 and 2 are Büchi automata representing ω-regular languages. The verification method (point 3) is testing language containment of the system automaton's language in the property automaton's language. In some more detail, this works as follows. [60][61]

The system s to verify is modelled as a Büchi automaton, say S. This Büchi automaton represents an ω-regular language $L(S)$, and each word of $L(S)$ represents in turn a possible *computation trace* of the system. A computation trace is an infinite[4] sequence of "situations" that the system is in at any point in time. Such a situation consists of a finite amount of information of, for example, the values of variables, registers, or buffers. The observation that such a trace can be represented as a word of an ω-regular langauges comes from the fact that it can be represented intuitively as a linear Kripke structure (which in turn is an interpretation for a temporal logic formula that can also be used to represent computations), which in turn can be represented by a word of a language whose alphabet is ranges over the powerset of the atomic propositions of the Kripke strucutre. A work that explains these intimate relations between computation, temporal logic, formal languages, and automata in more detail is [60]. In simple words, the language $L(S)$, represented by the system automaton S, represents everything that the system *can* do.

Similarly, a property p to be verified is represented as a Büchi automaton, say P, which represents the ω-regular language $L(P)$, whose words represent computation traces. These computation traces are all the computations of a system like s that satisfy the property p. If for example p is "deadlock-freeness", then the words of $L(P)$ represent all the possible computation traces that do *not* contain a deadlock. In this way, the language $L(P)$ represents everything that the system is *allowed* to do, with respect to a certain property.

The verification step is finally done by testing $L(S) \subseteq L(P)$. If this is true, then everything that the system *can* do is *allowed* to do, and the system satisfies the property p. If the language containment test

[4] The infinity of computation traces suggests that this type of formal verification (and model checking in general) is used for systems that are not expected to terminate and may run indefinitely. This type of systems is called *reactive* systems. They contrast with systems that are expected to terminate and produce a result. For this latter type of systems other formal verification techniques than model checking are used. See for example [19] and [5] for works that cover the formal verification of both types of systems.

is false, then the system *can* do a computation that is *not allowed* to do, and the system does not satisify the property p.

Summarising, the language containment approach to automata-theoretic model checking requires language containment of ω-regular languages, which, as we have seen, requires Büchi complementation. In the following, we will highlight the particular importance of Büchi complementation for this type of formal verification.

1.1.3 Significance of Büchi Complementation

The complexity of Büchi complementation makes the just described model checking approach nearly inapplicable in practice [49]. According to [61], there are so far no tools that include this approach. This is unfortunate, because the other Büchi operations for language containment, intersection and emptiness testing, have highly efficient solutions [61] (cf. [60]), and thus Büchi complementation is the only bottleneck. Existing practical applications are thus forced to circumvent the need for Büchi complementation. This is possible, has however certain disadvantages as we will see in the following.

One way to circumvent the complementation of non-deterministic Büchi automata is to specify the property as a deterministic Büchi automaton [49][61]. As we have mentioned, the complementation of deterministic Büchi automata has an efficient solution. The disadvantage of this approach is, however, that the property automaton may become exponentially larger, and that it is generally more complicated and less intuitive to represent a language as a deterministic automaton [49].

Another way is to use a different model checking approach altogether, which leads us back to the essence of model checking. In basic model checking, the property to be verified is represented as a formula φ of a temporal logic (typically LTL). The system to verify is represnted as a Kripke structure K, which serves as an interpretation of the formula φ. The verification step consists in checking whether K is a *model* of φ. An interpretation K is a model of a formula φ, if every state of the interpretation *satisifes* the formula, written as $K \models \varphi$. This test of modelhood is the reason that this verification approach is called *model checking* [60].

The modelhood test can be done automata-theoretically without the need for Büchi complementation [61] (in Figure 1.1, this would be a sibling to the language containment approach). The Kripke structure K is translated to a Büchi automaton A_K. The formula φ is negated and translated to the Büchi automaton $A_{\neg\varphi}$. Finally, one tests $\mathsf{empty}(A_K \cap A_{\neg\varphi})$. This correponds to teh language containment test $L(K) \subseteq L(\varphi)$, which is equivalent to the modelhood test $K \models \varphi$. The trick is that the complementation of the property, that is required for the language containment test, is pushed off from the complementation of a Büchi automaton to the negation of a temporal logic formula, which is trivial. This approach is used, for example, by the SPIN model checker [17]. The disadvantage is that the typically used temporal logic LTL is less expressive than Büchi automata, and hence the breadth of properties that can be expressed is limited. It has been stated that the expressivity of LTL is unsufficient for industrial verification applications [61].

For more information on model checking, as well as other formal verification techniques, we refer to the following works: [19][5][4].

As can be seen from these elaborations, having efficient procedures for Büchi complementation would be of great practical value. Even though handling the “worst-cases” will forever be unefficient,

1.2 Motivation

In the previous section we have seen that Büchi complementation is complex, and that it would be of practical value to better understand this complexity. In this section, we highlight the need for looking at this complexity in a way that has not received much attention in the past, namely empirically rather than theoretically.

In the following, we first present the traditional way of analysing the worst-case performance of complementation constructions, and then describe the empirical way for investigating their actual peformance.

This includes a review of the work that has been done so far. Note that we are using the terms complexity and performance interchangeably, and they both mean basically state growth.

1.2.1 Theoretical Investigation of Worst-Case Performance

The traditional performance measure for Büchi complementation constructions is their *worst-case state growth*[5]. This is the maximum number of states the construction *can* generate, in relation to the number of states of the input automaton.

For example, the initial complementation construction by Büchi (1962) [8] has a worst-case state growth of $2^{2^{O(n)}}$ does not mean that it produces a larger complement than Schewe's construction, for this concrete example. It might well be smaller. In fact, worst-case state complexities only allow to adequately deduce something about the specific worst-cases, and not about all the other automata. From a practical point of view, these worst cases are however not interesting, as their application is impracticable anyway (at least starting from a certain input automaton size). , where n is the number of states of the input automaton. At this point, two short comments. First, the state growth is often not given as an exact function, but uses the big-O notation. Second, for notating state growths, we will consistenly use the variable n, whose meaning is the number of states of the input automaton. This means, for example, that for an input automaton with 8 states, the maximum number of states that the output automaton of Büchi's construction can have is 1.16×10^{77} (if assuming the concrete function 2^{2^n}).

Different constructions exhibit different worst-case state growths, and one of the main objectives of construction creators is to reduce this number. For example, the much more recent construction by Schewe (2009) [44] has a worst-case state growth as low as $(0.76n)^n + n^2$. Given an input automaton with 8 states, the maximum number of states of the output automaton is approximately 119.5 million.

A related objective of research is the quest for the theoretical worst-case state growth of Büchi complementation *per se*. A first result of $n!$ has been proposed in 1988 by Michel [29]. He proved that there exists a family of automata whose complement *cannot* have less than $n!$ states (these automata are known as Michel automata, and we will use them as part of the test data for our experiments). This proves a *lower bound* for the fundamental worst-case complexity of Büchi complementation, as it is not known whether the Michel automata are the real worst cases, or if there are even worse cases. Indeed, in 2007, Yan [67] proved a new higher lower bound of $(0.76n)^n$ (Michel's $n!$ corresponds to approximately $(0.36n)^n$ [67]). The worst-case state growth of a concrete construction naturally serves as an *upper bound* to a known lower bound. Given Schewe's number $(0.76n)^n n^2$, the lower bound of $(0.76n)^n$ by Yan is regarded as "sharp", as the gap between the lower and upper bound is very narrow, and consequently, the lower bound canot rise much anymore.

Many construction developers aim at bringing the worst-case state growth of their construction close to the currently known lower bound. It goes so far that a construction matching this lower bound is regarded as "optimal".

1.2.2 Need for Empirical Investigation of Actual Performance

Worst-case state growths are interesting from a theoretical point of view, but they are poor guides to the actual performance of a construction [53]. For example, if we have a concrete automaton of, say, 15 states, and we complement it with Schewe's construction, the fact that the worst-case state complexity is $(0.76n)^n n^2$ does not reveal anything about how the construction will perform on this concrete automaton. In any case, we are not expecting the complement to have 1.6 quintillion (1.6×10^{18}) states (which would be the worst case), because this would most likely be practically infeasible.

Furthermore, if a construction has a higher worst-case state growth than another, it does not mean that it performs worse on a concrete case. In fact, worst-case state complexities only allow to adequately deduce the performance on the worst-case automata, but not on all the other automata. However, from a practical point of view, these worst cases are not interesting, as their application in practice is anyway infeasible [49] (at least starting from a certain input automaton size).

[5] As mentioned previously, also known as state complexity, state blow-up, or state explosion.

From a practical perspective we are interested how constructions perform on automata as they could occur in a concrete application of Büchi complementation, such as automata-theoretic model checking. This may include questions like the following. What is a reasonable complement size to expect for the given automaton with n states? Are there generally easier and harder automata? What are the factors that make an automaton especially easy or hard? How does the performance of different constructions on the same automata vary? Are there constructions which are better suited for a certain type of automata than other constructions?

Questions like this can be attempted to answer by empirical peformance investigations. As its two most important elements this includes an *implementation* of the investigated constructions and *test data*. With test data, we mean a set of concrete automata on which the implementations of the constructions are run. The analysis is then done on the generated complement automata.

There have been relatively few empirical attempts in the history of Büchi complementation [53], compared to the number of theoretical works. In the following, we give (non-exhaustive) list of empirical works in the past that illustrate the approach, and also show the line of research in which the work of this thesis is situated.

1995 Tasiran et al. [49] create an efficient implementation of Safra' construction[42] (determinisation-based) and used it for for automata-theoretic model checking tasks with the HSIS verification tool [3]. They state that they could easily complement property automata with some hundreds of states, however, they do not provide a statistical evaluation of the results.

2003 Gurumurthy et al. [16] implement Friedgut, Kupferman, and Vardi's construction [25] (rank-based) along with various optimisations that they propose as a part of the tool Wring [47]. They complement 1000 small automata, generated by translation from LTL formulas, and evaluate exeuction time, and number of states and transitions of the complement for the different versions of the construction.

2006 Althoff et al. [2] implement Safra's [42] and Muller and Schupp's [34] determinisation constructions[6] in a tool called OmegaDet, applied them on the Michel automata with 2 to 6 states, and compared the number of states of the determinised output automata.

2008 Tsay et al. [56] carry out a first comparative experiment with the publicly available[7] GOAL tool [55][56][57][54]. They include the constructions by Safra [42] (determinisation-based), Piterman [37] (determinisation-based), Thomas [52] (WAPA[8]), and Kupferman and Vardi[25] (rank-based or WAA[9]). These constructions are pre-implemented in GOAL. As the test data, they use 300 Büchi automata, translated from LTL formulas, with an average size of 5.4 states. They evaluate and compare execution times, as well as number of states and transitions of the complements.

2009 Kamarkar and Chakraborty [21] propose an improvement of Schewe's construction [44] (rank-based) and implement it, as well as Schewe's original construction, on top of the model checker NuSMV [10][9]. They run the constructions on 12 test automata and compare the sizes of the complements. Furthermore, they run the same tests with the constructions by Kupferman, and Vardi [25] (rank-based or WAA) and Piterman [37] (determinisation-based) within GOAL, and compare the results to the ones of their implementation of Schewe's construction.

2010 Tsai et al. [53] (paper entitled "State of Büchi Compelentation") carry out another experiment with GOAL. They compare the constructions by Piterman [37] (determinisation-based), Schewe [44] (rank-based), and Vardi and Wilke [64] (slice-based), with various optimsiations that they propose in the same paper. As the test data, they use 11,000 randomly generated automata with 15 states and an alphabet size of 2. The test set is organised into 110 automata classes that consist of the combinations of 11 transition densities and 10 accceptance densities. This test set is repeatedly used in subsequent work (including in this thesis), and we will refer to it as the GOAL test set (because it has been generated with the GOAL tool). Tsai et al. provide sophisticated evaluation of the states of the complements for all the tested constructions and construction versions.

[6]These determinisation constructions transform a non-deterministic Büchi automaton to a deterministic Rabin automataon (see Section ??), however, the are used as the base for determinisation-based complementation constructions.

[7]http://goal.im.ntu.edu.tw/wiki/doku.php

[8]Via **W**eak **A**lternating **P**arity **A**utomaton

[9]Via **W**eak **A**lternating **A**utomaton

2010 Breuers [6] proposes an improvement for the construction by Sistla, Vardi, and Wolper [46] (Ramsey-based), and creates an implementation of it. He generates his own test data (inspired by the work of Tsai et al. [53]) consisting of *easy*, *medium*, and *hard* automata, based on different transition density and acceptance density values. He evaluates the complement sizes produced by the construction for autoamta of sizes 5, 10, and 15 of all these difficulty categories.

2012 Breuers et al. [7] wrap the implementation of their improvement of Sistla, Vardi, and Wolper's construction [46] in the publicly available tool Alekto[10], and and run it on the GOAL test set. They compare the generated complement sizes, as well as the number of aborted complementation tasks (due to exceeding resource requirements) to the corresponding result for different constructions on the same test set by Tsai et al. [53].

2013 Göttel [14] creates a C implementation of the Fribourg construction [1], including the R2C optimisation (see Chapter 3), and executes it on the GOAL test set, as well as on the Michel automata with 3 to 6 states. He analyses the resulting complement sizes and execution times separately for each of the 110 classes that the GOAL test set consists of. The Fribourg construction[11] is a slice-based complementation construction that is being developed at the university of Fribourg, and which lies at the heart of this thesis. The entire Chapter 3 of this thesis is dedicated to explaining the Fribourg construction.

1.3 Aim and Scope

The aim of this thesis is an in-depth empirical performance investigation of the Fribourg construction. As mentioned, the Fribourg construction is a Büchi complementation construction that is being developed at the University of Fribourg [1]. By empirically investigating the behaviour of this specific construction, we want to follow up the track of empirical research that we have outlined in the last section.

This thesis is certainly not sufficient to describe the performance of the Fribourg construction in its entiretey, or in a way that is adequate to be relied on in industiral applications. Neither this thesis can answer general questions about the observed behaviour of Büchi complementation. Rather, we see this piece of work as a mosaic stone that we add to the very complex and multi-faceted picture of the complexity of Büchi complementation.

The empirical performance investigation will include testing of different versions of the construction, and comparison with other complementation constructions...

Aim: empirical performance investigation of a specific Büchi complementaiton construction, comparison with other constructions

Scope: two test sets, relatively small automata, no real world or "typical" examples,

1.4 Overview

[10] http://www.automata.rwth-aachen.de/research/Alekto/

[11] The authors of the constructions use the name *subset-tuple construction* (see [1]), however, in this thesis, we will use the name *Fribourg construction*.

Chapter 2

Background

Contents

In this chapter we treat several topics that serve as a background for the rest of the thesis. In particular, the goal of this chapter is to set the stage for our description of the Fribourg construction in Chapter 3, and the setup of our empirical performance investigation of the Fribourg construction in Chapter 4, as well as its results in Chapter 5.

In Section 2.1 we summarise some aspects about Büchi automata, as well as about other types of automata on infinite words, that are relevant to the purpose of Büchi complementation. In Section 2.2, we describe the principal techniques for run analysis of non-deterministic automata. This topic has a particular relation to Büchi complementation, and one of the run analysis techniques, reduced split trees, is the core of the Fribourg construction. In Section 2.3, we provide a review of proposed Büchi complementation constructions from the introduction of Büchi automata in 1962 until today. We organise the presentation of these constructions along the four main Büchi complementation approaches Ramsey-based, determinisation-based, rank-based, and slice-based. Some of the constructions that we describe in this section will be used in the performance investigation of the Fribourg construction in Chapter 4.

2.1 Büchi Automata and Other Omega-Automata

Büchi automata are a type of ω-automata. These are fninite state automata that run on infinite words (so-called ω-words). Externally, ω-automata look the same as the traditional finite state automata on finite words. It is possible to interpret any such automaton on finite words as an ω-automaton, and vice versa.

The difference between ω-automata and automata on finite words is their acceptance condition. An automaton on finite words accepts a word, if after finishing reading it, the automaton is in an accepting state. For ω-automata, this acceptance condition is not possible, because an ω-automaton never finishes reading a word (because the word "never ends"). Instead, the acceptance condition of ω-automata is defined on the set of the so-called *infinitely recurring states*. We are going to describe this concept in Subsection 2.1.1 below.

In Subsection 2.1.1 of this section, wer first treat Büchi automata, and in Subsection 2.1.2 the principal other types of ω-automata, in particular, Muller, Rabin, Streett, and parity automata. In the latter subsection, we also introduce a shorthand notation for different types of ω-automata that we will use throughout the thesis.

Note that in the entire section we omit overly formal notation and proofs of any kind, because it is not necessary for the aim of this thesis. More comprehensive and formally rigorous treatments of ω-automata can be found, for example, in the works by Thomas [50, 51], or Wilke [65].

2.1.1 Büchi Automata

Below we summarise the aspects of Büchi automata that are most significant for our purposes, including the acceptance condition of Büchi automata, the expressivity of non-deterministic and deterministic Büchi automata, and basics about the complementation of non-deterministic and deterministic Büchi automata. In the course of this, we always stress the difference between deterministic and non-deterministic Büchi automata, as this is one of the main sources for the intricacy of Büchi complementation [35].

Definition and Acceptance Condition

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

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

- F: a set of accepting states, $F \subseteq Q$

Note that this is the same definition as for non-deterministic finite state automata on finite words [18]. The difference between Büchi automata and automata on finite words is only the acceptance condition of Büchi automata that we describe below. For deterministic Büchi automata, the definition is similar to the above one, but with a different transition function δ that returns none or a single state, instead of a set of states

An important concept in automata theory the notion of a *run* of an automaton on a given word. A run ρ of automaton A on word α is a sequence of states $qinQ$ that A visits in the process of reading α. For finite words, the length of a run is finite as well, however, for ω-words, the length of a run may be infinite. Note that deterministic automata have at most one run for a given word, whereas non-deterministic automata may have multiple possible runs for the same word.

The Büchi acceptance condition decides whether a run ρ is accepting or non-accepting. This in turn determines the acceptance or non-acceptance of a word: a word α is accepted by an automaton A, if and only if it has at least one accepting run in A. The decision whether a run ρ is accepting or non-accepting is based on the set of *infinitely recurring states* of ρ that we denote by $\mathsf{inf}(\rho)$. This set contains all the states that occur infinitely often in ρ. In particular, the Büchi acceptance condition is as follows:

$$\text{Run } \rho \text{ is accepting} \iff \mathsf{inf}(\rho) \text{ contains at least one accepting state}$$

That is, a run is accepting, if and only if the run contains at least one accepting state infintely often. Formally, this can be written as $\mathsf{inf}(\rho) \cap F \neq \varnothing$.

An intuitive way for describing the Büchi acceptance condition has been given by Vardi [60]. If we imagine the automaton having a green light that blinks whenever the automaton visits an accepting state, then the run is accepting if we observe the green light blinking infinitely many times. The fact that there are only finitely man accepting states, but the light blinks infinitely many times, implies that at least one accepting state is being visited infinitely often.

Expressivity

A particularity of Büchi automata is that deterministic and non-deterministic automata are *not* expressively equivalent. In particular, the class of languages corresponding to the deterministic Büchi automata is a strict subset of the class of languages corresponding to the non-deterministic automata. This result has been proved by Büchi himself in his 1962 paper [8].

This contrasts, for example, with finite state automata on finite words. In this case, non-deterministic and deterministic automata are expressively equivalent, and consequently every non-deterministic automaton can be turned into an equivalent deterministic automaton. With Büchi automata, however, this is not possible, because there exist languages that can be expressed by a non-deterministic Büchi automaton, but not by a deterministic one. An example of such a language is $(0+1)^*1^\omega$, that is, the language of all words of 0 and 1 ending with an infinite sequence of 1. A non-deterministic automaton representing this language, cannot be turned into an equivalent deterministic automaton [60, 41]. Because of this fact, we say that Büchi automata can *in general* not be determinised to other Büchi automata. This fact has implications on the complementation of non-deterministic Büchi automata, as we will see below.

The class of languages that is equivalent to the *non-deterministic* Büchi automata is the class of *ω-regular languages*. A formal description of the ω-regular languages can be found, for example, in [50, 51, 65]. Regarding deterministic Büchi automata, consequently the set of languages that is equivalent to them is a strict subset of the ω-regular languages.

Complementation

Non-deterministic Büchi automata are closed under complementation. This means that the complement of every non-deterministic Büchi automaton is another non-deterministic Büchi automaton. This result has been proved by Büchi in his 1962 paper [8][1]. Deterministic Büchi automata, on the other hand, are not closed under complementation [50]. In particular, this means that the complement of a deterministic Büchi automaton is still a Büchi automaton, however, possibly a non-deterministic one.

As we already mentioned, the algorithmic difficulty and complexity of complementation is very different for deterministic and non-deterministic automata. For deterministic Büchi automata, there exists a simple procedure, introduced in 1987 by Kurshan [26], that can complement a deterministic Büchi automaton to a non-deterministic Büchi automaton in polynomial time and linear space.

For non-deterministic Büchi automata, however, there exists no easy solution. The main reason is that Büchi automata can in general not be determinised. If they could be determinised, then a solution would be to transform a non-deterministic Büchi automaton to an equivalent deterministic one, and complement the deterministic Büchi automaton with Kurshan's construction. This is by the wy the approach that is used for the complementation of non-deterministic automata on finite words: determinise a non-deterministic automaton with the subset construction [38], and then trivially complement the deterministic automaton by making the accepting states non-accepting, and vice versa. Unfortunately, for Büchi automata such a simple procedure is not possible, and this can be seen as the main reason that Büchi complementation is such a hard problem [35].

2.1.2 Other Omega-Automata

After the introduction of Büchi automata in 1962, several other types of ω-automata have been introduced. These automata differ from Büchi automata only in their acceptance condition, that is, in the way they decide whether a run ρ is accepting or non-accepting. All these acceptance condition are however based on the set of infintely recurring states $\mathsf{inf}(\rho)$ of ρ. The following are the most important of these alternative ω-automata along with the year of their introduction.

- Muller automata (1963) [32]
- Rabin automata (1969) [39]
- Streett automata (1982) [48]
- Parity automata (1985) [31]

Some of these automata types are used in complementation constructions, especially in determinisation-based complementation constructions (see Section 2.3). Table 2.1 lists the Muller, Rabin, Streett, and parity acceptance conditions, along with the Büchi acceptance condition for comparison.

Type	Definitions	Condition
Muller	$U \subseteq 2^Q$	$\mathsf{inf}(\rho) \in U$
Rabin	$\{(U_1, V_1), \ldots, (U_r, V_r)\}$, $U_i, V_i \subseteq Q$	$\exists i : \mathsf{inf}(\rho) \cap U_i = \varnothing \wedge \mathsf{inf}(\rho) \cap V_i \neq \varnothing$
Streett	$\{(U_1, V_1), \ldots, (U_r, V_r)\}$, $U_i, V_i \subseteq Q$	$\forall i : \mathsf{inf}(\rho) \cap U_i \neq \varnothing \vee \mathsf{inf}(\rho) \cap V_i = \varnothing$
Parity	$\pi : Q \to \{1, \ldots, k\}$, $k \in \mathbb{N}$	$\mathsf{min}(\{\pi(q) \mid q \in \mathsf{inf}(\rho)\}) \bmod 2 = 0$
Büchi		$\mathsf{inf}(\rho) \cap F \neq \varnothing$

Table 2.1: Acceptance conditions of Muller, Rabin, Streett, parity, and Büchi automata. Note that Q denotes the set of states and F the set of accepting states of the corresponding automaton.

Below, we briefly explain the acceptance conditions of Muller, Rabin, Streett, and parity automata in words. More detailed descriptions of these conditions can be found, for example, in [27], or [65].

Muller acceptance condition Includes a set U of subsets of states. A run ρ is accepting if and only if $\mathsf{inf}(\rho)$ equals one of the pre-defined subsets in U. The Muller acceptance condition is the most general one, and the Rabin, Streett, parity, and Büchi acceptance conditions can be expressed as Muller acceptance conditions [27, 65].

[1] Actually, the proof of closure under complementation of non-deterministic Büchi automata was a necessary step for Büchi to prove the equivalence of Büchi automata and S1S formulas (Büchi's Theorem). In order to prove this closure under complementation, Büchi described the first Büchi complementation construction in history.

Rabin acceptance condition Inludes a list of pairs (U, V) where U and V are subsets of states. A run ρ is accepting if and only if *there is* at least one pair for which U does not contain any states of $\mathsf{inf}(\rho)$ *and* V contains at least one state of $\mathsf{inf}(\rho)$. A pair (U, V) is called a Rabin pair.

Streett acceptance condition Inludes a list of pairs (U, V) where U and V are subsets of states. A run ρ is accepting if and only if *for all* the pairs either U contains at least one state of $\mathsf{inf}(\rho)$ *or* V does not contain any states of $\mathsf{inf}(\rho)$. A pair (U, V) is called a Streett pair. Note that the Streett condition is the dual of the Rabin condition. This means that if we have two identical Rabin and Streett automata with an identical list of pairs, then the Streett automaton accepts the complement language of the Rabin automaton [23].

Parity acceptance condition Assigns a number to each of the states of the automaton. A run ρ is accepting if and only if the smallest-numbered element of $\mathsf{inf}(\rho)$ has an even number. The numbers that are assigned to the states are sometimes called colours [28].

Regarding the expressivity of these automata, it turns out that they are all equivalent to non-deterministic Büchi automata, and thus to the ω-regular languages [65]. This holds for non-determnistic *and* deterministic automata of these types. That means that, unlike Büchi automata, deterministic and non-deterministic Muller, Rabin, Streett, and parity automata are expressively equivalent.

At this point we introdue a notation that we will occasionally use for denoting different types of ω-automata. This notation has been used by Piterman [36] and later by Tsai et al. [53]. It consists of three-letter acronymes of the form

$$\{N, D\} \times \{B, M, R, S, P\} \times W$$

The first letter specifies whether the automaton is non-deterministic (N) or deterministic (D). The second letter stands for the acceptance condition: B for Büchi, M for Muller, R for Rabin, S for Streett, and P for parity. The last letter specifies on which structure the automaton runs. In our case these are always words, thus the last letter is always W. For example, NBW means non-deterministic Büchi automaton, DBW means deterministic Büchi automaton, NMW means non-deterministic Muller automaton, DMW means deterministic Muller automaton, and so on.

2.2 Run Analysis of Non-Deterministic Automata

As mentioned, in a deterministic automaton, every input word has at most one run. In a non-deterministic automaton, however, every input word may have multiple runs. The analysis of the different runs of a non-deterministic automaton on a given input word is called *run analysis* and is central to the Büchi complementation problem.

The reason that run analysis is central to Büchi complementation is as follows. In a Büchi complementation task, we are given a non-deterministic Büchi automaton A, and we attempt to construct its complement B. For constructing B we have in fact to decide for every word α whether B must accept it or not. This decision is intrinsically tied to the entirety of the runs of A on α in the following way:

$$B \text{ accepts } \alpha \iff \textit{All} \text{ the runs of } A \text{ on } \alpha \text{ are } \textit{non-accepting}$$

B must accept a word α, if and only if *all* the runs of A on α are non-accepting. If, for example, A has 10 runs on α, and 9 of them are non-accepting and one is accepting, then A still accepts the word α, and consequently, B must not accept it. Only if all of the 10 runs of A on α are non-accepting, A does not accept α, and consequently B must accept α. This means that for constructing the complement B, we need to consider *all* the runs of the input automaton A on specific words, and the way this can be done is by run analysis on A.

In this section, we present the principal run analysis techniques for non-deterministic Büchi automata [65]. We start with *run DAGs* (DAG stands for directed acyclic graphs) in Section 2.2.1. Then in Sections **??**, 2.2.3, and 2.2.4, we present three techniques based on trees with increasing sophistication, namely *run trees*, *split trees*, and *reduced split trees*. The last of the tree techniques, reduced split trees, lies at the heart of the Fribourg construction that we describe in Chapter 3.

In the following subsections, we will give examples for the different run analysis techniques that are based on the non-deterministic Büchi automaton in Figure 2.1. Note that the alphabet of this automaton is $\Sigma = \{a\}$, and thus the only ω-word in Σ^ω is a^ω. However, the automaton has mutliple (infinitely many) runs for this word.

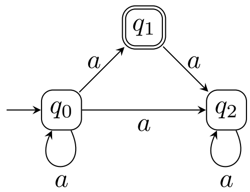


Figure 2.1: Non-deterministic Büchi automaton A used as example autoamton in this section.

2.2.1 Run DAGs

Run DAGs arrange all the runs of an automaton A on a word in a directed acyclic graph (DAG). This graph can be thought of as a matrix with rows and columns. The rows are called levels, and each column corresponds to a state of A. Consequently, the width (that is, number of columns) of a run DAG equals the number of states of A. Each level i (starting from 0) corresponds to the situation after reading the first i symbols of the word. Figure 2.2 shows the first five levels of the run DAG of the example automaton A in Figure 2.1 on the word a^ω.

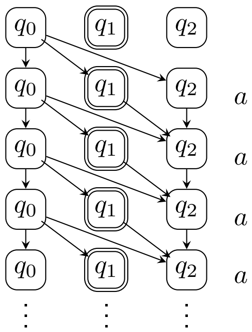


Figure 2.2: First few levels of the run DAG for the runs of automaton A (Figure 2.1) on the word a^ω.

Note that throughout this thesis, we are using rectangles with rounded corners for states of automata, and rectangles with sharp corners for vertices of graphs nodes of trees. In graphs and trees, we will however indicate vertices or nodes that correspond to accepting automaton states with double-lined rectangles.

As can be seen in Figure 2.2, every path of a run DAG corresponds to a run of the automaton on the given word. A run DAG is a structure that is able to represent an infinte number of runs by keeping a finite width. The number of levels of a run DAG is infinite for ω-words. A formal description of run DAGs can be found, for example, in [11].

Run DAGs are the basic structure for the rank-based complementation constructions, that we review in Section 2.3.3. Regarding this application, the fact that run DAGs have finite width is important, because the levels of run DAGs are mapped to states of the output automaton in these complementation constructions.

2.2.2 Run Trees

Trees in general are, after DAGs, the second structure that is used for run analysis. Run trees are the most basic of these tree structures. A run tree is basically a direct unwinding of all the runs of an automaton on a word as a tree. Figure 2.3 shows the first five levels of the run tree for the automaton A in Figure 2.1 on the word a^ω.

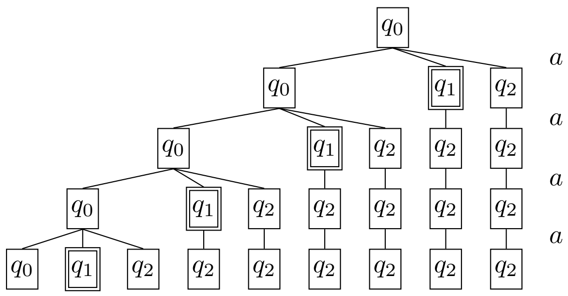


Figure 2.3: First five levels of the run tree for the runs of automaton A (Figure 2.1) on the word a^ω.

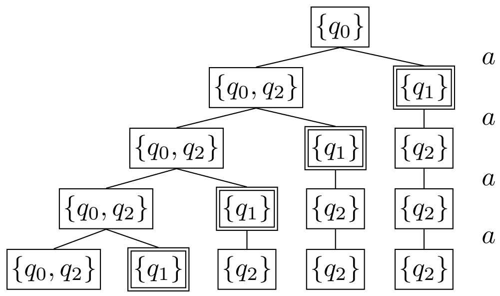


Figure 2.4: First five levels of the split tree for the runs of automaton A (Figure 2.1) on the word a^ω.

As can be see in Figure 2.3, there is a one-to-one mapping of paths in a run tree (from the root to the leaves) to runs of the corresponding automaton. This means that if the automaton has an infinite number of runs on a given word, then the corresponding run tree has an infinite maximum width. This makes run trees impractical to be used in Büchi complementation. The following tree techniques, split trees and reduced split trees, sacrifice a part of the information about individual runs, with the benefit of making the tree structure more compact.s

2.2.3 Split Trees

A split tree is basically a run tree where the accepting and non-accepting children of every node are merged. Consequently, a node of a split tree does not represent a single state of the automaton, but a set of states. The reduction of the number of children to at most two, makes the split tree furthermore a binary tree. Figure 2.4 shows the first five levels of the split tree of the automaton A (Figure 2.1) on the word a^ω.

Split trees sacrifice a part of the information about individual runs. For example, in Figure 2.4, we see that there must be a run that starts in q_0 (root node) and is in q_0 after reading four symbols (leftmost node on fifth level). However, the split tree does not provide the exact sequence of states. All that it reveals is that the sequence is $q_0 q_?^1 q_?^2 q_?^3 q_0$, where each $q_?^i$ is either q_0 or q_2. The run tree in Figure 2.3, on the other hand, shows the actual sequence unambiguously, which is $q_0 q_0 q_0 q_0 q_0$.

However, the split tree still provides an important piece of information, namely that the sequence $q_0 q_?^1 q_?^2 q_?^3 q_0$, whathever it actually looks like, does not contain any accepting states, because $q_?^i$ can only be q_0 or q_2, which are both non-accepting. It turns out that, with regard to the application of run analysis to Büchi complementation, this information is sufficient.

Split trees in fact embody a modified subset construction that does not mix accepting and non-accepting states. The result of applying such a construction to a non-deterministic automaton is an equivalent non-deterministic automaton whose degree of non-determinism is reduced to two (which means that each state has at most two successors for a given alphabet symbol). Such a construction has been described in [59].

A formal description of split trees can be found, for example, in [64] or [11]. Split trees are clearly more compact than run trees. However, their width may still grow infinitely. In the next section, we present a further reduction of the information contained in the tree, that bounds the maximum width of the tree to a finite number.

2.2.4 Reduced Split Trees

Reduced split trees are split trees that sacrifice even more information. However, the information that is retained is still sufficient for Büchi complementation. The rule is that on each level, going from right to left or from left to right, only the first occurrence of each state is kept, and the others are omitted. This bounds the maximum width of the tree to the number of states of the corresponding automaton.

The direction in which the first occurrence of a state is determined depends on whether right-to-left or left-to-right split trees are used. In *right-to-left split trees*, the accepting child is put to the right of the non-accepting child. The split tree in Figure 2.4 is thus a right-to-left split tree. In *left-to-right split trees*, on the other hand, the accepting child is put to the left of the non-accepting child. In the literature both versions are used. For example, Vardi and Wilke [64] describe the left-to-right version, whereas Fogarty et al. [11] describe the right-to-left version. In this thesis, we will use exclusively the right-to-left version.

In a right-to-left reduced split tree, the levels are processed from right to left. This means that only the rightmost occurrence of each state on a level is kept and the others are omitted. Figure 2.5 shows the first five levels of the reduced split tree of the automaton A (Figure 2.1) on the word a^ω.

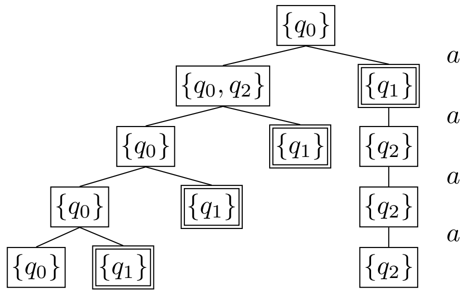


Figure 2.5: First five levels of the reduced split tree for the runs of automaton A (Figure 2.1) on the word a^ω.

As can be seen in Figure 2.5, each level contains at most one occurrence of each of the states q_0, q_1, and q_2. Comparing the reduced split tree in Figure 2.5 with the split tree in Figure 2.4 reveals which states have been omitted in the reduced split tree. The root level and level 1 are similar in both trees. On level 2, the state q_2 in the leftmost node is omitted. This is because there is already a q_2 farther to the right, namely in the rightmost node of level 2. In level 3, there are two omissions of q_2 for the same reason. One of them causes an entire node to disappear, because this node contained q_2 as its only state. Finally, on level 4, there are three omissions of q_2.

In the following, we illustrate what this omission of states entails and why the resulting "truncated" run analysis is still usabel for Büchi complementaton. By omitting states, we obviously omit runs from the tree. This omission of runs is targeted at so-called *re-joining runs*. Re-joining runs are runs that afer a certain number of steps end up in the same state again. For example, the automaton from Figure 2.1 has seven re-joining runs that after four steps end up in the state q_2. This can be easily seen in the corresponding run tree in Figure 2.3. A reduced split tree keeps exactly one of a group of re-joining runs and removes the others. This can be seen in Figure 2.5 which contains only one run from the root to q_2 in four steps.

The run that is kept is named *rightmost run*. The name refers to the fact that this run is the first of a group of re-joining runs that makes a *right turn* in the corresponding path of the tree. A right turn means the transition from a parent to an accepting child, which are situated to the right of the non-accepting child, hence the name[2]. The rightmost run of a group of re-joining runs has the following crucial property:

$$\text{Any re-joining run is accepting} \Longrightarrow \text{The rightmost re-joining run is accepting}$$

[2]Note that everything we explain here is also valid for left-to-right split trees, but one has to exchange *right*, *rightmost*, and *right turn* with *left*, *leftmost*, and *left turn*, respectively.

That is, if a group of re-joining runs contains an accepting run, then the rightmost run is accepting too. On the other hand, if the rightmost run is non-accepting, then all the other re-joining runs are non-accepting as well. A proof of this relation can be found, for example, in [64][3].

Practically, this means that we can reduce the existential question about the presence of acceptance in a group of runs to the rightmost run. Remember that for Büchi complementation we are interested in exactly such an existential question: is there an accepting run of the automaton A on the word α? If yes (it does not matter how many accepting runs there are), then A accepts α and the complement must not accept it. If no, then A does not accept α and the complement must accept it. By considering only the rightmost runs, we can still reliably answer this question.

Thus, reduced split trees include a tremendous reduction of the number of runs to be analysed, but still retain all the needed information for Büchi complementation. Most importantly, this reduction of runs bounds the width of the trees to a finite number, what makes them usable in Büchi complementation constructions, such as the Fribourg construction (see Chapter 3), and other slice-based constructions (see Section 2.3.4).

Note that an alternative name for rightmost run is *greedy run* [1]. This name refers to the fact that the rightmost run is the first of a group of re-joining runs that visits an accepting state, what makes it "greedy". It is interesting to note that reduced split trees are related to Muller-Schupp trees that are used in the Büchi determinisation construction by Muller and Schupp [34] (cf. [64, 11]). As a last remark, formal descriptions of reduced split trees can be found, for example, in [64] and [65] (left-to-right version), or [11] (right-to-left version).

This concludes our presentation of run analysis techniques. In the next section we will review the most prominent Büchi complementation constructions that have been proposed over the past 50 years. Many of them are based on the run analysis techniques that we have described in the present section.

2.3 Review of Büchi Complementation Constructions

Since the introduction of Büchi automata in 1962, many different Büchi complementation constructions have been proposed. In this section, we will briefly review some of the most important of them. Many of the constructions that we describe in this section are implemented in the GOALtool, that we use for the empirical performance investigation of the Fribourg construction (see Chapter 4). Three of them are actively used in our investigation as a comparison against the Fribourg construction, namely the constructions by Piterman [36, 37], Schewe [44], and Vardi and Wilke [64].

We organise our review of Büchi complementation constructions into the four approaches that have been proposed by Tsai et al. [53]: *Ramsey-based*, *determinisation-based*, *rank-based*, and *slice-based*. As a compact reference, we summarise below all the constructions that are included in the following review under their respective approach, and with their authors, year of introduction, and references.

Ramsey-based approach		
Büchi	1962	[8]
Sistla, Vardi, and Wolper	1987	[45, 46]
Determinisation-based approach		
Safra	1988	[42, 43]
Muller and Schupp	1995	[34]
Piterman	2007	[36, 37]
Rank-based approach		
Klarlund	1991	[22]
Kupferman and Vardi	1997/2001	[24, 25]
Thomas	1999	[52]
Friedgut, Kupferman, and Vardi	2006	[12, 13]
Schewe	2009	[44]

[3] Lemma 2.6

Slice-based approach		
Vardi and Wilke	2007	[64]
Kähler and Wilke	2008	[20]

2.3.1 Ramsey-Based Approach

The Ramsey-based complementation approach is the oldest approach and includes the earliest Büchi complementation constructions, including the construction described by Büchi himself [8]. Ramsey-based constructions are based on a branch of combinatorics called Ramsey theory [15], hence the name Ramsey-based approach[4]. They proceed by directly constructing the complement automaton out of the input automaton by applying combinatorial arguments.

Büchi (1962)

The first Büchi complementation construction in history, described by Büchi in his 1962 paper [8], is based on a combinatorial argument introduced by Ramsey in 1930 [40]. The construction is rather complicated [63], and has a doubly-exponential worst-case state growth of $2^{2^{O(n)}}$ [62]. This blow-up is enormous, for example, given an automaton with 9 states, the maximum size of the complement is 1.3×10^{154} states, tremendously more as there are atoms in the observable universe[5].

However, as mentioned in our discussion about the use of worst-case state growth as a performane measure for complementation constructions in Section 1.2, this number applies only to the worst case, which is a very extraordinary special case. Furthermore, we believe that Büchi's aim was not to create an *efficient* construction, but rather to prove that such a construction exists in the first place. Because for Büchi this served as a proof that Büchi automata are closed under complementation, which was in turn required to prove Büchi's Theorem [64].

Sistla, Vardi, and Wolper (1987)

In 1987, Sistla, Vardi, and Wolper proposed an improvement of Büchi's construction [46][6]. This construction was the first to include only an exponential, rather than doubly-exponential, worst-case state growth. The state growth function of the construction is $O(16^{n^2})$ [46]. Thus, given an automaton with 9 states, the maximum size of the complement is 3.4×10^{97} states.

2.3.2 Determinisation-Based Approach

The determinisation-based approach achieves complementation by chaining several conversions between different types of ω-automata. The most important of these conversions is the transformation of the non-deterministic Büchi automaton to a deterministic ω-automaton of a different type. Thus, the first step is to determinise the input automaton, hence the name determinisation-based approach.

Note that in this context *determinisation* refers to the conversion of a Büchi automaton to a *different type* of ω-automata, such as Muller, Rabin, Streett, or partity (see 2.1.2). As mentioned in Section 2.1.1, the determinisation of a Büchi automaton to a deterministic Büchi automaton is in general not possible, because determnistic Büchi automata are less expressive than non-deterministic Büchi automata.

This approach harnesses the fact that the complementation of deterministic automata is generally easier than the complementation of nono-deterministic automata. This means that once the non-deterministic Büchi automaton is turned into a deterministic ω-automaton, it can be easily complemented, and the result can then be converted back to a non-deterministic Büchi automaton.

[4] Ramsey was a British mathematician who lived at the beginning of the 20th century

[5] Assuming a number of 10^{80} atoms in the observable universe, according to http://www.wolframalpha.com/input/?i=number+of+atoms+in+the+universe.

[6] This paper was preliminarily published in 1985 [45].

Safra (1988)

The first determinisation-based complementation construction has been described in 1988 by Safra [42, 43]. Safra's main work is actually a construction for converting a non-deterministic Büchi automaton (NBW) to a deterministic Rabin automaton (DRW)[7]. This determinisation construction is what today commonly is known as *Safra's construction*. However, Safra also describes a series further conversions that, when chained together, result in a complementation construction. The complete series of conversions is as follows:

1. NBW $\longrightarrow$ DRW (Safra's construction)
2. DRW $\longrightarrow$ $\overline{\text{DSW}}$ (Complementation)
3. $\overline{\text{DSW}}$ $\longrightarrow$ $\overline{\text{DRW}}$
4. $\overline{\text{DRW}}$ $\longrightarrow$ $\overline{\text{NBW}}$

That means, the DRW resulting from Safra's determinisation construction is complemented to a DSW, which is converted to a DRW, which in turn is converted to an NBW. The resulting NBW is the complement of the input NBW. Note that a horizontal indicates that the automaton accepts the complement language of the automata without a bar.

Safra's construction is in fact a modified subset construction that guarantees the equivalence of the input and output automata[8]. The main difference of Safra's construction to the classical subset construction is that the states of the output automaton are labelled by trees of subsets of states, rather than just subsets of states. These trees are called *Safra trees*. Detailed analyses and explanations of Safra's construction can be found, next to Safra's papers [42, 43], in [41], [23], [2], or [27].

The complementation step from the DRW to the complement DSW can be trivially done by interpreting the Rabin acceptance condition as a Streett acceptance condition [23]. This works, because, as mentioned in Section 2.1.2, the two acceptance conditions are the duals of each other. The final conversions from DSW to DRW, and DRW to NBW are described by Safra in [42, 43] or alternatively in [23].

The entire construction from an NBW to a complement NBW has a worst-case state complexity of $2^{O(n \log n)}$. With this result, Safra's complementation construction matched the lower bound of $n!$ that has been introduced in the same year by Michel [29]. However, as pointed out by Vardi [62], the big-O notation hides a large gap between the two functions, and Safra's complementation construction is thus not as "optimal" as it seems. Thus, the quest for finding Büchi complementation constructions with an even lower worst-case state complexity continued.

Muller and Schupp (1995)

In 1995, Muller and Schupp proposed and improvement of Safra's determinisation construction [34]. This construction can be used at the place of Safra's original construction in the conversation chain that we described above for Safra's construction. Muller and Schupp's construction uses *Muller-Schupp trees* instead of Safra trees. While the principles of the two constructions are very similar, it is said that Muller and Schupp's construction is simpler and more intuitive than Safra's construction [41]. However, the drawback is that in many concrete cases Muller and Schupp's construction produces larger output automata than Safra's construction [2]. The worst-case state complexity of Muller and Schupp's construction is, similarly to Safra's construction, $2^{O(n \log n)}$. A detailed comparison of the determinisation constructions by Muller and Schupp, and Safra can be found in [2].

Piterman (2007)

A further improvement of Safra's determinisation construction has been proposed by Piterman from EPF Lausanne in 2007 [37][9]. The main difference is that Piterman's construction converts the input Büchi

[7] This is the notation for ω-automata types that we introduced in Section 2.1.2.

[8] Safra shows in his papers [42, 43] nicely that applying the classical subset construction to an NBW A results in a DBW B that may accept some words that are not accepted by A. This means that the subset construction is not *sound* with respect to Büchi automata.

[9] A preliminary version of this paper has appeared in 2006 [36].

automaton to a deterministic parity automaton (DPW), rather than a deterministic Rabin automaton (DRW). Furthermore, Piterman uses a more compact version of Safra trees, what results by trend in smaller output automata. Also in terms of worst-case complexity, Piterman's construction is more efficient than Safra's construction. The blow-up in Piterman's construction from the NBW to the DPW is $2n^n n!$, whereas in Safra's construction the blow-up from the NBW to the DRW is $12^n n^{2n}$ [36, 37].

Since Piterman's determinisation construction produces a DPW, it entails different conversions for complementation than Safra's construction. In particular, the procedure is as follows [53]:

1. NBW $\longrightarrow$ DPW (Piterman's construction)
2. DPW $\longrightarrow$ $\overline{\text{DPW}}$ (Complementation)
3. $\overline{\text{DPW}}$ $\longrightarrow$ $\overline{\text{NBW}}$

The complementation of the DPW can be trivially done by increasing the number assigned to each state by one [53] (see Section 2.1.2 for an explanation of the parity acceptance condition). The complemented DPW can then be converted directly to an NBW [53].

2.3.3 Rank-Based Approach

The rank-based approach is based on run analysis with run DAGs (see Section 2.2.1). The basic "mechanics" of rank-based construction is similar to the subset construction, that is, the output automaton is constructed state by state, by adding to every state a successor state for every symbol of the alphabet. In rank-based constructions, these states are derived from levels of a run DAG. The idea is that the run analysis with run DAGs is interweaved with the construction, so that the states of the output automaton represent levels of "virtual" run DAGs.

As we have seen in Section 2.2, the purpose of run analysis is to find out whether *all* runs of an input automaton A on a word α are non-accepting. Because in this case, the complement B must accept α, and in all other cases it must not accept it. In rank-based constructions this is achieved by the means of natural numbers called *ranks* that are assigned to the vertices of a run DAG (hence the name rank-based approach). These ranks are assigned in such a way that vertices corresponding to *accepting* states only get *even* ranks. Furthermore the ranks along paths of the run DAG are monotonically decreasing. The effect of this is that every path gets eventually trapped in a rank. If for a given path this trapped rank is odd, it means that the corresponding run does not visit any accepting states anymore starting from a certain point, and thus is non-accepting. If all the paths of a run DAG get trapped in an odd rank, then all the runs of the automaton on the given word are non-accepting. This situation is called *odd-ranking* and indicates that the complement must accept the word.

This basic idea is the same for all rank-based construction and described in many places, for example, [11, 25, 13, 62, 21, 44]. The individual rank-based constructions differ mainly in details of how the ranking is done.

Klarlund (1991)

Klarlund's construction [22] rather foreshadowed the rank-based constructions than being one itself. It was one of the first constructions that was neither Ramsey-based nor determinisation-based [22, 25]. Its fundamental idea is very similar to the later rank-based constructions, but it does not use the term "rank" and is not explicitely based on run DAGs. However, Klarlund uses so-called *progress measures*, which have a very similar role as the ranks of later constructions. This construction served as the base for Kupferman and Vardi's rank-based construction in 1997 [24]. The worst-case state growth of Klarlund's construction is $2^{O(n \log n)}$

Kupferman and Vardi (1997/2001)

This construction has been published for the first time in 1997 [24] and again in 2001 [25]. Both publications are entitled "Weak Alternating Automata Are Not That Weak". Kupferman and Vardi basically pick up Klarlund's construction and describe it in a different way [25]. Actually, they provide two different descriptions for the same construction, one based on weak alternating automata (WAA)[10], and one rank-based.

The first description includes several conversions between ω-automata types. It starts by interpreting the input NBW as a universal co-Büchi automaton (UCW) which accepts the complement language of the initial NBW. This UCW is converted to a WAA, which is finally converted to an NBW. The second description equals the rank-based approach that we described in the introduction to this section. Kupferman and Vardi state that their two versions and Klarlund's construction produce identical results. However, they claim that their descriptions make the construction easier to understand and implement. [25].

Thomas (1999)

This construction by Thomas [52] is based on the concept of alternating automata, like the previous construction by Kupferman and Vardi (1997) that proceeds via a WAA [24]. The difference of Thomas' construction is that it performs the complementation step in the framework of alternating automata. In particular, the input NBW is converted to a weak alternating parity automaton (WAPA), which is complemented to another WAPA, which finally is converted back to an NBW. Note that Thomas uses the concept of ranks that are assigned to the states of an alternating automataon in a similar way as we described in the introduction to this section. Thus, we categorise this construction under the rank-based approach, even if it does not proceed in the typical way.

Friedgut, Kupferman, and Vardi (2006)

In 2006, Friedgut, Kupferman, and Vardi [13][11] proposed an improvement of the rank-based construction by Kupferman and Vardi from 2001 [25]. The improvement consists in a better ranking function called *tight ranking* that reduces the maximum number of states in the output automaton[12] The worst-case state complexity of the construction is $(0.96n)^n$, which is an enormous reduction from the 2001 construction by Kupferman and Vardi with a result of $(6n)^n$. The title of Friedgut, Kupferman, and Vardi's paper is "Büchi Complementation Made Tighter". It refers to the fact that this construction provides a new upper bound for the precise state complexity of Büchi complementation that considerably *tightens* the gap between the best known upper bound and the then known lower bound by Michel [29] of $n!$.

Schewe (2009)

The work by Schewe [44] is a further optimisation to the construction by Kupferman, Friedgut and Vardi [13]. It consists of the use of *turn-wise* tests in the *cut point construction* [44]. This optimisation allows to reduce the worst-case state complexity of the construction to $(0.76n)^n$ modulo a polynomial factor in $O(n^2)$ [44], that is, around $(0.76n)^n O(n^n)$. This comes very near to the lower bound of $(0.76n)^n$ that has previously been established by Yan [66, 67]. Schewe's paper is entitled "Büchi Complementation Made Tight", which refers to the fact that finally the gap between the lower and upper bounds for Büchi complementation has been made really "tight" (after having been made only "tighter" by Friedgut, Kupferman, and Vardi in 2006 [13]).

2.3.4 Slice-Based Approach

The slice-based approach can be seen as the brother of the rank-based approach. The main diffrence is that the slice-based approach is based on *reduced split trees* (see Section **??**), rather than run DAGs. As for the rank-based approach, the "mechanics" of the slice-based approach is similar to the subset construction, that is, the output automaton is constructed state by state. In a slice-based construction these states are derived from levels of reduced split trees (as opposed to rank-based constructions where

[10] Weak alternating automata have been introduced by Muller, Saoudi, and Schupp in 1986 [33].

[11] This paper has been preliminarily appeared in 2004 [12].

[12] Alternative descriptions of the tight ranking function can be found, for example, in [11], [44], or [62].

the states are derived from levels of run DAGs). The levels of reduced split trees are called *slices* [64], hence the name slice-based approach.

Slice-based constructions use decorations, also called colours [1], that they assign to the vertices of the corresponding reduced split trees (and thus to the components of the states that are derived from the slices of the tree). These decorations serve the same role as the ranks in rank-based constructions, namely the detection whether all runs of the input automaton on a specific word are non-accepting, in which case the complement must accept the word. A comparative analysis of the slice-based and the rank-based approach has been done by Fogarty et al. [11][13]. Below we briefly present the two principal slice-based constructions.

Vardi and Wilke (2007)

With this construction, Vardi and Wilke have established the slice-based approach [53]. The construction is divided into an *initial phase* and a *repetition phase*. In the initial phase, slices (thus, states) are constructed without decorations. At some point, the construction guesses to transfer to the repetition phase in which the vertices of the slices (thus, components of states) are decorated by one of the decorations *inf*, *new*, or *die*. These decorations determine whether the enclosing state is accepting or non-accepting. A loose upper bound for the worst-case state complexity of this construction is $(3n)^n$ [64]. Vardi and Wilke describe their construction using left-to-right reduced split trees.

Kähler and Wilke (2008)

This construction by Kähler and Wilke [20] is a generalisation of the previous slice-based construction by Vardi and Wilke [64]. The construction is modified such that it can treat complementation and disambiguation[14] of Büchi automata in a uniform way [53]. The generalisation however comes at the price of a higher worst-case state complexity of $4(3n)^n$ [53]. Kähler and Wilke also use left-to-right trees for their construction.

With this review of the four main Büchi complementation approaches, we conclude the present chapter. In the next chapter we will explain in detail the Fribourg construction, which is another slice-based construction, and which is the subject of our empirical performance investigation that we present later in the thesis.

[13] In this work, the authors even propose a construction that unifies the two approaches.

[14] Disambiguation means the conversion of an automaton to an equivalent *unambiguous* automaton. An automaton is unambiguous if every accepted word has only one accepting run [30]. Note that this differs from determinisation, as a word may still have multiple runs, but only one accepting run.

Chapter 3

The Fribourg Construction

Contents

In this chapter we present the Fribourg construction. The Fribourg constructio is the Büchi complementation consruction whose empirical investigation of performance makes up the core of this thesis (Chapters 4 and 5). The Fribourg construction belongs to the *slice-based* complementation approach that we reviewed in Section 2.3.4. This means that it is based on reduced split trees, which we explained in Section 2.2.4. Of the two versions of reduced split trees, left-to-right and right-to-left, our description of the Fribourg construction in this chapter uses right-to-left reduced split trees.

The Fribourg construction is being developed at the University of Fribourg by Joel Allred and Ulrich Ultes-Nitsche[1]. A detailed description of the construction has been published internally in 2014 as a technical report entitled "Complementing Büchi Automata with a Subset-tuple Construction" [1].

The aim of this chapter is to give an intuitive and practically oriented description of the construction. We try to make its concrete application as clear as possible. Our focus lies therefore not on a formal description of the construction. For the formal background, as well as proofs of correctness, we refer to the work by the authors of the construction [1].

This chapter is structured as follows. In Section **??**, we describe the basic Fribourg construction. The construction is divided into two sub-constructions, and we explain each of them separately. In Section 3.2, we present three optimisations that have been proposed for the Fribourg construction. These optimisations will also be subject to our empirical performance investigation in the next chapter. Finally, in Section 3.3, we give some general remarks about the Fribourg construction, especially putting it into relation with other complementation constructions.

3.1 The Construction

In this section we explain the basic Fribourg construction without any applied optimisations. The construction consists of two sub-constructions that we call upper-part construction and lower-part construction. In the following we first describe features that are common to these two sub-construction (Section 3.1.1). Then, in Section 3.1.2 and 3.1.3, we describe in detail the upper-part construction and the lower-part construction, respectively. We illustrate the application of these two sub-construction with an example which we work through step by step.

3.1.1 Basics

Below we first give a high-level view of the construction, and then present the state generation procedure of the construction. The state generation procedure determines a successor state for an existing state, and lies at the heart of the construction. In its basic form, the state generation procedure is common to the two sub-constructions that we present in the next two sections.

High-Level View

As mentioned, the Fribourg construction consists of two sub-constructions, the *upper-part construction* and the *lower-part construction*. These two sub-constructions are chained together, such that the output of the upper-part construction becomes the input of the lower-part construction. Figure 3.1 illustrates this in a functional view of the Fribourg construction.

The automaton A to be complemented is the input for the upper-part construction. The upper-part construction builds up a new automaton by starting with an initial state and adding state by state. The output of the upper part constructio is an intermediate automaton U. This intermediate automaton U will be the *upper part* of the final complement automaton. The lower-part construction takes U as input and adds further states and transitions to it until finally outputting an automaton B which is the complement of the input automaton A. We say that the lower-part construction attaches the *lower part* of the final complement automaton to the upper part. This terminology comes from the fact that it is

[1] As mentioned, the authors call their construction*subset-tuple construction*, in this thesis we will however use the name *Fribourg construction*.

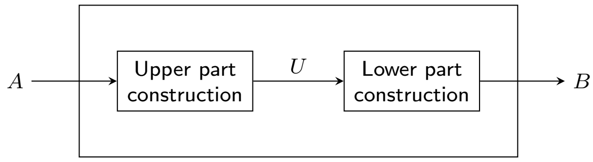


Figure 3.1: Functional view of the Fribourg construction with an automaton A as input and its complement B as output.

convenient to draw the states of the lower part below the states of the upper part [1]. This also explains why the two sub-constructions are calleed upper-part construction and lower-part construction.

State Generation

The two sub-constructions can be seen as modified subset constructions. Like the subset construction, they work on a set of existing states and add to each state one sucessor for each symbol of the alphabet. The principle of how successor states are generated is the same for both sub-construction. The difference between the two is that the lower-part construction adds additional information to the states (the colours), so that the states generated by the lower-part construction are different from the states generated by the upper-part construction. Below we explain this basic principle of how new states are generated.

A state of the Fribourg consists of a tuple (that is, an ordered sequence) of subsets of states of the input automaton (note that we will from now on refer to the states of the input autmaton as the *input-states*, and to the states of the output automaton as *output-states*). This contrasts with the subset construction in which the output-states consist of a single subset of input-states. The states of the Fribourg construction are thus more structured than the states of the subset construction. We refer to the subsets of a tuple of a state as the *components* of this state.

The sequence of components of each state of the Fribourg construction is imposed by a slice (that is, a level) of a reduced split tree. We will explain the process of going from an existing state over the slices of a reduced split tree to a successor state further below. For now, we look at how a slice defines the structure of a state. Figure 3.2 shows a reduced split tree (on the left), and for each slice (framed by a shaded box), the state that this slice would define (on the right).

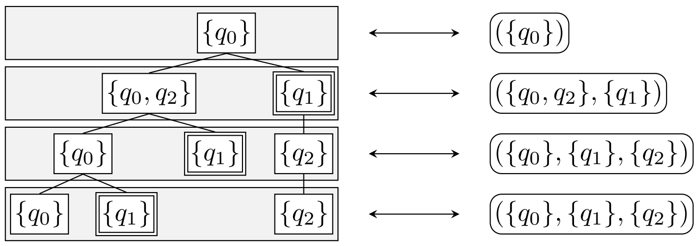


Figure 3.2: Relation between slices of a reduced split tree (left) and states of the output automaton of the Fribourg construction (right). The sequence of nodes in a slice determines the sequence of components in a state.

As can be seen in Figure 3.2, each node of a slice gives rise to a component in the corresponding state. To this end, it does not matter whether the nodes in the slices are siblings or whather their familial relations are. It is important to note, however, that the order of the nodes in a slice must be preserved in the order of the components in a state. It is similarly possible to interpret a state as a slice of a reduced split tree (indicated by the double-ended arrows in Figure 3.2). In this case, it is not possible to deduce the ancestors of the resulting nodes, however, this is not necessary for the construction.

The procedure of determing the successor of a given existing state on a certain alphabet symbol is as follows:

- Interpret the existing state as a slice of a reduced split tree
- Create the subsequent slice for the appropriate alphabet symbol

- Interpret the new slice as a state

The state that results from thew new slice is the successor of the existing state for the given alphabet symbol. If, for example, we want to define the successor on the symbol a (a-successor) of the state $(\{q_0\})$, then as the first step we would create a slice representation of it that looks like the first slice from the top in Figure 3.2. Then, we create based on the input automaton the subsequent slice for symbol a, that might look like the second slice from the top in Figure 3.2. Interpreting this slice as a state results in $(\{q_0\}2, \{q_1\})$, which thus is the a-successor of $\{q_0\}$.

Note that the translation of states to and from slices is only a mental aid and not a necessary ingredient. It is possible to deduce $(\{q_0\}2, \{q_1\})$ from $\{q_0\}$, in our previous example, directly without the detour over slices of a reduced split tree. However, by conceptually using reduced split trees, we can cover the central points of the state generation process by re-using our knowledge of the previously known and independent concept of reduced split trees. This includes the splitting of accepting and non-accepting input-states, the placement of the accepting subset to the right of the non-accepting subset, and the retention of only the first occurrence of an input-state from right to left.

3.1.2 Upper-Part Construction

In this subsection, we recapitulate the upper-part construction, and demontrate its application with an example. The example is based on the input automaton in Figure 3.3.

Description

The upper-part construction consists basdically only of the state generation procedure that we explained the last section. It starts with an inital state with a single component containing the inital state of the input automaton. This will be the initial state of the final complement automaton. Then, the state generation procedure is applied for each alphabet symbol for each unprocessed state.

If a determined successor state is identical to a state the already exists in the automaton, then only a transition to this state is added. If a determined successor state is empty, that is, contains no components, then no new state or transiton at all is added to the automaton. In this case, the state being processed has no successor for the corresponding alphabet symbol, and we say that it is incomplete.

The state generation process is repeated until all states have been processed. At this stage, a deterministic intermediate automaton without any accepting states has been produced. This intermediate automaton will be the upper part of the final complement automaton.

A peculiarity is that the upper part must be complete. An automaton is complete, if every state is complete, which means having an outgoing transition for each symbol of the alphabet. If the upper-part automaton is not complete, then it must be made complete by adding a sink state. A sink state is a state that serves as the “missing successor” of all the incomplete states. Note that this sink state that is potentially added to the upper-part automaton must be accepting.

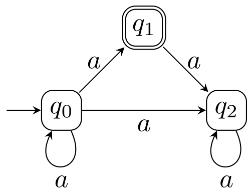


Figure 3.3: Non-deterministic Büchi automaton used for the examples in this section.

Example

We demonstrate the application of the upper part construction by applying it to the example automaton A in Figure 3.3. Note that this automaton has an alphabet size of one, and thus can run only on a single

ω-word, namely a^ω. Furthermore, the automaton does not accept this word, which means that it does not accept *any* word and is thus empty. We selected this special automaton to keep our example handy and small. However, the construction works in exactly the same way for non-empty automata with larger alphabets.

The steps of the upper-part construction on the automaton A are showin in Figure 3.4 (a) to (d). Below, we walk through these steps one by one.

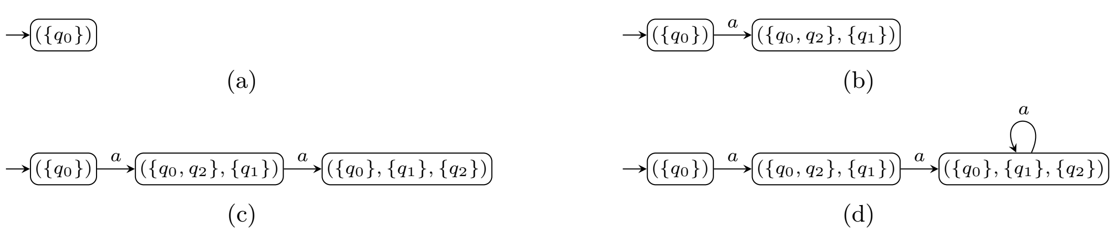
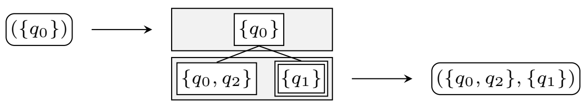


Figure 3.4: Steps of the upper-part construction applied to the example input automaton in Figure 3.3.

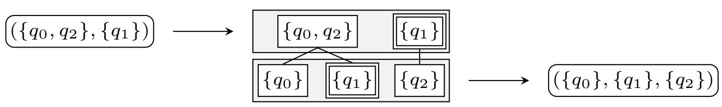
The first step in Figure 3.4 (a) is to start with the initial state. As can be seen, this state $(\{q_0\})$ contains only one component with the initial state of the input automaton A. This state will be the initial state of the final complement automaton B.

The next step Figure 3.4 (b) is to add the a-successor of the inital state $(\{q_0\})$. This is done by the state generation procedure that we described in the last section. In particular, this means: (1) interpret the state $(\{q_0\})$ as a slice of a reduced split tree, (2) based on the input automaton A, determine the subsequent slice for symbol a, and (3) interpret this new slice as a state. This new state is the a-successor of $(\{q_0\})$. We can illustrate these steps in the following way:



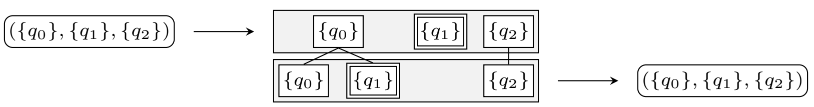
The state $(\{q_0\})$ results in a slice with only one node containing $\{q_0\}$. For creating the next slice, we determine the set of states of A that can be reached from $\{q_0\}$ on symbol a, which is $\{q_0, q_1, q_2\}$. According to the rules of reduced split trees, this set is split into a non-accepting $\{q_0, q_2\}$, and an accepting $\{q_1\}$. The non-accepting set $\{q_0, q_1, q_2\}$ becomes the left child, and the accepting set $\{q_1\}$ becomes the right child of the current node. Translating the resulting new slice to a state results in $(\{q_0, q_2\}, \{q_1\})$. This state is thus set as the a-successor of $(\{q_0\})$ in the automaton under construction.

The next step in Figure 3.4 (c) is to determine the a-successor the newly created state $(\{q_0, q_2\}, \{q_1\})$. Applying the same procedure as above, results in the following:



The state $(\{q_0, q_2\}, \{q_1\})$ corresponds to a slice with two nodes. For creating the next slice, these nodes must be processed from right to left. Regarding the node with $\{q_1\}$, the set of states in A that is reachable from $\{q_1\}$ on a is $\{q_2\}$, which becomes thus the only child of this node. Regarding the next node with $\{q_0, q_2\}$, the state set reachable from $\{q_0, q_2\}$ is $\{q_0, q_1, q_2\}$. Now, since q_2 already exists in the slice under construction, it is removed from the set. The resulting set $\{q_0, q_1\}$ is split into a non-accepting $\{q_0\}$ and an accepting $\{q_1\}$, which become the left child and right child of this node, respectively. The resulting slice corresponds to the state $(\{q_0\}, \{q_1\}, \{q_2\})$, which thus becomes the a-successor of $(\{q_0, q_2\}, \{q_1\})$.

The last step in Figure 3.4 (d) consists in determining the a-successor of $(\{q_0\}, \{q_1\}, \{q_2\})$. The state generation procedure in this case looks as follows:



The state $(\{q_0\}, \{q_1\}, \{q_2\})$ results in a slice with three nodes. Again, we have to process these nodes from right to left. The node with $\{q_2\}$ has a non-accepting $\{q_2\}$ as its only child. Regarding the node with $\{q_1\}$, the reachable set in A is $\{q_2\}$ However, since q_2 already exists in the slice, it is omitted, and since this empties the set, no child of this node is added to the new slice. Regarding the last node with $\{q_0\}$, the set of states reachable from $\{q_0\}$ is $\{q_0, q_1, q_2\}$, but again, due to the omission of q_2 and the splitting into an accepting and a non-accepting set, this results in the accepting right child $\{q_1\}$ and the non-accepting left child $\{q_0\}$. The state corresponding to this slice is $(\{q_0\}, \{q_1\}, \{q_2\})$, which is identical to the state we are currently processing. Consequently, we only add a transition from $(\{q_0\}, \{q_1\}, \{q_2\})$ back to itself.

At this stage, all the states in the intermediate automaton have been processed. Since the resulting automaton is complete, there is no need to add a sink state to it. This means that the upper-part construction is finished.

Note that in this example, we had to determine only one successor for each state, because there is only one symbol in the alphabet of our example automaton. For input automata with more than one alphabet symbols, the procedure that we did for each state must be repeated for each symbol of the alphabet. This increases the number of required steps, however, the basic principles generating states stay exactly the same.

3.1.3 Lower-Part Construction

Below, we describe in detail the lower-part construction. The lower-part construction can be seen as an extended version of the upper-part construction. The difference is that the lower-part construction additionally assigns colours to all the components. These colour assignments are the main subject of our explanations below. After explaining the lowe-part construction, we demonstrate its application by an example. To this end, we will use and continue the same example that we used for the upper-part construction.

Description

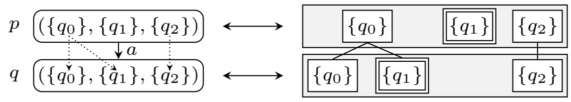
The lower-part construction takes the intermediate automaton resulting from the upper-part construction as input. It works on the states of the upper part by regarding them as the unprocessed states to be processed. This results in additional successors for the states of the upper part, and finally in the entire lower part growing out of the upper part. Note that in the final complement automaton, there will be transitons from the upper part to the lower part, but not back from the lower part to the upper part. This means that once a run of the output automaton enters the lower part, it cannot go back to the upper part again.

The basic state generation procedure of the lower-part construction is similar as in the upper-par construction. The difference between the two sub-constructions is that the lower-part construction adds an additional piece of information to each component of new states. This piece information consists of one of three colours, that we denote by 0, 1, and 2. We shed light on the meaning of these colours further below. For now, we explain how the colour of a component is determined.

The colour of component c of a new state q that is the successor (for a given alphabet symbol) of an existing state p, is determined by three pieces of information:

1. Whether the component c is accepting or non-accepting
2. The colour of the predecessor component of c
3. Whether the predecessor state p contains any 2-coloured components

Regarding the first point, a component is accepting if it contains accepting input-states. Regarding the second point, each component c in a state q has exactly one predecessor component c_{pred} in the predecessor state p. This relation emerges from the parent-child relation in the reduced split tree representation of the two states. That is, from this perspective c_{pred} is the parent of c. Consequently, every component has exactly one predecessor component, but every component may have zero, one, or two successor components. This relation of components can be best illustrated by an example. Consider the following two state p and q and their corresponding slice representation:



The edges of the slice representation of the states (on the right) can be mentally merged into the states (indicated as dotted lines). This makes the relation between the components of the two states apparent. Regarding the componnets of q, we can see that the predecessor component of $\{q_0\}$ is $\{q_0\}$, of $\{q_1\}$ is $\{q_0\}$, and of $\{q_2\}$ is $\{q_2\}$. On the other hand, regarding the components of p, we can see that $\{q_0\}$ has two successor components ($\{q_0\}$ and $\{q_1\}$), $\{q_1\}$ has no successor components, and $\{q_2\}$ has one successor component ($\{q_2\}$). For the case of $\{q_1\}$ which has no successor components, we say that the input-runs leading through this components "disappear".

Having explained the terms accepting component and predecessor component, we can now turn to the actual rules that determine the colour of a given component. One remark is however still necessary. The colouring rules need a way to recognise components that belong to the upper part of the output automaton. A way to guarantee this to preliminarily assign the colour -1 to all the components of the upper-part states. In the following, we will rely on this convention. Figure 3.5 shows the complete rules for determining the colour of a component c.

Colour of c_{pred}	c is non-accepting	c is accepting
-1	**0**	**2**
0	**0**	**2**
1	**2**	**2**

(a) Case A: the predecessor state has *no* 2-coloured components.

Colour of c_{pred}	c is non-accepting	c is accepting
0	**0**	**1**
1	**1**	**1**
2	**2**	**2**

(b) Case B: the predecessor state *has* 2-coloured components.

Figure 3.5: Rules for determining the colour of a component c (in bold). The identifier c_{pred} denotes the predecessor component of c.

As can be seen in Figure 3.5 the colour rules are divided into two cases. Case A applies if the predecessor state does not contain any components with the colour 2, and case B applies if the predecessor state contains one or more components with the colour 2. In each of the cases, the colour of a component c is unambiguously determined by the two remaining factors, colour of the predecessor component c_{pred}, and by the fact whether c is an accepting or not. We provide the rationale behind these rules below, when we explain the meaning of the colours.

Note that in case A, the possible colours of the predecessor component includes -1, but not 2. The missing of 2 is simply because by definitoin case A applies only if the predecessor state has no 2-coloured components, thus the colour of the predecessor component cannot be 2. However, in this case, the colour of the predecessor component may be -1, namely in the case that the predecessor state is an upper-part state. In case B, on the other hand, colour -1 is missing. This is because case B applies only if the predecessor state contain 2-coloured components, in which case it cannot be an upper-part state.

Having explained the rules for determining the colours of components, we now give some insight into the meaning of these colours. The purpose of the colours is to signalise the presence or absence of certain runs of the input automaton (input-runs) entailed by a corresponding run of the output automaton (output-run). We divide these input-runs into *dangerous* and *safe* runs. Dangerous input-runs have visited at least one accepting input-state since entering the lower part. These runs are dangerous in the sense that they might become accepting if they keep visiting accepting input-states. Safe runs, on the other hand, are runs that have not yet visited an accepting input-state since enetering the lower part. Input-runs that never become dangerous correspond to non-accepting runs in the input automaton.

The purpose of the distinction of safe and dangerous input-runs is to recognise whether *all* input-runs corresponding to an output-run on a specific word are safe. In this case, the output-run must accept the word. From a practical point of view, the colours of the components of a state determine whether the state must be added to the accepting set (according to a rule that we present at the end of this section). In particular, the meaning of the three colours 0, 1, and 2 is as follows:

Colour 2 Signalises the presence of dangerous input-runs, that is, runs that have visited at least one accepting input-state. Colour 2 appears when a group of input-runs, that have previously been safe, visits an accepting component (see Figure 3.5 (a) lines 1 and 2). However, once a component has colour 2, it passes on this colour to all its successor components, no matter if they are accepting or non-accepting (see Figure 3.5 (b) line 3). This means that a path of 2-coloured components can only be cut if at some point the input-runs going through this component disappear (that is, the components has no successors). In this case, we say for brevity that the 2-coloured components disappears.

Colour 1 Means basically the same as colour 2, namely the presence of dangerous runs. The idea behind colour 1 is a caveat that can arise if we would assign colour 2 to *every* component whose corresponding input-runs just visited for the first time an acceptign state. In this case, it could happen that in a sequence of output-states, there are constantly newly appearing 2-coloured components, and at the same time constantly disappearing 2-coloured components. The disappearance of the 2-coloured components means that the dangerous runs disappear, which in fact makes the corresponding output-states safe. However, this disappearance is hidden by the constant appearance of new 2-coloured components. The reason for colour 1 is to prevent this from happening. If the predecessor state already contains 2-coloured components (case B in Figure 3.5), then all the components that according to case A in Figure 3.5 would deserve colour 2, get colour 1 instead (see Figure 3.5 (b) line 1 and 2). We say that these are dangerous runs that are *on hold*. However, as soon as all 2-coloured components disappear, and consequently case A applies again, then all the successors of the 1-coloured component get colour 2 (see Figure 3.5 (a) line 3). In this way, the dangerous components that were previously on hold, become the new active dangerous components.

Colour 0 Signalises that all input-runs are safe. This means that these input-runs have not yet visited an accepting input-state since entering the lower part. If these runs stay safe indefinitely, then they are non-accepting in the input automaton.

As mentioned, the colours of the components of a state determines if a state must be made accepting. The corresponding rule is as follows:

Each state of the lower part that contains no 2-coloured components is an accepting state.

That is, states that contain only 0-coloured and 1-coloured components must be made accepting. The rationale behind this rule is that if an output-run visits such a state infinitely often, then all the input-runs on the corresponding word are non-accepting. For states with only 0-coloured components, this is clear, as colour 0 means that there are no dangerous runs. For states with 1-coloured components, the case is slightly more complicated, because 1-coloured components actually signalise the presence of dangerous runs. However, note that the situation that a state contains 1-coloured components, but no 2-coloured components (case A in Figure 3.5), can only occur if previously all 2-coloured components disappeared. If this happens an infinite number of times (which is implied by the output-run visiting this state infinitely often), then an ifninite number of 2-coloured components disappear. This means that all the corresponding dangerous input-runs die after reading a finite number of symbols, which means that none of them can be accepting. Consequently, a state with 1-coloured components also guarantees that all the corresponding input-runs are non-accepting.

All these rules and mechanisms become clearer by seeing them in action. Below we demonstrate the application of the lower-part construction by an example.

Example

We continue the complementation of the example automaton A in Figure **??** that we left off after the upper-part construction in the last section. In this section, we step by step add the lower part to the upper part, and in the end we obtain the complement automaton of A.

Figure 3.6 shows a selection of the steps of the lower-part construction applied to the upper part depicted in Figure 3.4 (d). Note that we use the following notation to indicate the colour of a component $\{q_i,\dots\}$:

- $\widehat{\{q_i,\dots\}}$: colour -1
- $\{q_i,\dots\}$: colour 0
- $\overline{\{q_i,\dots\}}$: colour 1
- $\overline{\overline{\{q_i,\dots\}}}$: colour 2

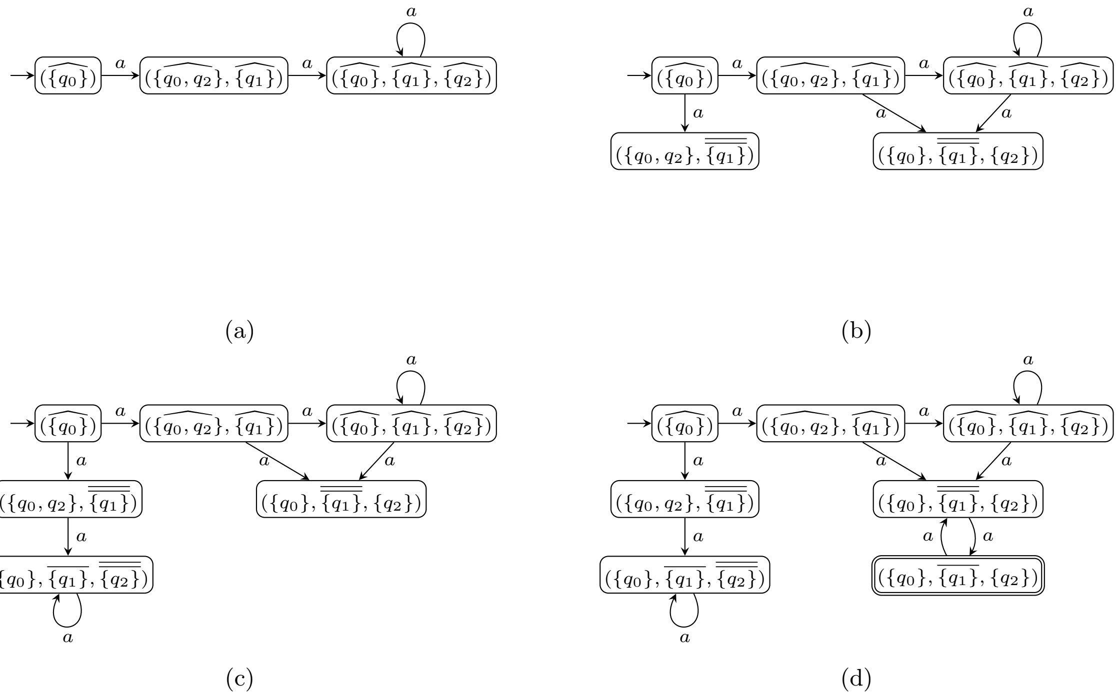


Figure 3.6: Selected steps of lower-part construction applied to the upper part depicted in Figure 3.4 (d).

The first step of the lower-part construction in Figure 3.6 (a) is to start with the upper part (as depicted in Figure 3.4 (d)), and assign colour -1 to all the components. As mentioned, we use this convention with colour -1 throughout this thesis.

The following steps of the lower-part construction consist in fact of the state generation function (as described in Section 3.1.1) plus the assignment of colours as described above. In Figure 3.6 (b) we add the a-successos for all the upper-part states. The state generation procedure results in the successor states $(\{q_0,q_2\},\{q_1\})$, $(\{q_0\},\{q_1\},\{q_2\})$, and $(\{q_0\},\{q_1\},\{q_2\})$. To determine the colours of the components of these states, we consult the colour rules in Figure 3.5. The predecessor states of all the new states do not contain a 2-coloured component, thus case A in Figure 3.5 (a) applies. Furthermore, since the predecessor states are upper-part states, all the predecessor components have colour -1. That means that for each component only the rule in line 1 of Figure 3.5 applies. According to this rule, the non-accepting components get colour 0 and the accepting components get colour 2. This results in the successor states $(\{q_0,q_2\},\overline{\overline{\{q_1\}}})$ for $(\widehat{\{q_0\}})$, $(\{q_0\},\overline{\overline{\{q_1\}}},\{q_2\})$ for $(\widehat{\{q_0,q_2\}},\widehat{\{q_1\}})$, and $(\{q_0\},\overline{\overline{\{q_1\}}},\{q_2\})$ for $(\widehat{\{q_0\}},\widehat{\{q_1\}},\widehat{\{q_2\}})$. Note that this makes the upper-part states non-deterministic. In particular, this is the way how the runs of the final output automaton can switch non-deterministically to the lower part.

In Figure 3.6 (c), we create the a-successor of $(\{q_0,q_2\},\overline{\overline{\{q_1\}}})$. Again, the new state is first determined by the state generation procedure, what results in $(\{q_0\}, \{q_1\}, \{q_2\})$. Now, we have to determine the colour for each of the components $\{q_0\}$, $\{q_1\}$, and $\{q_2\}$. In this case, the predecessor state $(\{q_0, q_2\}, \overline{\overline{\{q_1\}}})$ contains a 2-coloured component, thus case B of the colour rules in Figure 3.5 applies. Next, we need to know the colour of the predecessor of each component. To figure this out, we refer to the slice representation of the two states that we used for the state generation procedure. As explained earlier, we can mentally merge the edges between the nodes of the slices into the two states. This results in the following picture, where the component relations are indicated by dotted arrows:

$$\begin{array}{c} \boxed{(\{q_0, q_2\}, \overline{\overline{\{q_1\}}})} \\ \downarrow a \\ \boxed{(\{q_0\}, \{q_1\}, \{q_2\})} \end{array}$$

Consequently, the predecessor component of $\{q_0\}$ and $\{q_1\}$ is $\{q_0, q_2\}$, and the predecessor component of $\{q_2\}$ is $\overline{\overline{\{q_1\}}}$. Regarding $\{q_0\}$ and $\{q_1\}$, the colour of their predecessor component is 0, thus the rule on line 1 of Figure 3.5 (b) applies. Since $\{q_0\}$ is non-accepting, it gets colour 0, and since $\{q_1\}$ is accepting, it gets colour 1. Note how the assignment of colour 1 instead of colour 2 to $\{q_1\}$ sets it *on hold* because there exists already a 2-coloured component in the predecessor state. Regarding $\{q_2\}$, the colour of its predecessor component is 2, and thus, according to the rule on line 3 of Figure 3.5 (b) it gets colour 2. This results in the state $(\{q_0\}, \overline{\{q_1\}}, \overline{\overline{\{q_2\}}})$.

Still in Figure 3.6 (c) we create the a-successor of this new state $(\{q_0\}, \overline{\{q_1\}}, \overline{\overline{\{q_2\}}})$. By the state generation procedure, we obtain $(\{q_0\}, \{q_1\}, \{q_2\})$. Since the predecessor state $(\{q_0\}, \overline{\{q_1\}}, \overline{\overline{\{q_2\}}})$ has a 2-coloured component, case B of Figure 3.5 applies. Regarding the predecessors of the components in $(\{q_0\}, \overline{\{q_1\}}, \overline{\overline{\{q_2\}}})$ we observe the following relations:

$$\begin{array}{c} \boxed{(\{q_0\}, \overline{\{q_1\}}, \overline{\overline{\{q_2\}}})} \\ \downarrow a \\ \boxed{(\{q_0\}, \{q_1\}, \{q_2\})} \end{array}$$

By applying the appropriate colour rules depending on the colour of the predecessor component and the acceptance or non-acceptance of the component itself, $\{q_0\}$ gets colour 0, $\{q_1\}$ gets colour 1, and $\{q_2\}$ gets colour 2. The resulting state is $(\{q_0\}, \overline{\{q_1\}}, \overline{\overline{\{q_2\}}})$, which is identical to its predecessor state. Consequently, we just add an a-loop to $(\{q_0\}, \overline{\{q_1\}}, \overline{\overline{\{q_2\}}})$.

In Figure 3.6 (d), we add the successor of the state $(\{q_0\}, \overline{\overline{\{q_1\}}}, \{q_2\})$. After the state generation procedure that generates $(\{q_0\}, \{q_1\}, \{q_2\})$, again case B of the colour rules in Figure 3.5 applies. The predecessor component relations are as follows:

$$\begin{array}{c} \boxed{(\{q_0\}, \overline{\overline{\{q_1\}}}, \{q_2\})} \\ \downarrow a \\ \boxed{(\{q_0\}, \{q_1\}, \{q_2\})} \end{array}$$

By applying the colour rules, $\{q_0\}$, gets colour 0, $\{q_1\}$ gets colour 1, and $\{q_2\}$ gets colour 0. This results in the state $(\{q_0\}, \overline{\{q_1\}}, \{q_2\})$. It is interesting to see that between these two states, the 2-coloured component $\overline{\overline{\{q_1\}}}$ disappears, because it has no successors. This disappearance is reflected in the fact that $(\{q_0\}, \overline{\{q_1\}}, \{q_2\})$ contains a 1-coloured component but no 2-coloured components, which can only be the case if a 2-coloured component disappears (as we explained above).

Still in Figure 3.6 (d), we create the a-successor of this new state $(\{q_0\}, \overline{\{q_1\}}, \{q_2\})$. After the state generation procedure, which results in $(\{q_0\}, \{q_1\}, \{q_2\})$, we have to apply case A of the colour rules in Figure 3.5, because $(\{q_0\}, \overline{\{q_1\}}, \{q_2\})$ does not contain a 2-coloured component. The component relations between the two states are as follows:

$$\begin{array}{c} \boxed{(\{q_0\}, \overline{\{q_1\}}, \{q_2\})} \\ \downarrow a \\ \boxed{(\{q_0\}, \{q_1\}, \{q_2\})} \end{array}$$

By applying the appropriate colour rules, $\{q_0\}$ gets colour 0, $\{q_1\}$ gets colour 2, and $\{q_2\}$ gets colour 0, which results in the state $(\{q_0\}, \overline{\overline{\{q_1\}}}, \{q_2\})$. This state already exists in the automaton, and thus we only add an a-transition to it.

At this point, all the states in the automaton have been processed. The last thing that remains to do is to define the accepting set. The rule is that every state of the lower part that does not contain any 2-coloured components must be made accepting. In our case, this applies only to the state $(\{q_0\}, \overline{\{q_1\}}, \{q_2\})$, and thus, this is the only accepting state of the automaton.

The resulting automaton, as depicted in Figure 3.6 (d), is the final complement automaton B. It is easy to verify that this automaton is indeed the complement of the input automaton A in Figure 3.3. Both automata have an alphabet size of one, and thus work only on the word $\Sigma^\omega = \{\alpha^\omega\}$. Whereas A does not accept this word (A is empty), B accepts it (B is universal), and is thus the complement of A.

A loose upper bound for the worst-case state complexity of the entire construction (including the construction of both, the upper and the lower part) has been calculated to be in $O((1.59n)^n)$ [1]. This result is however subject ongoing research, and may be sharpened to smaller number in future work.

This concludes our explanation of the basic Fribourg construction. In the next section, we present three optimisations that can be applied to the Fribourg construction in order to increase its performance.

3.2 Optimisations

The authors of the Fribourg construction propose three optimisations for the construction [1]. In this section, we briefly describe each of them. Further details about all the optimisations can be found in [1].

For easy reference, we define the abbreviations R2C, M1, and M2 for these three optimisations. The optimsiations can be added to the construction like add-ons, and they can be combined in different ways. This results in different *versions* of the Fribourg construction. Throughout this thesis, we specify a specific version of a construction by appending the included optimisations after the construction name. For example, Fribourg+R2C means the Fribourg construction including the R2C optimisation, and Fribourg+M1+R2C means the Fribourg constrution including the M1 and the R2C optimisation.

3.2.1 R2C: Omission of States Whose Rightmost Component is 2-Coloured

The R2C optimisation can be summarised as follows:

If the input automaton is complete, then states of the lower part whose rightmost component has colour 2 can be omitted.

It is important to highlight that his optimisation can only be applied if the input automaton is complete. We hint at this restriction by the letter C in R2C. If the completeness of the input automaton is given, then the optimisation allows to remove all the states whose rightmost component is 2-coloured.

The reason that it is allowed to remove these states is as follows. If the input automaton is complete, then evey input-state has at least one successor state on every alphabet symbol. This means that in the corresponding reduced split trees, that are used for the Fribourg construction, the rightmost nodes are guaranteed to have a successor node. Note that this guarantee applies only to the rightmost nodes, because the successors of the other nodes might be omitted in the right-to-left processing of the slices. Consequently, in terms of states, all rightmost components are guaranteed to have a successor component.

If the colour of the rightmost component is 2, then all successor components inherits this colour (see the rule on line 3 of Figure 3.5 (b)). Since rightmost colours *always* have a successor, *all* the successor states with a rightmost 2-coloured component will contain a 2-coloured component. And since states containing a 2-coloured component are non-accepting, these stats form a non-accepting cycle in the automaton. Consequently, it is safe to remove this cycle without changing the language of the automaton.

In our example from the last section, we could thus remove the states $(\{q_0, q_2\}, \overline{\overline{\{q_1\}}})$ and $(\{q_0\}, \overline{\{q_1\}}, \overline{\overline{\{q_2\}}})$ from the output automaton in Figure 3.6 (d). However, remember that this is only allowed if the input automaton is complete, which in our case is true for our input automaton in Figure 3.3. The R2C optimisation is described in [1] (Section 4.F).

3.2.2 M1: Merging of Adjacent Components

The M1 optimisation allows the merging certain combinations of adjacent components depending on their colour. By merging, we mean the creation of the union of two adjacent sets. Components can be merged in three cases in the following way:

1. Two adjacent 1-coloured components can be merged to a single 1-coloured component
2. Two adjacent 2-coloured components can be merged to a single 2-coloured component
3. Two adjacent 2-coloured and 1-coloured components, where the 1-coloured component is to the right of the 2-coloured component, can be merged to a single 2-coloured component

These mergers can be done recursively. This means that the result of a merger is in turn subject to further mergers. For example, given the state $(\{q_0\}, \overline{\overline{\{q_1\}}}, \overline{\overline{\{q_2\}}}, \overline{\{q_3\}}, \overline{\{q_4\}}, \overline{\overline{\{q_5\}}})$, the following chain of mergers is possible:

$$
\begin{aligned}
(\{q_0\}, \overline{\overline{\{q_1\}}}, \overline{\overline{\{q_2\}}}, \overline{\{q_3\}}, \overline{\{q_4\}}, \overline{\overline{\{q_5\}}}) \quad &\longrightarrow \quad (\{q_0\}, \overline{\overline{\{q_1, q_2\}}}, \overline{\{q_3, q_4\}}, \overline{\overline{\{q_5\}}}) && \text{By rules 1 and 2} \\
&\longrightarrow \quad (\{q_0\}, \overline{\overline{\{q_1, q_2, q_3, q_4\}}}, \overline{\overline{\{q_5\}}}) && \text{By rule 3} \\
&\longrightarrow \quad (\{q_0\}, \overline{\overline{\{q_1, q_2, q_3, q_4, q_5\}}}) && \text{By rule 2}
\end{aligned}
$$

This means that with the M1 optimisation, we add the state $(\{q_0\}, \overline{\overline{\{q_1, q_2, q_3, q_4, q_5\}}})$ to the automaton rather than $(\{q_0\}, \overline{\overline{\{q_1\}}}, \overline{\overline{\{q_2\}}}, \overline{\{q_3\}}, \overline{\{q_4\}}, \overline{\overline{\{q_5\}}})$. For the formal description and proof of correctness of the M1 optimisation, we refer to [1] (Section 4.E).

The application of these mergers obviously reduces the maximum number of states that the construction can produce. The worst-case state growth regarding the number of states of the lower part has been calculated in [1] to be $(1.195n)^n$. This is added to a worst-case state growth of the upper part of $(0.53n)^n$. It will be a subject of our empirical performance investigation in Chapter 4 to investigate the impact of the M1 optimisation on *acutal* automata.

3.2.3 M2: Single 2-Coloured Component

The M2 optimisation is the most complicated of the three optimisations. Its effect on the output automaton can be summarised as follows:

States of the lower part contain at most one 2-coloured component.

Note that without the M2 optimisation, it is possible that states contain two or more 2-coloured components. This can happen if the predecessor state contains no 2-coloured components and the new state contains multiple accepting components (see colour rules on line 1 and 2 of Figure 3.5 (a)). With the M2 optimisation, in this case only one of the accepting components gets colour 2, and the others get colour 1. The purpose of the M2 optimisation is to further reduce the maximum number of states that the construction can produce.

It is important to note that the M2 optimsiation is dependent on the M1 optimisation. That means that the M2 optimisation can only be applied if the M1 optimisation is also applied. We selected the abbreviation M2 for this optimisation to highlight that it is based on M1.

Below we first explain how a single 2-coloured component of a state is created. Next, we explain in detail the steps that the disappearance of this single 2-coloured component entails.

Creation of a 2-Coloured Component

As mentioned, when creating a new state, only one of the components, that without the M2 optimisation would become 2-coloured, gets colour 2. This component is defined to be the rightmost of the candidates for colour 2. A possible procedure to ensure this restriction is to determine the colours for the components from right to left, and assign colour 2 only if it has not been previously assigned in the state. If it has been previously assigned, then assign colour 1 instead of 2.

A special case applies if the component that gets assigned colour 2 has a 2-coloured predecessor component and a sibling to the left of it. With sibling we mean an adjacent component that has the same predecessor component. If this two criteria are fulfilled, then the sibling gets colour 2 as well. Note that without this special treatment, it would get colour 1 instead of 2. Thus, there are two 2-coloured components in the state. However, since the M2 optimisation also requires the M1 optimisation to be applied, these two 2-coloured components are merged to a single 2-coloured components. In this way, statement that all states have at most one 2-coloured components is still true.

Actions on the Disappearence of a 2-Coloured Component

The restriction of having only one 2-coloured component in a state entails special actions when this 2-coloured components disappears (that is, has no successor components). Imagine a state p with a single 2-coloured component, and its successor state q. If after the assignment of colours as described above, q has no 2-coloured component, then we have the situation that the 2-coloured component of p disappeared. The required action in this situation is to change the colour of one of the 1-coloured components of q (if there are any) to 2. If q does not contain any 1-coloured components, then nothing is done and the creation of the state is finished. If q contains exactly one 1-coloured component, then the colour of this component is changed to 2. If q contains multiple 1-coloured components, then the following procedure is applied.

- Go to the position in q where the successor of the disappeared 2-coloured component of p *would* be
- From this position, go to the left until arriving at a 0-coloured component
  - If the search arrives at the leftmost component, continue with the rightmost component
- From this 0-coloured component, go to the left until arriving at the first 1-coloured component
  - If the search arrives at the leftmost component, continue with the rightmost component
- Change the colour of this 1-coloured component to 2

That is, if there are multiple 1-coloured components in the state, then we select the first 1-coloured component that is to the left of the first 0-coloured component that is to the left of the position where the successor of the disappeared component would be, and change its colour to 2. In the course of this, we understand the relation “to the left of” as cyclically, such that the component to the left of the leftmost component is the rightmost component.

The effect of changing the colour of a 1-coloured component of a state q to 2 is that q, which previously contained no 2-coloured components, now contains a 2-coloured component. However, since a 2-coloured component disappeared, q must still be accepting, even though it contains a 2-coloured component. For this reason, we marks states such as q with a star ($*$). In general, each state that has been subject to a change of a 1-coloured component to a 2-coloured component is marked with a star. Finally, the rule for adding states to the accepting set is changed such that it includes all the states of the lower part that contain no 2-coloured component *and* all the states that contain a 2-coloured component but are marked with a star.

Figure 3.7 shows the result of applying the Fribourg construction with the M1 and M2 optimisations to the example automaton in Figure 3.3. The difference of this automaton to the complement resulting from the Fribourg construction without the M1 and M2 optimisation (see Figure 3.6 (d)) is the state at the bottom right of the automaton. As can be seen in the figures, without the optimisation the component $\{q_1\}$ of this state is 1-coloured. With the M1 and M2 optimisations, however, $\{q_1\}$ is 2-coloured. In both cases, the 2-coloured component of the predecessor state disappeared, however, with the M1 and M2 optimisations, the resulting 1-coloured $\{q_1\}$ is made 2-coloured. Additionally, the state is marked with a star, so that it is still part of the accepting set. Regarding the successors of this marked state, the same procedure is repeated indefnitely: the 2-coloured component disappears, and one of the 1-coloured components is made 2-coloured. This is the reason why this state has a loop.

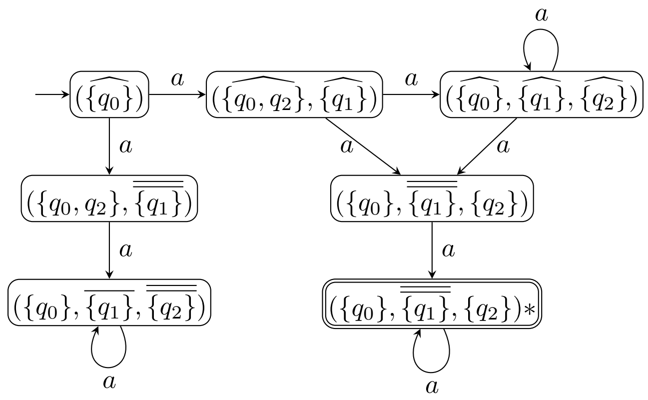


Figure 3.7: Result of applying the Fribourg construction with the M1 and M2 optimisation to example automaton in Figure 3.3.

The M2 optimisation further reduces the maximum number of states that the construction can produce. A loose upper bound of $(0.86n)^n$ for the worst-case state growth of the lower part of the automaton has been calculated in [1] (Section 4.H). However, calculations of actual values suggest that the real bound for the lower part might be as low as $(0.76n)^n$. The investigation of the worst-case state complexity of the M1 and M2 optimisation is subject to ongoing research by the authors of the Fribourg construction. Note that $(0.76n)^n$ number of states in the lower part, and thus for the overall complexity of the construction , the size of the upper part of $(0.53n)^n$ must be added to it.

3.3 General Remarks

The Fribourg construction has many similarities to the slice-based construction by Vardi and Wilke[64] (see Section 2.3.4) that has been published in 2007. The upper part and lower part of the Fribourg construction correspond to the initial phase and repetition phase of Vardi and Wilke's construction. Furthermore, the colours 0, 1, and 2 of the Fribourg construction correspond to the decorations *inf*, *new*, and *die* of Vardi and Wilke's construction. However, the Fribourg construction has been developed independently, and is not based on Vardi and Wilke's construction.

Rather, the Fribourg construction is based on Kurshan's complementation construction for *deterministic* Büchi automata [26]. This construction also uses an upper part and a lower part which are both deterministic, except for non-determnistic transitions from the states of the upper part to states of the lower part. The Fribourg construction has emerged as an adaptation of Kurshan's construction for non-determnistic Büchi automata, with the slice-based run analysis appraoch in mind.

With these remarks, we conclude the presnt chapter about the Fribourg construction. At this point, we have provided all the background that is needed for our empirical performance investigation of the Fribourg construction, which we present in the next chapter.

Chapter 4

Empirical 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. The aim of this investigation is to find out how the Fribourg construction performs on *actual* automata. This is opposed to the theoretical investigation of the worst-case performance, which reveals how the construction performs on a specific *worst-case* automaton. The goal of this chapter is to describe in detail the empirical study that we carried out. The actual results of the investigations are presented in Chapter 5.

For investigating the performance of the Fribourg construction, we proceed in two tracks. First, we want to test the Fribourg construction with its different combinations of optimisations and compare them to each other. Second, we want to compare the Fribourg construction with other complementation constructions. We refer to the first track of investigation as the *internal tests*, and to the second track as the *external tests*. Note that our main performance measure is the number of states of the complement automaton. However, we sometimes also use the execution times of complementation taks as a secondary measure.

For an empirical performance investigation of a Büchi complementation construction, one needs basically three things: an implementation of the construction, test data, and an experimental setup, including an execution environment. Regarding the *implementation*, we decided to create it as a part of the GOAL tool. GOAL is an ω-automata tool, which has already been used previously for empirical investigations of Büchi complementation constructions (see Section 1.2.2). Furthermore GOAL contains many pre-implemented complementation constructions, so that, for the external tests, we can run all the tested complementation constructions in a common framework.

Test data means basically a set of concrete automata that are given as input to a construction, in order to analyse the output of the construction. We use two different test sets for our investigation, that we refer to as the GOAL test set and the Michel test set (consisting of Michel automata [29]). The automata of both test sets have been used in previous investigations of Büchi complementation construction by other authors (see Section 1.2.2). We believe that the use of test data that has been previously used increases the validity of our results, as they can be compared to related work.

Our *experimental setup* includes a definition of the concrete tests that are run, that is, which construction, or version of a construction, is executed on which test data. It also states time and memory limits that are impopsed on the experiments, and specifies the execution environment. Regarding the execution environment, since our study includes in large computations, we decided to use professional high-performance computing (HPC) facilities. Concretely, we execute all the experiments on a Linux-based HPC cluster named UBELIX of the University of Bern[1].

In this chapter, we discuss each of the mentioned points in more detail. We treat the implementation of the Fribourg construction in Section 4.1, the test data in Section 4.2, and the experimental setup in Section 4.3.

4.1 Implementation

As mentioned, we implemented the Fribourg construction as part of the GOAL tool. In this section, we first present the GOAL tool in a general way (Section 4.1.1). In Section 4.1.2, we describe how we implemented the Fribourg construction with its optimisations as a part of this tool. Clearly, an implementation is only useful if it is correct. Therefore, in Section 4.1.3, we describe how we verified the correctness of our implementation.

4.1.1 The GOAL Tool

GOAL stands for *Graphical Tool for Omega-Automata and Logics* and is being developed at the Department of Information Management at the National Taiwan University (NTU)[2]. The tool has been described in various scientific publications [55, 56, 57, 54]. The GOAL tool is freely available on

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

http://goal.im.ntu.edu.tw. It is a Java program, and thus runs on every platform having a Java Runtime Environment (JRE).

GOAL is a graphical and interactive tool for creating and manipulating ω-automata. It provides a large number of operations that can be applied to these automata. These operations range from input testing, conversions to other types of ω-automata, to union and intersection. Figure 4.1 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 implementations of many existing Büchi complementation constructions. We will discuss the complementation constructions that GOAL provides further below.

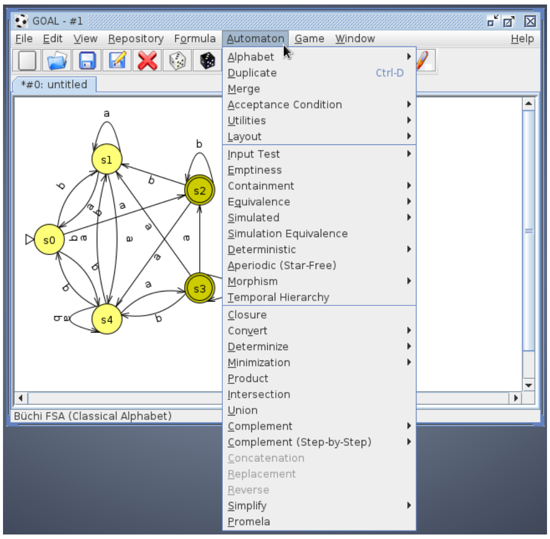


Figure 4.1: The graphical user interface of GOAL (version 2014–11–17). The open menu item gives an idea about the different types of manipulations that can be applied to ω-automata.

The versions of GOAL have names of the form YYYY–MM–DD that correspond to the release date of the version. The latest version at the time of this writing, and the version that we used in our experiments, is 2014–11–17.

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

As mentioned, GOAL has a graphical user interface, as shown in Figure 4.1. However, almost the entire functionality of GOAL is also available through a command line interface. For example, complementing an automaton can then be done with the command `gc complement -m safra -o out.gff in.gff` from the commandline. The effect of this command is to complement the automaton in the file `in.gff` with Safra's complementation construction, and write the complement automaton to the file `out.gff`. The command `gc` is the entry point to GOAL's command line interface. For our empirical performance investigation, we use the command line interface of GOAL.

Let us now have a closer look at the Büchi complementation constructions that are pre-implemented in GOAL. The 2014–11–17 version of GOAL provides a number of 10 complementation constructions. We list these constructions along with their authors and reference to the literature in Table 4.1.

As can be seen in Table **??**, almost all complementation constructions that GOAL provides have been reviewed in Section 2.3. The only exception is ModifiedSafra, which is a minor modification to Safra's construction that has been proposed by Althoff et al. [2]. Furthermore, GOAL has at least one construction of each of the four main complementation approaches. Ramsey belongs to the Ramsey-based approach, Safra, ModifiedSafra, Piterman, and MS belong to the determnisation-based approach, Rank, WAA, and WAPA to the rank-based approach, and Slice and Slice+P to the slice-based approach. Note that the identifiers that are defined in Table 4.1 for each construction are also used by GOAL, and we will use them throughout this thesis to refer to the corresponding constructions.

Identifier	Description	Authors (year)	Reference
Ramsey	Ramsey-based construction	Sistla et al. (1987)	[45, 46]
Safra	Safra's construction	Safra (1988)	[42, 43]
ModifiedSafra	Slightly modified Safra's construction	Althoff et al. (2006)	[2]
Piterman	Safra-Piterman construction	Piterman (2007)	[36, 37]
MS	Muller-Schupp construction	Muller, Schupp (1995)	[34]
Rank	Rank-based construction	Schewe (2009)	[44]
WAA	Via weak alternating automata	Kupferman, Vardi (1997/2001)	[24, 25]
WAPA	Via weak alternating parity automata	Thomas (1999)	[52]
Slice+P	Slice-based construction	Vardi, Wilke (2007)	[64]
Slice	Enhanced slice-based construction	Kähler, Wilke (2008)	[20]

Table 4.1: The Büchi complementation constructions provided by GOAL version 2014–11–17.

The construction Slice+P is actually the construction Slice with the P option. That is, from the perspective of GOAL, there is only the single construction Slice, however, if the P option is specified, then the slice-based construction by Vardi and Wilke [64] is used, and otherwise the slice-based construction by Kähler and Wilke [20] is used.

At this point, it is worth pointing at a related project of the same research group, called the *Büchi Store* [58]. This is an online repository of categorised ω-automata that can be downloaded in different formats (including GFF). The Büchi Store is accessible over the web on http://buchi.im.ntu.edu.tw/. Furthermore, there is a binding in GOAL to the Büchi Store, so that automata can be directly downloaed into GOAL. For our study we did not use the Büchi Store, but it is might be an interesting option for future work.

4.1.2 Implementation of the Construction

The GOAL Plugin Architecture

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 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 plugin writers need 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.

It is thanks to this open architecture of GOAL that welcomes extension that we were able to seamlessly add the Fribourg construction to this tool. In particular, we created a GOAL-plugin that contains the

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

Option	Description
R2C	Apply R2C optimisation
M1	Apply M1 optimisation
M2	Apply M2 optimisation
C	Make input automaton complete
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 4.2: The options for the Fribourg construction in our Fribourg construction plugin for GOAL.

implementation of the Fribourg construction. We present this plugin below.

The Fribourg Construction Plugin and Its Options

The name of our plugin containing our implementation of the Fribourg construction is ch.unifr.goal.complement[4]. The plugin is freely available and can be installed by any GOAL user. The installation in GOAL is very easy. We give instructions on how to obtain, install, and use the plugin in Appendix A.

We aimed at fully integrating the Fribourg construction in GOAL. The current version of the plugin includes menu integration of the Fribourg construction, the pertinent saving of option defaults, and a step-by-step execution facility for the Fribourg construction. This last point is a great way for learning how the Fribourg construction works.

Our implementation of the Fribourg construction also includes the three optimisations R2C, M1, and M2 that we described in Section 3.2. These optimisations are freely selectable by the user and can be combined in different ways. In the GUI, the optimisations are presented to the user as selectable check boxes before the start of the construction. In the command line mode, they can be set as flags of the entered command.

In addition to these three optimisations, we added further options to our construction. Table 4.2 provides a listing of all of them. Note that each option has an identifier consisting of upper-case letters. We use these identifiers throughout the rest of this thesis to refer to the corresponding options.

We discuss each of the options in Table 4.2 below. The first three options are the three optimisations to the Fribourg construction that we described in Section 3.2. The R2C optimisation is implemented so that it applies only to input automata that are complete. That is, if R2C is activated for the complementation of an automaton that is not complete, then this option has no effect. The options M1 and M2 stand for 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.

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, and if this is not the case, makes it complete by adding a sink state. The purpose of the C option is to be used in conjunction with the R2C optimisation. Making an automaton complete before the start of the construction ensures that the R2C optimisation will be applied. The question arises whether, in terms of performance, it is worth to do this, because making an automaton complete increases its size by one, what increases the potential size of the output automaton. This question has been investigated in previous work on the Fribourg construction by Göttel [14]. In this thesis will re-investigate this point in an extended form (see Section 4.3).

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

[4] By convention, JPF plugins are named after the corresponding Java package.

of the automaton. The pre-implemented complementation constructions Ramsey, Piterman, Rank, and Slice contain a similar R option.

An exmple usage case of the R option is to investigate the number of unreachable and dead state that a complementation construction produces. A way to do this is to complement the same automaton with and without the R option, and calculating the difference of the sizes of the two complements. Such investigations have been done with by Tsai et al. [53] in their comparative study of several Büchi complementation constructions in GOAL. We will do similar investigations, although it is not the main focus of our study.

The RR option is similar to the R option, with the difference that it removes the unreachable and dead states from the input automaton rather than from the output automaton. This option is not common among the other complementation constructions in GOAL, and we are not using it in our study.

The MACC option is again an option that modifies the input automaton before the start of the construction. It maximises the accepting set of the input automaton. This means that as many states as possible are made accepting so that the language of the automaton is not changed. The idea is that the larger number of accepting state simplifies the complementation task. This technique has been introduced and empirically investigated by Tsai et al. [53]. The pre-implemented complementation constructions Ramsey, Piterman, Rank, and Slice also contain the MACC option. In our study we will however not use the MACC option.

Finally, the B option only affects the formatting of the labels of the output-states. It has no effect on the consruction itself. In particular, it activates the bracket notation, which is an informal alternative way to indicate the colours of components of states. It does so by using different types of brackets for the subsets, rather than wrapping the subset in a tuple with the colour number, like the default notation.

4.1.3 Verification of the Implementation

It is important to verify that our implementation of the Fribourg construction is correct. The construction is correct if the output automata it produces are really the complements of the corresponding input automata. Note that this is not the same sense of correctness as the verification that our implementation faithfully represents the specification of the construction. However, with regard to the fact that our implementation produces correct complements, we believe that this is also the case.

In order to verify that our implementation produces correct complements, we performed so called complementation-equivalence tests in GOAL. This works as follows. Take an automaton A and complement it with one of the existing constructions in GOAL, resulting in the complement automaton B'. Then, complement the same automaton A with our implementation of the Fribourg construction, which yields the automaton B''. Now, by the means of GOAL's equivalence operation test whether A' and A'' are equivalent. If yes, then our implementation produced a correct complement.

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 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 issues, a sufficient number of passed tests, still makes us sufficiently confident that our implementation is correct.

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 that we used as the ground-truth is Piterman. The result was that all the tests were successful.

With four states, the automata used for the tests are rather small, and it would be interesting to do the tests with larger 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 in turn is based on complementation. This means that the equivalence test includes the complementation of the complements of the test automata. By using larger test automata, their complements might be already so large that a further complementation is practically infeasible with our available resources. Nevertheless, we think that the number of passed tests with these small automata indicates the absence of implementation errors with a high probability.

4.2 Test Data

After an implementation, the test data is the second main ingredient of an empirical performance investigation. The test data consists of the sample automata that are given as input to the construction, in order to analyse the produced 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 significant, it should not be biased in favour of the tested algorithm, and it should cover test scenarios that are relevant for evaluating the performance of the tested algorithm. The ideal case is to use publicly available test data, that has been previously used for other studies. In this way, the test data can be critically inspected by everybody, and furthermore the results of the study are comparable to the results of other studies.

We decided to use two sets of test data that have both been used in other studies. The first is the GOAL test set, that has been created by Tsai et al. [53], and used in other studies, such as [7] and [14]. It consists of 11,000 non-deterministic Büchi automata of size 15. The second set of automata, which we call Michel test set, consists of the smallest four Michel automata. Michel automata are the automata that Michel used to prove the lower bound of $n!$ for the worst-case state complexity of Büchi complementation [29]. The Michel automata have been used as the test data for other empirical studies, such as [2], or [14]. In the following subsections, we first describe the GOAL test set, and then our used Michel test set.

4.2.1 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.

Structure

The GOAL test set has been created by Tsai et al. [53] for an empirical study evaluation the effects of several optimisations on existing Büchi complementation constructions. This study has been executed in GOAL and the automata in the test set are available in the GOAL file format. This is why we call the data GOAL test set.

The GOAL test set consists of 11,000 automata of size 15[5]. It is 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 (consising of the symbols 0 and 1). The further properties of the automata, namely

[5]There exists a second version with 11,000 automata of size 20.

[6]This link is maintained by the author of the thesis. The link to the original location of the GOAL test set is http://goal.im.ntu.edu.tw/wiki/lib/exe/fetch.php?media=goal:ciaa2010_automata.tar.gz. This package contains additionally the second version of the GOAL test set with the 11,000 automata of size 20.

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 particular, 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. For example, an automaton of the test set with an acceptance density of 0.5 contains 8 accepting states.

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 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. According to [53], these parameter values were chosen to provide a broad range of complementation problems ranging from easy to hard.

Completeness, Universality, and Emptiness

We tested all the automata in the GOAL test set for completeness, universality, and emptiness. It is interesing to know about these special cases of automata because they may have an effect on the complementation results. The ideal complement of an empty or universal automaton has a size of one. Thus, by comparing this against the number of actually produced states, the number of "superfluous" states that the construction generated becomes apparent. Regarding completeness, the R2C optimisation applies only to complete automata, thus it is interesting to see on which automata this optimisation has in fact an effect.

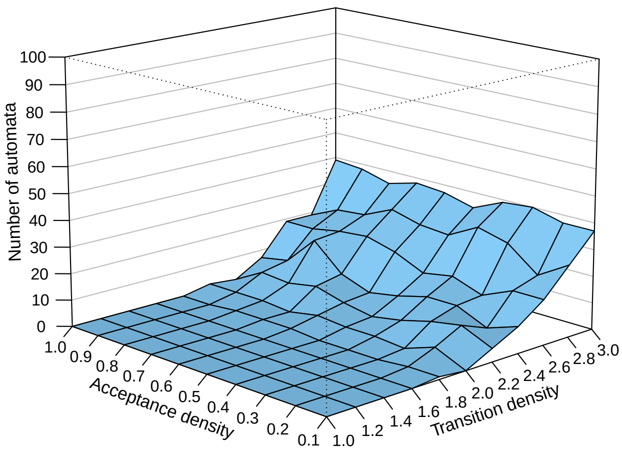
GOAL provides a command for testing emptiness. However, it does not provide commands for testing completeness and universality. Therefore, we implemented these commands on our own and bundled them as a separate GOAL plugin. This plugin is called ch.unifr.goal.util and is also available as described in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

With these GOAL commands we tested and recorded for each of the 11,000 automata whether it is complete, unviversal, or empty. We then determined the number of complete, universal, and empty automata in the entire test set, as well as in each of the 110 transition density/acceptance density classes. The number of complete, universal, and empty automata in the entire test set of 11,000 automata are as follows:

- 990 complete automata (9%)
- 6,796 universal automata (61.8%)
- 63 empty automata (0.6%)

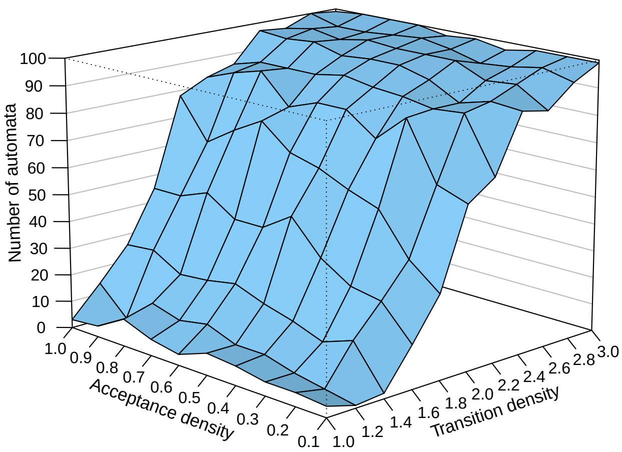
The fact that only 9% of the automata are complete means that the R2C optimisation of the Fribourg construction affects only 9% of the test data. This is an interesting fact for the analysis of the effect the R2C optimisation has on the overall performance of the construction. A surprisingly high number of 61.8% of the automata are universal. A reason for this might be the small alphabet consisting of only two symols of the automata. 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

	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



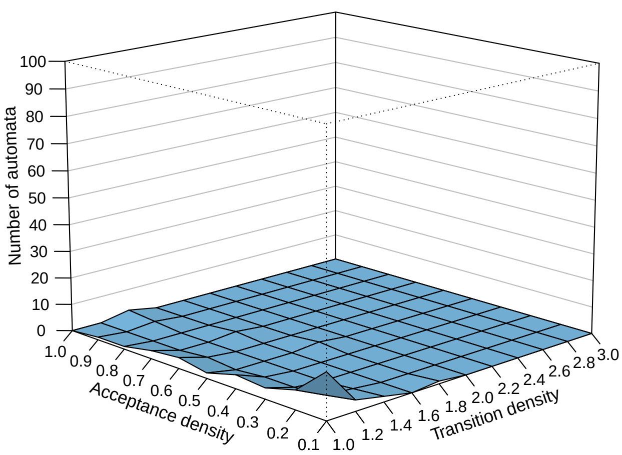
(a) Number of *complete* automata per class.

	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 4.2: Number of complete, universal, and empty automata in each of the 110 transition density/acceptance density classes of the GOAL test set. Note that each class contains 100 automata.

low. This can be seen as the reverse of the same effect that causes the number of universal automata to be high.

In Figure 4.2, we show the number of complete, universal, an d empty automata for each of the 110 classes. By the way, in this figure we introduce two ways for representing per-class data that we will use throughout this thesis. On the left side of Figure 4.2 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 4.2, 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 at the corresponding matrix from beyond 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 5, we mainly use perspective plots, however, we provide all the corresponding matrices in Appendix B.

Regarding the complete automata per class in Figure 4.2 (a), we can see that 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 can similarly be viewed as percentages. 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 4.2 (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 4.2 (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 investigate the number of universal automata in automata with a bigger alphabet size. We let this as an idea for future work.

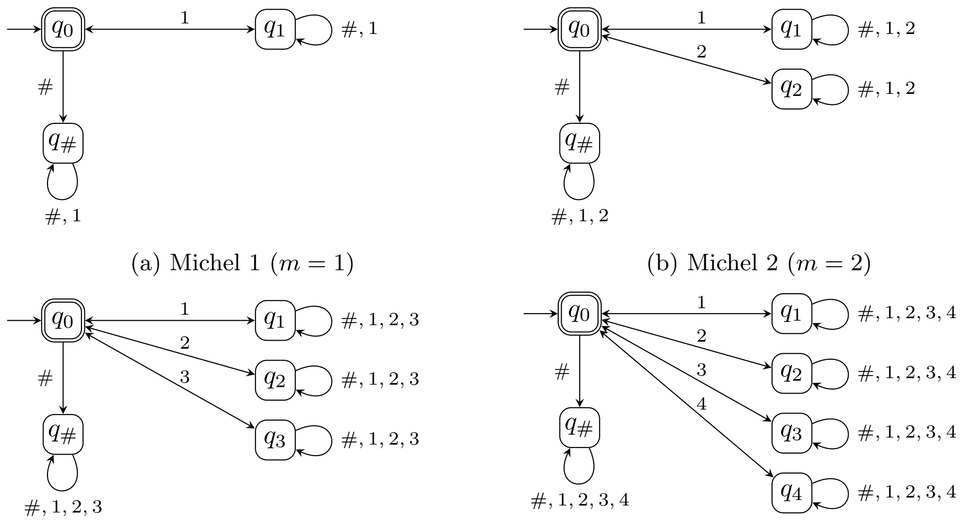
Conversely to the high number of universal automata, the number of empty automata is very low. The totally 63 empty automata are mainly concentrated in the upper-left corner of the matrix in Figure 4.2 (c). That is, the automata with a low transition density and a low acceptance density have the highest probability to no accept any word, and thus being empty. The reasons for this is roughly the opposite reasons for the distribution of the universal automata.

4.2.2 Michel Test Set

The Michel test set is very different from the GOAL test set. It consists of only four automata, that however exhibit an exceptionally high state growth. Michel automata have been introduced in 1988 by Michel [29] in order to prove a lower bound for the worst-case state complexity of Büchi complementation of $(n-2)!$ [51]. The Michel autoamate are characterise by the parameter m. They have an alphabet size of $m+1$, and $m+2$ states. Michel 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 minimum state growth of these automata is $(n-2)!$.

In practice, however, the state growth that is obseved for Michel automata is much higher. It is so high that for practical reasons we are restricted to include only the first four Michel automata 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 extrapolations toward larger Michel automata in Section 5.1.2. Despite the small number of tested automata, we still obtain very interesting results, as we will see in Chapter 5.

The four Michel automata in our test set are shown in Figure 4.3. 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.



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

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

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

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

Figure 4.3: The Michel automata with $m = \{1, \ldots, 4\}$, an alphabet size of $m + 1$, and $m + 2$ states.

It is interesing to use Michel automata to investigate the performance of Büchi complementation constructions, because they force the constructions to generate large number of states. By furthermore using several Michel automata with different sizes, it is possible to fit a state growth function on the resulting complement sizes. The state growth observed from the Michel automata is most likely not equal to the worst-case state growth (there are even worse automata), but it still provides a lower bound for the worst-case state growth. This can be interesting in case the worst-case state growth has not yet been precisely determined. This function will certainly not be equal to the theoretical state growth function, but it still provides a lower bound for the worst-case state growth

4.3 Experimental Setup

In this section, we describe the experimental setup of our empirical performance investigation. This includes the specification of the concrete constructions (or versions of constructions) that are tested, constraints that are imposed on the runs, and the execution environment. As mentioned, the investigation is 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 of the versions of the Fribourg construction with a set of different complementation constructions.

In Section 4.3.1, we present the tested verions for the internal tests. These versions differ for the GOALand the Michel test set, and we present them separately. In Section **??**, we present the tested constructions (including one of the versions of the Fribourg construction) for the external tests. In Section 4.3.4, we describe the computing environment in which the experiments were executed. Finally, in Section 4.3.3, we present the time and memory limits that were imposed on the individual executions.

4.3.1 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 4.2). The sets of versions are different for the GOAL and the Michel test set.

GOAL Test Set

For the internal tests on the 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

The first version, Fribourg, is the plain Fribourg construction without any optimisations or options. The next two versions, Fribourg+R2C and Fribourg+R2C+C, aim at investigating the R2C optimisation. In Fribourg+R2C, the R2C optimisation is applied only to complete input automata. As we have seen in Section 4.2.1 these are just 9% of the automata. Fribourg+R2C+C, on the other hand, makes the incomplete input automata complete so that the R2C optimisation is applied to all automata. The fact that we want to find out is whether it is worth to increase the size of an automaton by one (for adding the sink state) for the sake of being able to apply the R2C optimisation.

A very similar question has been investigated in previous work on the Fribourg construction by Göttel [14]. In terms of our above listing, he compares the performance of Fribourg with Fribourg+R2C+C. As the test data, he also uses the GOAL test set. His results are that the overall mean complement size resulting from Fribourg+R2C+C is higher than for Fribourg. However, by looking closely at Göttel's results we suppose that the median complement size (which is not reported in [14]) might be lower for Fribourg+R2C+C than for Fribourg. This would be an interesting relation, and therefore, we decided to re-investigate this question.

Fribourg+M1 and Fribourg+M1+M2 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 5, Fribourg+M1 shows a better performance on the GOAL test set than Fribourg+M1+M2. Therefore, we do not further investigate any versions based on Fribourg+M1+M2, however, we further investigate versions based on Fribourg+M1. This is why the next two versions are Fribourg+M1+R2C and Fribourg+M1+R2C+C. With these versions, we investigate the effect of the R2C optimisation in combination with the M1 optimisation. The last version, Fribourg+R, corresponds to the first version, Fribourg, but with the difference that in Fribourg+R 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 Fribourg.

The versions Fribourg+M1+R2C and Fribourg+M1+R2C+C enhance the "better" of Fribourg+M1 and Fribourg+M1+M2 with R2C and R2C+C, respectively. As we will see in Chapter 5, the better one of these two versions terms of median complement sizes is Fribourg+M1. That is, the application of M2 results in a decline, rather than a gain, in performance compared to the application of M1 alone. It is imporant to highlight that such results are specific to the used test data, and not universally valid. With different test data Fribourg+M1+M2 might be better than Fribourg+M1. As we will see in the next section, this is for example the case for our Michel test.

The last version, Fribourg+R, corresponds to the application of the plain Fribourg construction without any optimsations or options, but with the difference that at the end of the construction all unreachable and dead states are removed from the complement automaton. Comparing the results of Fribourg+R with the results of Fribourg reveals how many unreachable and dead states the Fribourg construction produces. This idea is inspired by the study by Tsai et al. [53] in which the number of unreachable and dead states is one of the main performance metrics for the tested constructions..

Michel Test Set

For the internal tests on 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 rationale behind the selection of 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. This is why Fribourg+M1+M2+R2C is based on Fribourg+M1+M2 rather than Fribourg+M1.

4.3.2 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 other complementation constructions that are implemented in GOAL.

Regarding the other complementation constructions, we could theoretically include all the constructions listed in Table **??**. However, practical reasons prevent us from doing so. In preliminary tests we observed that most of these constructions are not efficient enough on the GOAL test set for our purposes. In simple terms, they take too long and use too much memory. Using these constructions would require higher time and computing resources that we can allocate. According to our tests, this excludes all but Piterman, Slice, Rank, and Safra from being reasonably used. A similar experience has been made by Tsai et al. in their own empirical study [53]. They observed that the Ramsey construction could not complement *any* of the 11,000 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 Piterman, Slice, and Rank in our external tests. These constructions furthermore represent three of the four main complementation approaches, namely the determinisation-based, rank-based, and slice-based approach. The fourth approach would be Ramsey-based, however, as mentioned the implementation of the Ramsey-based construction by Sistla et al. [45] in GOAL is not efficient enough for our study. It would have been possible to also include Safra. However, as we include Piterman, which is in fact an improvement of Safra (see Section 2.3.2, we decided that it is not necessary to additionally include Safra.

For Slice, we actually use the construction that is indicated as Slice+P in Table 4.1. Slice+P is the construction by Vardi and Wilke [64], whereas Slice is the consruction by Kähler and Wilke [20]. Since the construction by Vardi and Wilke is more efficient than the construction by Kähler and Wilke (see Section 2.3.4), we decided to use Vardi and Wilke's construction, that is Slice+P. Note that in the following, we will nevertheless refer to this construction as Slice, however, we mean by this at any point Vardi and Wilke's construction [64].

Piterman, Rank, and Slice also have a set of selectable optimisations in GOAL (some of these optimisations have been proposed by Tsai et al. [53]). We decided to include the set of optimisations for each of the three constructions that is activated by default in GOAL, except the MACC and R optimisations. The reason to exclude MACC and R is that they are not part of the actual construction, but rather just modify the input automaton before the construction starts, or the output automaton after the construction ends, respectively. By otherwise using the default optimisations, we are likely to use the most optimal versions of these constructions. This makes the comparison with the Fribourg construction fair, because for the Fribourg construction we also choose the most optimal version for the external tests.

Note that for Piterman, we also exclude the SIM optimisation, even though it is set by default in GOAL. This is because the SIM optimisation of Piterman simplifies the complement DPW of the construction. This can be seen as a similar optimisation as the R option, and thus, we do not want to include it in our tests.

Using the optimisations for Piterman, Slice, and Rank according to these rules, we obtain the precise versions Piterman+EQ+RO, Slice+P+RO+MADJ+EG, and Rank+TR+RO. Regarding the Fribourg construction, we use the best versions for each of the two test sets. As we will see in Chapter 5, these are Fribourg+M1+R2C for the GOAL test set, and Fribourg+M1+M2+R2C for the Michel test set.

Consequently, for the GOAL test set, the tested constructions are:

1. Piterman+EQ+RO
2. Slice+P+RO+MADJ+EG
3. Rank+TR+RO
4. Fribourg+M1+R2C

For the Michel test set, on the other hand, the tested constructions are:

1. Piterman+EQ+RO
2. Slice+P+RO+MADJ+EG
3. Rank+TR+RO
4. Fribourg+M1+M2+R2C

4.3.3 Time and Memory Limits, and Effective Samples

We imposed a time and memory limit on every complementation task of the GOAL test set. For the Michel test set, we did not set any limits, because it consists of a much smaller number of tasks. These limits are based on the limits that have been used by Tsai et al. [53]. Our time limit is 600 seconds CPU time, and our memory limit corresponds to 1 GB Java heap. If a complementation taks is not finished after 600 seconds CPU time, or requires more than 1 GB heap memory, then the tasks aborted.

The reasons for these limits are our limited computing and time resources. Clearly, 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 high complementation complexity that Büchi automata may have, some few extreme cases may cause our available resources to be exceeded[7]. The study is for example ultimately limited by the physically available memory on the nodes, and the maximum running time of a job permitted by the cluster mangement software . 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. Processes that reach this time limit 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 generated by the tested constructions are stored on the heap. It is possible to set the maximum Java heap size with the `Xmx` option to the Java Virtual Machine[8]. We also 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 what could distort the measured execution times.

The presence of time and memory limits, and thus aborted complementation tasks, require the use of so-called *effective samples*. The concept of effective samples is also used by Tsai et al. . These limits are based on the limits that have been used by Tsai et al. [53]. The effective samples are those automata which have been successfully completed by *all* constructions that are compared to each other. Imagine, for example, two constructions C_1 and C_2 that are used to complement a test set of 1,000 automata. C_1

[7] As we will see in Chapter 5, the distribution of the complexity of the tested Büchi automata is extremely right-skewed, that is, most are easy, and very few are hard.

[8] Example usage: `java -Xmx1G`

successfully complements all the automata, whereas C_2 gets aborted for 100 of these automata. If we would statistically evaluate and compare the 1,000 results of C_1 and the 990 results of C_2, then C_2 would be advantaged, because the results of the 100 apparently hardest automata are not taken into account, whereas they are included in the evaluatio of C_1. This is why in this case only the 990 effective samples that have been successfully completed by both constructions must be evaluated. In Chapter 5, we will therefore only evaluate the effective samples of the compared constructions.

4.3.4 Execution Environment

Due to the computational intensity, we executed all the complementation tasks, that we outlined in the last section, on a HPC computer cluster. In particular, we used the computer cluster named UBELIX of the University of Bern[9]. Note that we also executed the correcntess tests for the implementation of the Fribourg construction (see Section 4.1.3) and the analysis of the GOAL test set (see Section **??**) on UBELIX.

Generally, a computer cluster consists of interlinked *nodes* (computers) on which the *jobs* (computation tasks) of the users of the clusters are run. Typically, cluster users submit their jobs to a cluster management software, which automatically dispatches these jobs to suitable and available nodes. In the case of UBELIX, this cluster management software is Oracle Grid Engine (formerly known as Sun Grid Engine) version 6.2[10].

We ensured that all our tasks are executed 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

Note that the memory of the nodes is either 64 GB or 256 GB. The tasks on the GOAL test set were run on nodes with 64 GB RAM, the tasks on the Michel test set 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 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 which was sufficient to complement all the Michel automata.

Regarding multicore usage, the behaviour of our tasks depends on GOAL, and thus ultimately on Java. GOAL is multi-threaded and thus our taks may use multiple CPUs. Theoretically, they can use up to the total number of 16 CPUs of a node. However, we observed that our tasks typically used between two and four CPUs[11].

We also measured the execution time of each complementation task as CPU time and real time (also known as wallclock time). 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). These measurements were done with Bash's `time` reserved word. 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, on the other hand, this is not possible, and the CPU time may only be equal to or lower than the real time. In the analysis of our results in Chapter 5, we sometimes present statistics of the execution times. These times are always *CPU times*.

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

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 our measured execution times, and acts like a constant. According to our observations, the JVM startup time is approximately two CPU time seconds.

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 node loads. We do not know whether this has an influence on our experiments, in particular, on the measurement of the execution times, and on the imposed time limit (see next section). The load of a node has however no influence on the complements that are produced by the complementation constructions, which is after all our main interest. Hence, we leave the topic of the influcence of node load on execution time for future work.[12]

This concludes the present chapter that described the setup of our empirical performance investigation of the Fribourg construction. We explained how we implemented the Fribourg construction as a part of the GOALtool. Then, we presented our test data, consisting of the GOAL test set and the Michel test set. Finally, we presented our experimental setup, specifying exactly which constructions wer are testing, under which constraints. In the next chapter, we present the results of this investigation.

[12]The idea of cluster usage is that each job has a requested number of CPUs exclusively for itself, in which case the used CPUs would not be affected of varying loads. However, jobs are not prevented from using more than the requested number of CPUs on the same node, what means that there is no guarantee that the used CPUs are not also used by other jobs, what may put them under varying load.

Chapter 5

Results and Discussion

Contents

In this chapter, we present and discuss the results that we obtained from the empirical performance investigation of the Fribourg construction. Our approach is to statistically evaluate and present the data in different ways that should make the relevant aspects of the results apparent.

As mentioned, our main performance measure is the number of states of the complements that are produced by the tested constructions. However, we also consider the number of timeouts and memory excesses (based on the time and memory limits presented in Section **??**) and the execution times of the different constructions. Note that while the results of the first measure, complement size for a given automaton for a given construction, are definitive and repeatable, the results of the second measure are dependent on the implementation and the execution environment. That is, running the same tests with different implementatios in a different execution environment is likely to produce different results. This is why we take these runtime-dependent measure with care, and focus on the more robust measure of complement sizes.

Complement size means number of states of the complement.

Section 5.1 presents the results of the internal tests, and Section 5.2 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 5.3 we summarise and discuss the most important results and insights gained from the study. Finally, in Section 5.3.2, we identify the limitations of our study.

5.1 Results of the Internal Tests

The internal tests consists of the testing of different versions of the Fribourg construction with both, the GOAL test set and the Michel test set (see Section 4.3.1). The tested construction versions differ for the two test sets. Below, we present the results of these two tests sets separately.

5.1.1 GOAL Test Set

For the GOAL test set, the tested versions of the Fribourg construction are (see Section 4.3.1):

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

Below we present the results for all these eight versions from different perspectives. We first analyse the overall results, that is the results for the entire test set. Then, we analyse the results for each of the 110 transition density/acceptance density classes in which the GOAL test set is divided into. Finally, we generalise these per-class results by trying to establish a categorisation of the GOAL test set automata into *easy*, *medium*, and *hard* automata.

Overall Results

First of all, we have to see how many timeouts and memory excesses there are in order to establish the effective samples, that is, the set of automata that have been successfully complemented by *all* of the tested constructions.All the remaining analyses in this section are based on this set of effective samples. Table 5.1 shows the number of timeouts and memory excesses for each of the tested Fribourg construction versions.

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

As can be seen in Table 5.1, there is no memory excess for any of the construtions. That is, none of the constructions required more than 1 GB Java heap for any of the 11,000 complementation tasks in the GOAL test set. However, as can be seen, there are some timeouts. For example, Fribourg, has 48 timeouts, which means that 48 of its complementation tasks were aborted, because they exceeded the time limit of 600 CPU time seconds per automaton.

Based on these results, we have to determine the effective samples, that is, the set of automata that have been successfully processed by *all* the eight constructions. The analysis yields a set of effective samples of 10,939 automata. This means that 61 automata (0.55%) are excluded because their complementation failed for at least one of the constructions. All the following analyses will be based on these 10,939 automata of the effective samples.

Having determined the effective samples, we can turn to the analysis of our main performance measure, namely the sizes of the produced complements. To get a first impression, in Figure 5.1 we plot all the measured complement sizes in a stripchart. The complement sizes of all the tested constructions are represented as a strip containing a circle for each of the 10,939 automata of the effective samples. The number of states of the corresponding complements are indicated on the x-axis.

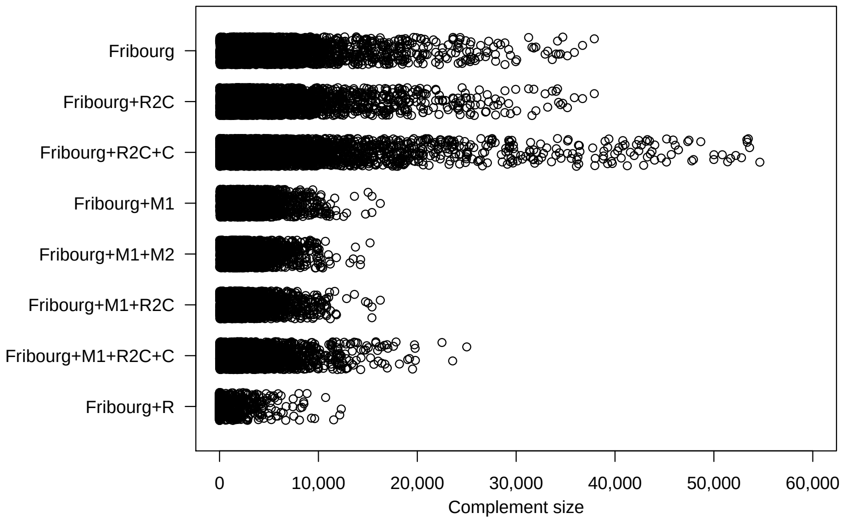


Figure 5.1: Complement sizes of the 10,939 effective samples for each tested version of the Fribourg construction on the GOAL test set.

The first thing to note is that the distribution of complement sizes is right-skewed (also known as positive-skewed). This means that there are many small complements, and then gradually fewer larger complements. This results in a long tail toward the right on the x-axis. Generally, in right-skewed distributions, the mean is larger than the median, because the mean is increased by the few number of larger or extremely large values. We will see that this is the case for our complement size distribution further below.

The stripchart allows to intuitively compare the density of large complements that are produced by the different versions. 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 4.2.1, only 9% of the automata of the GOAL test set are alreacy complete. Thus 91% of the automata are affected by the C option.

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.

Fribourg+M1+R2C+C again has a longer tail than Fribourg+M1+R2C. The reason for this is most likely the effect of the C option that we already observed for Fribourg+R2C and Fribourg+R2C+C.

Finally, Fribourg+R has the shortest tail of all versions. Fribourg+R is a special case, because it is the same construction as Fribourg, but with the difference that at the end of the constrcution, all unreachable and dead states are removed. Comparing the strips of Fribourg and Fribourg+R thus gives an idea of how many unreachable and dead states the Fribourg construction produces for the tested automata.

The stripchart in Figure 5.1 gives a good first impression about the distribution of produced complement sizes of the different versions. However, for further insights, we need to statistically evaluate these distributions. Table 5.2 shows the mean complement size for each version, along with the classical five-number summary consisting of the 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 5.2: Statistics of the complement sizes of the 10,939 effective samples for each tested version of the Fribourg construction on the GOAL test set.

As can be seen in Table 5.2, the mean is for all constructions significantly higher than the median. This is typical for right-skewed distributions. Generally, the median is said to be more robust than the mean, because it is not affected by extreme values. For example, for the distribution $(1, 2, 3, 4, 5)$ the mean and the median are both 3. If this distribution happens to contain an extreme value, for example, $(1, 2, 3, 4, 1000)$, then the mean is increased to 202, while the median is still 3. Thus, the median better represents the “typical” values of a distribution, whereas the mean better reflects the effect of extreme values. Both aspects may be important for the analysis of the data at hand. For our own analyses, we will mainly focus on the median, because we are more interested in the “typical” behaviour of the tested constructions. However, we will always put the median in relation with the mean and the the other statistical indicators in Table 5.2.

The medians in Table 5.2 include provide interesting insights. Regarding Fribourg and Fribourg+R2C, there is a decrease of the median complement size from 761 to 689. This is expectable because the R2C removes certain states from the complements of the complete input automata (although the complete input automata make up only 9% of all automata).

Regarding Fribourg+R2C+C the median complement size drops significantly to 451. This is surprising insofar as Fribourg+R2C+C has the highest mean of all versions, and in the stripchart in Figure 5.1, it has the longest tail. Thus at first glance, Fribourg+R2C+C might look like the worst version. However, regarding the median, and also the 25th percentile, Fribourg+R2C+C belongs actually to the best versions. On the other hand, regarding the 75th percentile and the maximum, Fribourg+R2C+C has the highest of all values. A possible explanation for this is that the R2C+C option combination makes small complements smaller and large complements larger. We will further elaborate on this phenomenon when we analyse the complement sizes for each of the transition density/acceptance density classes below.

Regarding Fribourg+M1 and Fribourg+M1+M2, the median of the former is lower than the median of the latter (482 against 496). The same applies to the 25th and 75th percentile. This means that the application of the M1 optimisation alone results in smaller complements than the application of the M1 and M2 optimsiations together. At first sight this is surprising, because Fribourg+M1+M2 has a lower worst-case complexity than Fribourg+M1 (see Section 3.2). However, as we already hinted at in Section 1.2.2 in the introductory chapter, a lower worst-case state complexity does not mean smaller complements in concrete cases. With the results of Fribourg+M1 and Fribourg+M1+M2 we have an empirical evidence for this claim. Note that the mean of Fribourg+M1+M2 is slightly lower than the mean of Fribourg+M1, which might result from the cases where Fribourg+M1 produces larger complements than Fribourg+M1+M2. However, regarding the median, 25t percentile, and 75th percentile values, we still see Fribourg+M1 as the more performant version for the GOAL test set, as we have indicated in Section 4.3.1.

Fribourg+M1+R2C decreases the median of Fribourg+M1 from 482 to 447. Also the 25th and 75th percentile are decreased. This results from the removal of states that the R2C option allows for the 9% of complete automata in the GOAL test set.

Comparing Fribourg+M1+R2C with Fribourg+M1+R2C+C, we observe a similar phenonmenon as for Fribourg+R2C and Fribourg+R2C+C. The median and 25th percentile drop significantly, whereas the mean, 75th percentile, and maximum value increase. Again it seems that making the 91% of incomplete automata preliminiarily complete (by the C option) in order to apply the R2C optimisation to all of them, has a very positive effect on some automata, but a negative effect on others.

Finally, the results of Fribourg+R make apparent that this version is a special case. The median is 1, and further analysis reveals that all complements up to the 61st percentile have a size of 1. This number coincides well with the 61.8% of automata in the GOAL test set that are universal (see Section **??**). These universal automata indeed have a minimum complement consisting of a single non-accepting state (an empty automaton). It seems that whatever complement the Fribourg construction produces for these automata, the R option prunes all but one state. Even though universal automata are special cases, the results of Fribourg+R still give an idea about the number of unreachable and dead states that are produced by the Fribourg construction.

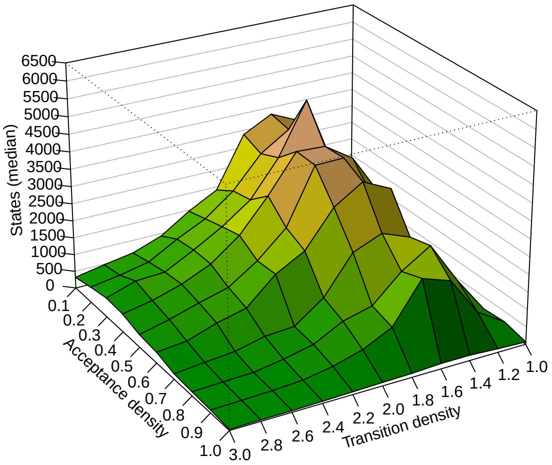
Results by Transition Density/Acceptance Density Class

Up to now, we looked at the results that are aggregated over the entire test set. This mixes together all the automata of the 110 transition density/acceptance density classes. In this part, we analyse the results for each of these 110 classes separately. The goal is to reveal how the constructions perform on automata with very different characteristics, and to reveal the relative differences between them.

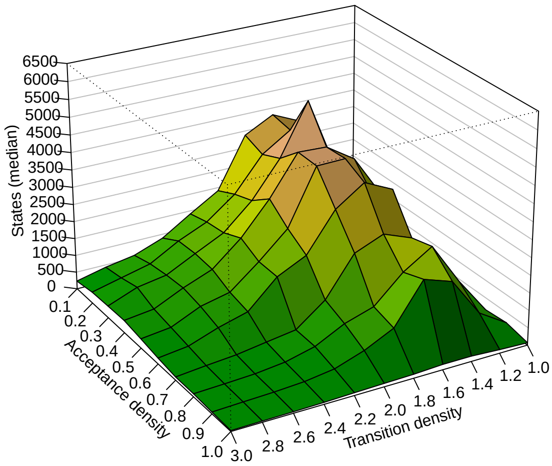
Such class-based analysis means that there is not just one value per statistical indicator for each construction, but 110 values, one for each class. For example, for every tested construction there are 110 means, 110 medians, and so on. In order to be able to present this amount of data, we restrict ourselves to the median values of each class. As already mentioned, our main focus is on the median, because it best reflects the "typical" results. Consequently, all the reported values in the following are median values.

The per-class median complement sizes result in a similar type of data that we encountered in the analysis of the complete, universal, and empty automata in the GOAL test set in Section 4.2.1. There, we presented this data in two forms, as matrices and as perspective plots. The advantage of matrices is that they show the explicit values, the advantage of perspective plots is that they better show the relative differences and the overall pattern. For the sake of brevity, we use only perspective plots in this chapter. However, we provide the matrices corresponding to all the perspective plots of this chapter in Appendix B.

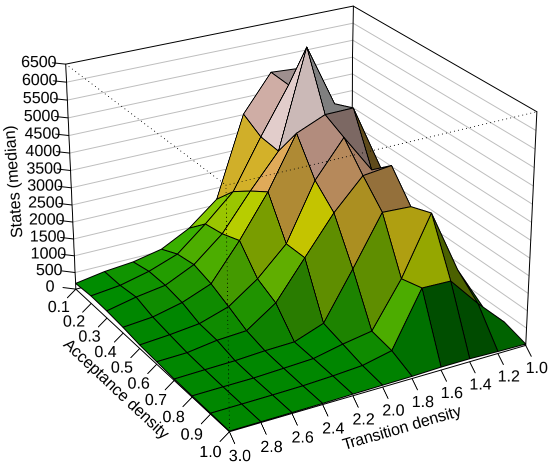
Figures 5.2 and 5.3 show the perspective plots with the per-class median complement sizes for the eight tested Fribourg construction version. Note that looking at these perspective plots corresponds to looking at the corresponding matrix from the bottom right corner. We keep this orientation for all the perspective plots in this chapter. Note that this orientiation is different from the orientatio used for the perspective plots presenting the number of complete, universal, and empty automata of the GOAL test set in Section 4.2.1. The colours of the rectangles (called facets) in the perspective plots depend on the "height" of the plot at their four corners, and are chosen to draw an analogy with mountains.



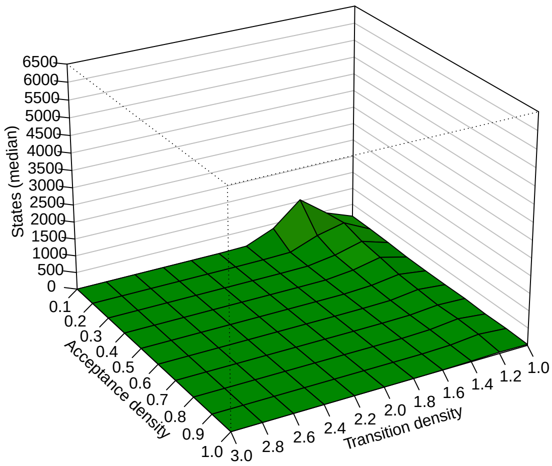
(a) Fribourg



(b) Fribourg+R2C



(c) Fribourg+R2C+C



(d) Fribourg+R

Figure 5.2: Median complement sizes of the 10,939 effective samples for each of the 110 transition density/acceptance density classes of the GOAL test set.

As can be seen in the perspective plots of Figures 5.2 and 5.3, there are large differences of the median complement sizes between the 110 transitio density/acceptance density classes. All constructions exhibit a mountain (in some it is only a hill) 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 acceptance densities. The summit of the ridge is roughly at a transition density of 1.6. In the direction of lower acceptance densities, the mountain keeps its high altitude. In the direction of higher acceptance densities, on the other hand, the mountain gradually flattens down until nearly ground level at an acceptance density of 1.0.

Comparing the height of the mountains in the perspective plots with the overall median complement sizes in Table 5.2 might be surprising at first. The mountains have a height of roughly 2,000–5,000 states (except Fribourg+R), which corresponds to the median complement sizes of the corresponding classes. On the other hand, the overall median complement size of all construction is 761 states or less. However, a closer look at the perspective plots (and the corresponding matrices in Appendix B) reveals that around half of the classes have a rather low median complement size ($< 1,000$) what justifies the low overall median complement sizes in Table ??. At the same time, this highlights the importance of a per-class analysis of the results on the GOAL test set. It reveals much more accurately what to expect of the complementation of a specific automaton that resembles one of the classes of the GOAL test set.

Note that in the following, we will refer to specific points in the perspective plots in the form of coordinats (t, a), where t stands for the transition density, and a for the acceptance density. For example, $(1.6, 0.3)$ denotes the point at a transition density of 1.6 and an acceptance density of 0.3.

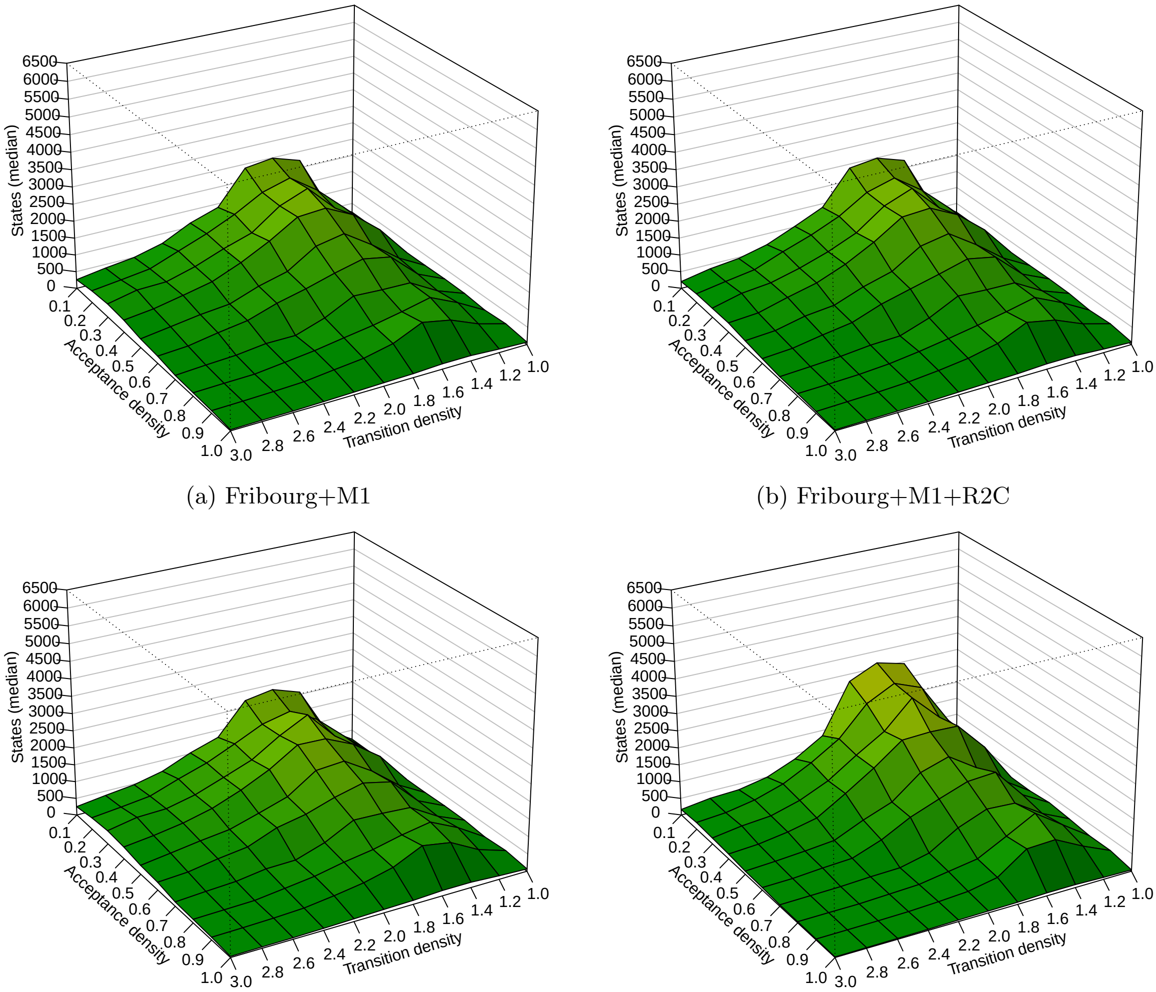
Let us now turn to the relative differences between the tested constructions. The plots of Fribourg and Fribourg+R2C in Figure 5.2 (a) and (b) are rather similar. For both, the ridge has between 3,500 and 4,000 states with a peak of around 4,900 states at $(1.6, 0.3)$. In fact Fribourg+R2C only improves the performance on the complete input automata, compared to Fribourg. As we have analysed in Section 4.2.1, the density of complete automata is highest in classes with high transition densities. By looking closely at the two plots, the values for the high transition density classes are indeed slightly lower for Fribourg+R2C than for Fribourg. Obviously, for classes that do not contain any complete automata, the values of the two constructions are identical.

Fribourg+R2C+C in Figure 5.2 (c) has the highest mountain of all constructions. This means basically that with Fribourg+R2C+C the "mountain classes" (the classes that result in large complements), result in even larger complements than for the other constructions. This might be surprising again, because as can be seen in Table 5.2, Fribourg+R2C+C has one of the lowest *overall* median complement sizes. However, by comparing the classes with low median complement sizes of Fribourg+R2C+C with the other constructions, we see that they are indeed often even lower. This means that while the "mountain classes" have exceptionally high median complement sizes, the "flatland classes" have exceptionally low ones. This supports our previous proposition that the R2C+C combination, which includes the adding of a sink state to incomplete automata (which constitute 91% of the GOAL test set), makes easy complementation tasks easier and hard ones harder.

Fribourg+R in Figure 5.2 (d) has, as expected, extremely low values for all the classes. In particular, 68 of the 110 classes have a median complement size of 1 (see corresponding matrix in Figure B.1 (d) in Appendix B). We already hinted previously at the relation of a median complement size of 1 and the universal automata in the GOAL test set. The per-class results of Fribourg+R reveal that all the classes with a median complement size of 1 contain 50 or more universal automata (compare matrices in Figure B.1 (d) in Appendix B and Figure 4.2 (b)). If all these univeral automata are reduced by the R option to a single-state automaton, then it explains the observed median complement size of 1 for these classes (remember that there are 100 automata in each class). However, also the classes which contain less than 50 universal automata have significantly lower complement median sizes than in the Fribourg construction without the R option. This indicates that the Fribourg construction also generates numerous unreachable and dead states for non-universal automata.

Figure 5.3 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



(a) Fribourg+M1

(b) Fribourg+M1+R2C

(c) Fribourg+M1+M2

(d) Fribourg+M1+R2C+C

Figure 5.3: Median complement sizes of the 10,939 effective samples for each of the 110 transition density/acceptance density classes of the GOAL test set.

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 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 (≈ 135 hours) to 60,919 seconds (≈ 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 **??**.

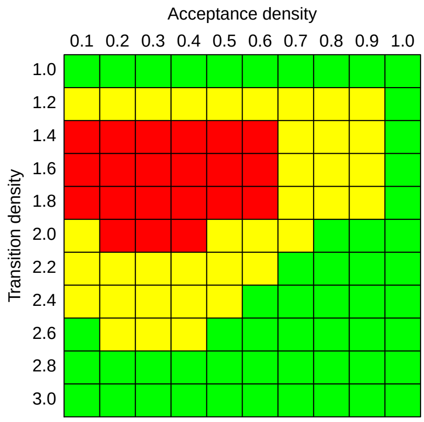


Figure 5.4: Difficulty categories *easy* (green), *medium* (yellow), and *hard* (red) of the 110 transiton density/acceptance density classes of the GOAL test set.

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.

5.1.2 Michel Test Set

For the Michel test set, the tested versions are (again, see Section 4.3.1):

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

Our second test set consists of the four Michel automata that are listed in Figure **??** in Section 4.2.2. 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 5.3.

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 5.3: 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 5.3, 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 5.3 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 5.1.1 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 5.1.1.

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 [29][51], 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 5.4 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 5.4: 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 5.5, 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.

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

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

Michel 5 to 8, between 18 months and 1,142 years, is definitely too long, even for a master's thesis.

5.2 Results of the 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.

5.2.1 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 5.6 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 5.6: Number of timeouts and memory excesses for the GOAL test set.

With Rank

Most salient in Table 5.6 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 5.5 we present the complement sizes of these 7,204 effective samples as a stripchart.

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 5.6 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 5.7.

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

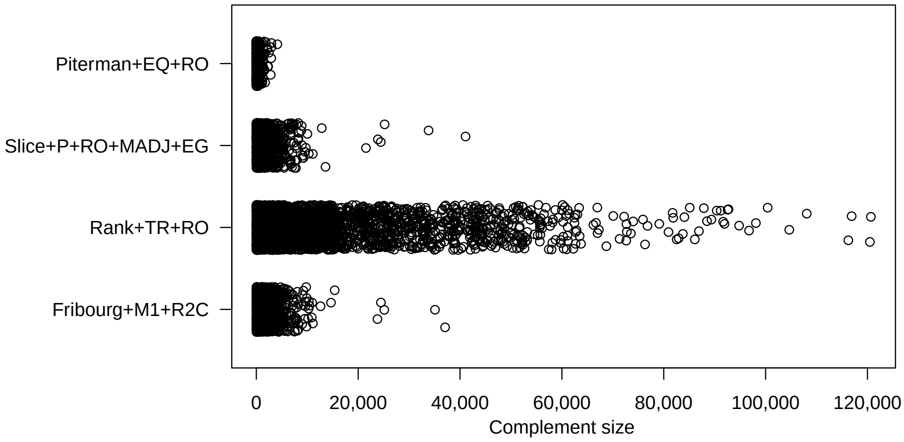


Figure 5.5: Complement sizes of the 7,204 effective samples.

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 5.7: Statistics of complement sizes of the 7,204 effective samples

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 5.6 we display the complement sizes of these 10,998 effective samples as a stripchart.

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 5.8. 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

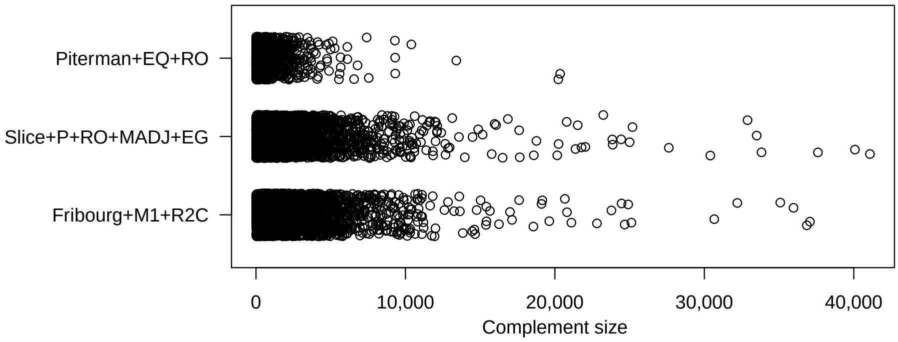


Figure 5.6: Complement sizes of the 10,998 effective samples.

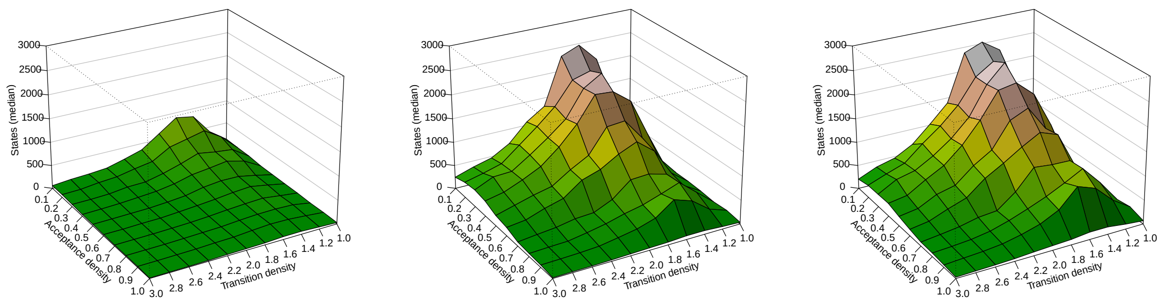
like Piterman would produce complements that are throughout five times smaller than the complements of Fribourg and Slice.

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 5.8: 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 5.7 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.



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

Figure 5.7: Median complement sizes (10,998 samples)

Note that the plot of Fribourg+M1+R2C in Figure 5.7 (c) is the same as the one in Figure 5.3 (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.

5.2.2 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 5.8. 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.

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 5.8: Complement sizes of the first four Michel automata.

Considering the results of the GOAL test set, the results in Table 5.8 are surprising. Rank is the most efficient construction. It produces the smallest complements for all Michel automata, and with $(0.91n)^n$ 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 5.9 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 5.9: Execution times for the first four Michel automata.

Most interesting in Table 5.9 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 5.9 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.

5.3 Discussion

5.3.1 Sumary of the Results

5.3.2 Limitations of the Study

Chapter 6

Conclusions

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	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

(d) Fribourg+R

	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

(e) Fribourg+M1

	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	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

(g) 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	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

(h) Fribourg+M1+R2C+C

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] A. Aziz, F. Balarin, S.-T. Cheng, et al. HSIS: A BDD-Based Environment for Formal Verification. In *Design Automation, 1994. 31st Conference on*. pp. 454–459. June 1994.

[4] C. Baier, J.-P. Katoen, et al. *Principles of Model Checking*. MIT press Cambridge. 2008.

[5] M. Ben-Ari. *Mathematical Logic for Computer Science*. Springer. 3rd edition ed.. 2012.

[6] S. Breuers. Optimization and experimental analysis of the algebraic complementation construction for Büchi automata. Bachelor's thesis, RWTH Aachen. October 2010.

[7] 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.

[8] 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.

[9] A. Cimatti, E. Clarke, E. Giunchiglia, et al. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In E. Brinksma, K. Larsen, eds., *Computer Aided Verification*. vol. 2404 of *Lecture Notes in Computer Science*. pp. 359–364. Springer Berlin Heidelberg. 2002.

[10] A. Cimatti, E. Clarke, F. Giunchiglia, et al. NuSMV: A New Symbolic Model Verifier. In N. Halbwachs, D. Peled, eds., *Computer Aided Verification*. vol. 1633 of *Lecture Notes in Computer Science*. pp. 495–499. Springer Berlin Heidelberg. 1999.

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

[12] E. Friedgut, O. Kupferman, M. Vardi. Büchi Complementation Made Tighter. In F. Wang, ed., *Automated Technology for Verification and Analysis*. vol. 3299 of *Lecture Notes in Computer Science*. pp. 64–78. Springer Berlin Heidelberg. 2004.

[13] E. Friedgut, O. Kupferman, M. Y. Vardi. Büchi Complementation Made Tighter. *International Journal of Foundations of Computer Science*. 17(04):pp. 851–867. 2006.

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

[15] R. L. Graham, B. L. Rothschild, J. H. Spencer. *Ramsey theory*. vol. 20. John Wiley & Sons. 1990.

[16] S. Gurumurthy, O. Kupferman, F. Somenzi, et al. On Complementing Nondeterministic Büchi Automata. In D. Geist, E. Tronci, eds., *Correct Hardware Design and Verification Methods*. vol. 2860 of *Lecture Notes in Computer Science*. pp. 96–110. Springer Berlin Heidelberg. 2003.

[17] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*. 23(5):pp. 279–295. 1997.

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

[19] M. Huth, M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press. 2004.

[20] 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.

[21] H. Karmarkar, S. Chakraborty. On Minimal Odd Rankings for Büchi Complementation. In Z. Liu, A. Ravn, eds., *Automated Technology for Verification and Analysis.* vol. 5799 of *Lecture Notes in Computer Science.* pp. 228–243. Springer Berlin Heidelberg. 2009.

[22] N. Klarlund. Progress measures for complementation of omega-automata with applications to temporal logic. In *Foundations of Computer Science, 1991. Proceedings., 32nd Annual Symposium on.* pp. 358–367. Oct 1991.

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

[24] O. Kupferman, M. Y. Vardi. Weak Alternating Automata Are Not that Weak. In *Proceedings of the 5th Israeli Symposium on Theory of Computing and Systems.* pp. 147–158. IEEE Computer Society Press. 1997.

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

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

[27] C. Löding. *Methods for the Transformation of ω-Automata: Complexity and Connection to Second Order Logic.* Master's thesis. Christian-Albrechts-University of Kiel. 1998.

[28] C. Löding. Optimal Bounds for Transformations of ω-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.

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

[30] M. Mohri. A Disambiguation Algorithm for Finite Automata and Functional Transducers. In N. Moreira, R. Reis, eds., *Implementation and Application of Automata.* vol. 7381 of *Lecture Notes in Computer Science.* pp. 265–277. Springer Berlin Heidelberg. 2012.

[31] 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.

[32] 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.

[33] D. E. Muller, A. Saoudi, P. E. Schupp. Alternating automata, the weak monadic theory of the tree, and its complexity. In L. Kott, ed., *Automata, Languages and Programming.* vol. 226 of *Lecture Notes in Computer Science.* pp. 275–283. Springer Berlin Heidelberg. 1986.

[34] 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.

[35] 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.

[36] N. Piterman. From Nondeterministic Buchi and Streett Automata to Deterministic Parity Automata. In *Logic in Computer Science, 2006 21st Annual IEEE Symposium on.* pp. 255–264. 2006.

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

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

[39] 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.

[40] F. P. Ramsey. On a Problem of Formal Logic. *Proceedings of the London Mathematical Society*. s2-30(1):pp. 264–286. 1930.

[41] 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.

[42] S. Safra. On the Complexity of Omega-Automata. *Journal of Computer and System Science*. 1988.

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

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

[45] A. Sistla, M. Vardi, P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. In W. Brauer, ed., *Automata, Languages and Programming*. vol. 194 of *Lecture Notes in Computer Science*. pp. 465–474. Springer Berlin Heidelberg. 1985.

[46] 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.

[47] F. Somenzi, R. Bloem. Efficient Büchi automata from LTL formulae. In *Computer Aided Verification*. pp. 248–263. Springer. 2000.

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

[49] S. TaŞiran, R. Hojati, R. Brayton. Language Containment of Non-Deterministic ω-Automata. In P. Camurati, H. Eveking, eds., *Correct Hardware Design and Verification Methods*. vol. 987 of *Lecture Notes in Computer Science*. pp. 261–277. Springer Berlin Heidelberg. 1995.

[50] 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.

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

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] M. Vardi. Automata-Theoretic Model Checking Revisited. In B. Cook, A. Podelski, eds., *Verification, Model Checking, and Abstract Interpretation*. vol. 4349 of *Lecture Notes in Computer Science*. pp. 137–150. Springer Berlin Heidelberg. 2007.

[62] M. Vardi. The Büchi Complementation Saga. In W. Thomas, P. Weil, eds., *STACS 2007*. vol. 4393 of *Lecture Notes in Computer Science*. pp. 12–22. Springer Berlin Heidelberg. 2007.

[63] M. Y. Vardi. Buchi Complementation: A Forty-Year Saga. In *5th symposium on Atomic Level Characterizations (ALC'05)*. 2005.

[64] 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.

[65] T. Wilke. ω-Automata. In J.-E. Pin, ed., *Handbook of Automata Theory*. European Mathematical Society. To appear, 2015.

[66] Q. Yan. Lower Bounds for Complementation of ω-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.

[67] Q. Yan. Lower Bounds for Complementation of ω-Automata Via the Full Automata Technique. *CoRR*. abs/0802.1226. 2008.