Performance Investigations of a Subset-Tuple Büchi Complementation Construction

Daniel Weibel

May 16, 2015

Abstract

This will be the abstract.

Acknowledgements

First of all, I would like to thank my supervisors Prof. Dr. Ulrich Ultes-Nitsche and Joel Allred to bring me to the topic of this thesis in the first place, and for the many enlightening discussions and explanations. A very special thank goes to Ming-Hsien Tsai, the main author of the GOAL tool, from the National Taiwan University. Without his very prompt and patient responses to my many questions regarding implementation details of GOAL, the plugin would not have been possible in its current form. I also would like to thank Michael Rolli and Nico Färber from the Grid Admin Team of the University of Bern for their very friendly and helpful responses to my questions about the Linux cluster where the experiments of this thesis have been executed.

Contents

Chapter 1

Introduction

At the beginning of the 1960s, a Swiss logician named Julius Richard Büchi at Michigan University was looking for a way to prove the decidability of the satisfiability of monadic second order logic with one successor (S1S). Büchi applied a trick that truly founded a new paradigm in the application of logic to theoretical computer science. He thought of interpretations of a S1S formula as infinitly long words of a formal language and designed a type of finite state automaton that accepts such a word if and only if the interpretation it represents satisfies the formula. After proving that every S1S formula can be translated to such an automaton and vice versa (Büchi's Theorem), the satisfiabilty problem of an S1S formula could be reduced to testing the non-emptiness of the corresponding automaton.

This special type of finite state automaton was later called Büchi automaton.

A Büchi complementation construction takes as input a Büchi automaton A and produces as output another Büchi automaton B which accepts the complement language of the input automaton A. Complement language denotes the "contrary" language, that is, B must *accept* (over a given alphabet) every word that A *does not* accept, and must in turn *not accept* every word that A *accepts*.

Büchi automata are finite automata (that is, having a finite number of states) which operate on infinite words (that is, words that "never end"). Operating on infinite words, they belong thus to the category ω-automata. An important application of Büchi automata is in model checking which is a formal system verification technique. There, they are used to represent both, the description of the system to be checked for the presence of a correctness property, and (the negation of) this correctness property itself.

In one approach to model checking, the correctness property is directly specified as a Büchi automaton One approach to model checking requires that the Büchi automaton representing the correctness property is complemented. It is here that the problem of Büchi complementation has one of its practical applications.

The complementation of non-deterministic Büchi automata is hard. It has been proven to have an exponential lower bound in the number of generated states [cite]. That is, the number of states of the output automaton is, in the worst case, an exponential function of the number of states of the input automaton. However, since the introduction of Büchi automata in the 1960's, significant process in reducing the complexity (in other words, the degree of exponentiality) of the Büchi complementation problem has been made. Some numbers [list complexities of the different constructions].

Chapter 2

The Büchi Complementation Problem

Contents

2.1 Preliminaries

2.1.1 Büchi Automata

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

he had proved the decidability of the monadic-second order theory of the natural numbers with successor func- tion by translating formulas into finite automata [12] (p. 1)

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

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

Definitions

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

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

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

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

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

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

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

Continued/discontinued runs

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

Expressiveness

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

However, this equivalence does not hold for DBW (Büchi showed it too). There are ω-regular languages that cannot be recognised by any DBW. A typical example is the language $(0+1)^*1^\omega$. This is the language

of all infinite words of 0 and 1 with only finitely many 0. It can be shown that this language can be recognised by a NBW (it is thus a ω-regular language) but not by a DBW [**?**][**?**]. The class of languages recognised by DBW is thus a strict subset of ω-regular languages recognised by NBW. We say that DBW are less expressive than NBW.

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

2.1.2 Other ω-Automata

After the introduction of Büchi automata in 1962, several other types of ω-automata have been proposed. The best-known ones are by Muller (Muller automata, 1963) [**?**], Rabin (Rabin automata, 1969) [**?**], Streett (Streett automata, 1982) [**?**], and Mostowski (parity automata, 1985) [**?**].

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

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

In the Muller acceptance condition, the set of infinitely occuring states of a run ($\inf(\rho)$) must match a predefined set of states. The Rabin and Streett conditions use pairs of state sets, so-called accepting pairs. The Rabin and Streett conditions are the negations of each other. This allows for easy complementation of deterministic Rabin and Streett automata [**?**], which will be used for certain Büchi complementation construction, as we will see in Section2.4. The parity condition assigns a number (color) to each state and accepts a run if the smallest-numbered of the infinitely often occuring states has an even number. For all of these automata there exist non-deterministic and deterministic versions, and we will refer to them as NMW, DMW (for non-deterministic and deterministic Muller automata), and so on.

In 1966, McNaughton made an important proposition, known as *McNaughton's Theorem* [**?**]. Another proof given in [**?**]. It states that the class of languages recognised by deterministic Muller automata are the ω-regular languages. This means that non-deterministic Büchi automata and deterministic Muller automata are equivalent, and consequently every NBW can be turned into a DMW. This result is the base for the determinisation-based Büchi complementation constructions, as we will see in Section 2.4.2.

It turned out that also all the other types of the just introduced ω-automata, non-deterministic and determinstic, are equivalent among each other [**?**][**?**][**?**][**?**][**?**]. This means that all the ω-automata mentioned in this thesis, with the exception of DBW, are equivalent and recognise the ω-regular languages. This is illustrated in Figure 2.1

2.1.3 Complementation of Büchi Automata

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

It is interesting to see that this closure does not hold for the specific case of DBW. That means that while for every DBW a complement Büchi automaton does indeed exist, following from the above closure property for Büchi automata in general, this automaton is not necessarily a DBW. The complement of a DBW may be, and often is, as we will see, a NBW. This result is proved in [**?**] (p. 15).

The problem of Büchi complementation consists now in finding a procedure (usually called a construction) that takes as input any Büchi automaton A and outputs another Büchi automaton B with

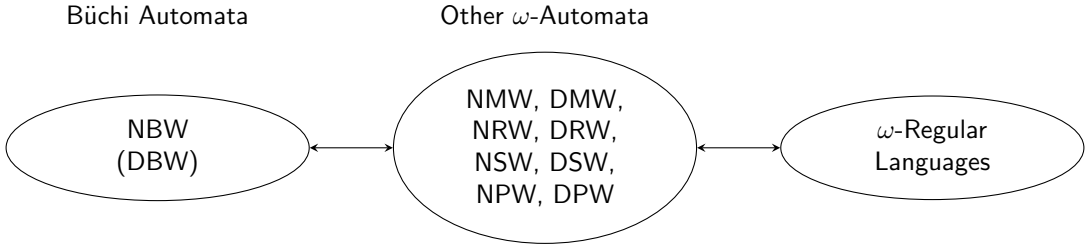
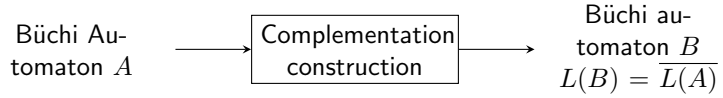


Figure 2.1: Non-deterministic Büchi automata (NBW) are expressively equivalent to Muller, Rabin, Streett, and parity automata (both deterministic and non-deterministic), and to the ω-regular languages. Deterministic Büchi automata (DBW) are less expressive than NBW.

$L(B) = \overline{L(A)}$, as shown below.



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

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

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

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

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

2.1.4 Complexity of Büchi Complementation

Constructions for complementing NBW turned out to be very complex. Especially the blow-up in number of states from the input automaton to the output automaton is significant. For example, the original complementation construction proposed by Büchi [**?**] involved a doubly exponential blow-up. That is, if the input automaton has n states, then for some constant c the output automaton has, in the worst case, c^{c^n} states [8]. If we set c to 2, then an input automaton with six states would result in a complement automaton with about 18 quintillion (18×10^{18}) states.

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

A question that arises is, how much this number can be brought down? Researchers have investigated this question by trying to establish so called lower bounds. A lower bound is a function for which it is

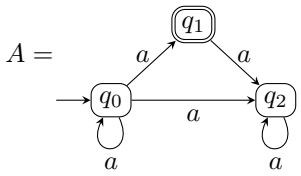


Figure 2.2: Example NBW A that will be used in different places throughout this thesis. The alphabet of A consists of the single symbol a, consequently, A can only process the single ω-word a^ω. This word is rejected by A, so the automaton is empty.

proven that no state blow-up of any construction can be less than it. The first lower bound for Büchi complementation has been established by Michel in 1988 at $n!$ [**?**]. This means that the state blow-up of any Büchi complementation construction can never be less than $n!$.

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

Michel's lower bound remained valid for almost two decades until in 2006 Yan showed a new lower bound of $(0.76n)^n$ [**?**]. This does not mean that Michel was wrong with his lower bound, but just too reserved. The best possible blow-up of a construction can now be only $(0.76n)^n$ and not $(0.36n)^n$ as believed before. In 2009, Schewe proposed a construction with a blow-up of exactly $(0.76n)^n$ (modulo a polynomial factor) [7]. He provided thus an upper bound that matches Yan's lower bound. The lower bound of $(0.76n)^n$ can thus not rise any further and seems to be definitve.

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

2.2 Run Analysis

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

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

2.2.1 From Run Trees to Split Trees

The one tree data-structure that truly represents *all* the runs of an automaton on a word are run trees. The other variants of trees that we present in this section are basically derivations of run trees that sacrifice information about individual runs, by merging or discarding some of them, at the benefit of becoming more concise. Figure 2.8 shows the first few levels of the run tree of the example automaton A from Figure 2.2 on the word a^ω.

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

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

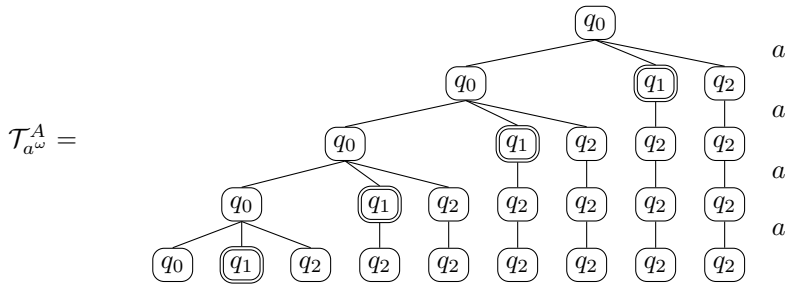


Figure 2.3: Automaton A and the first five levels of the run tree of the runs of A on the word a^ω.

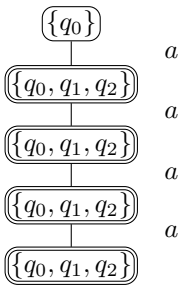


Figure 2.4: Automaton A and the first five levels of the run tree of the runs of A on the word a^ω.

definition of split trees, and at the same time shows why the subset construction fails for determinising NBW [1].

For determinising an automaton A, the subset construction in effect merges all the diverse runs of A on word w to one single run by merging all the states on a level of the corresponding run tree to one single state. This state will be a state of the output automaton B, and is labelled with the set of A-states it includes. Figure 2.4 shows this effect with our example automaton from Figure 2.2 and the word a^ω.

Clearly, this form of tree created by the subset construction is the most concise form a run tree can be brought to. However, almost all information about individual runs in A has been lost. All that can be said by looking at the structure in Figure 2.4 is that there must be at least one continued A-run on a^ω(all the other runs visiting the other A-states on each level, might be discontinued). But which states a possible continued run visits cannot be deduced.

This lack of identification A-runs is the reason why the subset construction fails for determinising Büchi automata. Note that a B-state of the subset construction is accepting if the set of A-states it represents contains at least one accepting A-state. For our example, this means that the state $\{q_0, q_1, q_2\}$ is accepting (this is also indicated in Figure 2.4). This state is visited infinitely often by the unified run on a^ω. Hence, the DBW B, resulting from applying the subset construction to the NBW A, accepts a^ωwhile A does not accept it.

By looking closer at the trees in Figure 2.4 and 2.8, the reason for this problem becomes apparent. If we look for example at the second level of the subset-construction tree we can deduce that there must be an A-run that visits the accepting A-state q_1. Let us call this run r_{q_1}. However, at the third level, we cannot say anything about r_{q_1} anymore, whether it visits one of the non-accepting states or again q_1 on the third level, or whether it even ended at the second level. In turn, what we know on the third level in our example is that there is again an A-run, r'_{q_1}, that visits q_1. However, whether r'_{q_1} is r_{q_1}, and in turn the future of r'_{q_1} cannot be deduced. In our example we end up with the situation that there are infintiely many visits to q_1 in the unifed B-run, but we don't know if the reaon for this are one or more A-runs that visit q_1 infinitely often, or infinitely man A-runs where each one visits q_1 only finitely often (the way it is in our example). In the first case, it would be correct ot accept the B-run, in the second case however it would be wrong as the input automaton A does not accept the word. The subset construction does not distinguish these two cases and hence the determinised automaton B may accept words that the input automaton A rejects. In general, the language of an output DBW of the subset construction is a superset of the language of the input NBW.

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

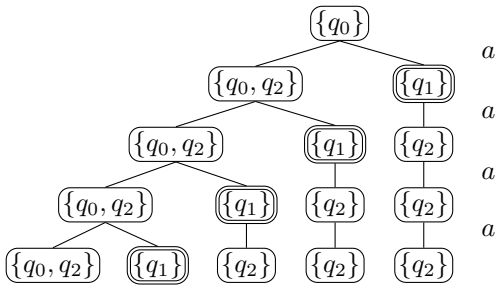


Figure 2.5: Automaton A and the first five levels of the split tree of the runs of A on the word a^ω.

This raises the question how the subset construction can be minimally modified such that the output automaton is equivalent to the input automaton. One solution is to not mix accepting and non-accepting A-states in the B-states. That is, instead of creating one B-state that contains all the A-states, as in the subset construction, one creates two B-states where one contains the accepting A-states and the other the non-accepting A-states. Such a construction has been formalised in [**?**]. The output automaton B is then not deterministic, but it is equivalent to A. The type of run analysis trees that correspond to this refined subset construction are split trees. Figure 2.9 shows the first five levels of the split tree of our example automaton A on the word a^ω.

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

Split trees can be seen as run trees where some of the branches are contracted to unified branches. In particular, a split tree unifies as many branches as possible, such that the resulting tree still correctly represents the Büchi acceptance of all the runs included in a unified branch. This can form the basis for constructions that transform an NBW to another equivalent NBW. Split trees are for example the basis for Muller-Schupp trees in Muller and Schupp's Büchi determinisation construction [4], cf. [1].

2.2.2 Reduced Split Trees

It turns out that split trees can be compacted even more. The resulting kind of tree is called reduced split tree. In a reduced split tree, each A-state occurs at most once on every level. Figure 2.6 shows the reduced split tree corresponding to the split tree in Figure 2.9. As can be seen, only one occurrence of each A-state on each level is kept, the other are discarded. To allow this, however, the order of the accepting and non-accepting siblings in the tree matters. Either the accepting child is always put to the right of the non-accepting child (as in our example in Figure 2.6, or vice versa. We call the former variant a right-to-left reduced split tree, and the latter a left-to-right reduced split tree. In this thesis, we will mainly adopt the right-to-left version.

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

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

A very important property of reduced split trees is that they have a fixed width. The width of a tree is the maximal number of vertices on a level. For reduced split trees, this is the number of states of the

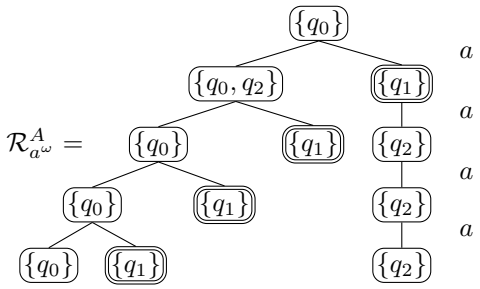


Figure 2.6: Automaton A and the first five levels of the reduced split tree of the runs of A on the word a^ω.

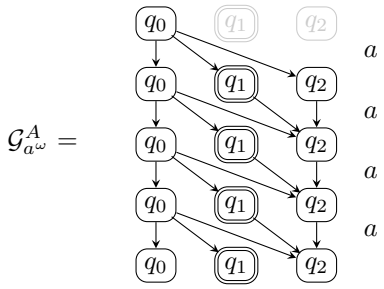


Figure 2.7: Automaton A and the first five levels of the run DAG of the runs of A on the word a^ω.

input automaton A. As we will see, the subset-tuple construction (like other slice-based constructions) uses levels of a reduced split tree as states othe output automaton, and the limited size of these levels ensures an upper bound on the number of states these constructions can create.

By deleting A-states from a level of a reduced split tree, we actually delete A-runs that reach the same A-state on the same substring of the input word. For example, in the split tree in Figure 2.9 we see that there are at least for A-runs on the string $aaaa$ from the inital state q_0 to q_2. The reduced split tre in Figure 2.6, however, contains only one run on $aaaa$ from q_0 to q_2, namely the rightmost branch of the tree. The information about all the other runs is lost. This single run that is kept is very special and, as we will see shorty, it represents the deleted runs. We will call this run the *greedy run*. The reason for calling it greedy is that it visits an accepting state earlier than any of the deleted runs. In a right-to-left reduced split tree, the greedy run is always the rightmost of the runs from the root to a certain A-state on a certain level. In left-to-right reduced split tree, the greedy run would in turn be the leftmost of these runs.

We mentioned that the greedy run somehow represents the deleted runs. More precisely, the relation is as follows and has been proved in [12]: if any of the deleted runs is a prefix of a run that is Büchi-accepted (that is, an infinite run visiting infinitely many accepting A-states), then the greedy run is so too. That means that if the greedy cannot be expanded to a Büchi-accepting run, then none of the deleted runs could be either. Conversely, if any of the deleted runs could become Büchi-accepting, then the greedy run can so too. So, the greedy run is sufficient to indicate the existence or non-existence of a Büchi-accepting run with this prefix, and it is safe to delete all the other runs.

2.2.3 Run DAGs

DAGs (directed acyclic graphs) are, after trees, the second form for analysing the runs of a non-deterministic automaton on a given word. A run DAG has the form of a matrix with one column for each A-state and a row for each position in the word. The directed edges go from the vertices on one row to the vertices on the next row (drawn below) according to the transitions in the automaton on the current input symbol. Figure 2.12 shows the first five rows of the run DAG of the example automaton in Figure 2.2 on the word a^ω.

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

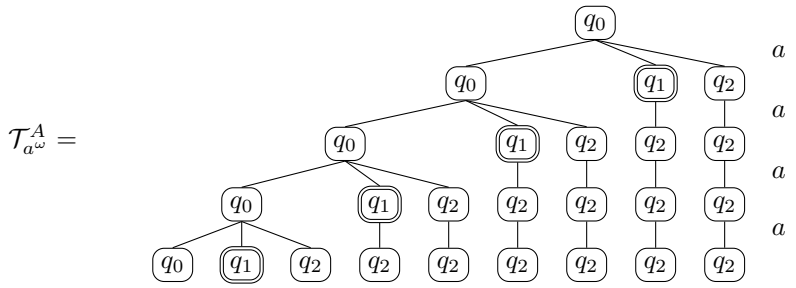


Figure 2.8: Automaton A and the first five levels of the run tree of the runs of A on the word a^ω.

2.3 Run Analysis

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

2.3.1 Run Trees

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

Figure 2.8 shows an example automaton A and the first five levels of the run tree for the word $w = a^\omega$ (infinite repetitions of the symbol a). Each branch from the root to one of the leaves represents a possible way for reading the first four positions of w. On the right, as a label for all the edges on the corresponding level, is the symbol that causes the depicted transitions.

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

We define by the width of a tree the maximum number of vertices occurring at any level [4]. Clearly, for ω-words the width of a run tree may become infinite, because there may be an infinite number of levels and each level may have more vertices than the previous one.

2.3.2 Failure of the Subset-Construction for Büchi Automata

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

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

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

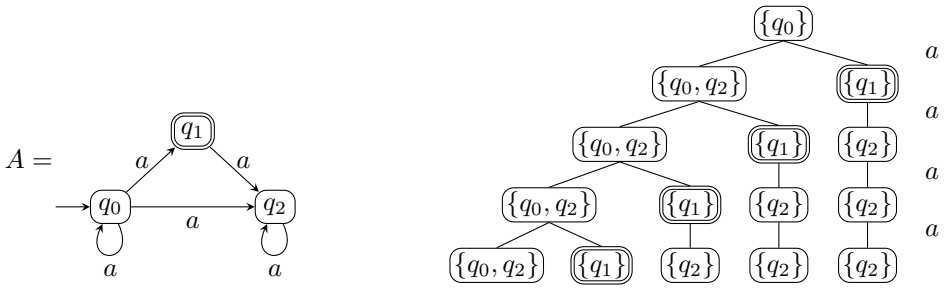


Figure 2.9: Automaton A and the first five levels of the split tree of the runs of A on the word a^ω.

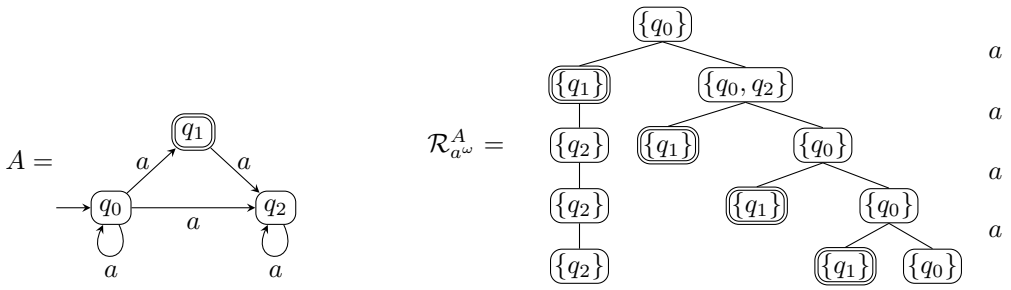


Figure 2.10: Automaton A and the first five levels of the left-to-right reduced split tree of the runs of A on the word a^ω.

2.3.3 Split Trees

Split trees can be seen as run trees where the accepting and non-accepting descendants of a node n are aggregated in two nodes. We will call the former the *accepting child* and the latter the *non-accepting child* of n. Thus in a split tree, every node has at most two descendants (if either the accepting or the non-accepting child is empty, it is not added to the tree), and the nodes represent sets of states rather than individual states. Figure 2.9 shows the first five levels of the split tree of automaton A on the word a^ω.

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

The notion of split trees (and reduced split trees, see next section) has been introduced by Kähler and Wilke in 2008 for their slice-based complementation construction [2], cf. [**?**]. However, the idea of separating accepting from non-accepting states has already been used earlier, for example in Muller and Schupp's determinisation-based complementation construction from 1995 [4]. Formal definitions os split trees can be found in [2][**?**].

2.3.4 Reduced Split Trees

The width of a split tree can still become infinitely large. A reduced split tree limits this width to a finite number with the restriction that on any level a given state may occur at most once. This is in effect the same as saying that if in a split tree there are multiple ways of going from the root to state q, then we keep only one of them.

2.3.5 Run DAGs

A run DAG (DAG stands for directed acyclic graph) can be seen as a graph in matrix form with one column for every state of A and one row for every position of word w. The edges are defined similarly than in run trees. Figure 2.12 shows the run DAG of automaton A on the word $w = a^\omega$.

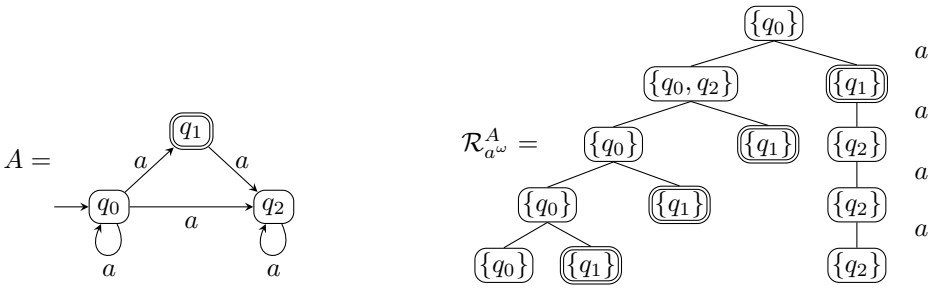


Figure 2.11: Automaton A and the first five levels of the left-to-right reduced split tree of the runs of A on the word a^ω.

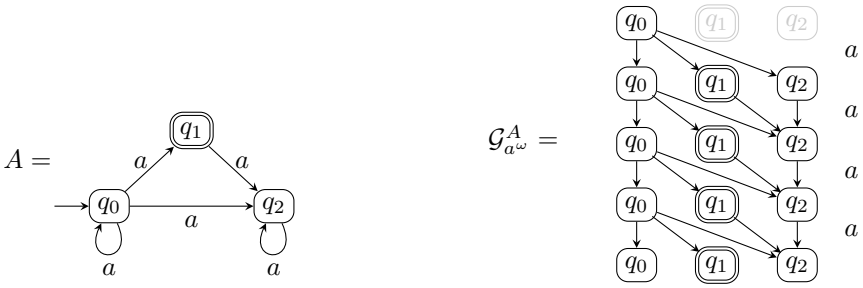


Figure 2.12: Automaton A and the first five levels of the run DAG of the runs of A on the word a^ω.

2.4 Review of Büchi Complementation Constructions

2.4.1 Ramsey-Based Approaches

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

2.4.2 Determinisation-Based Approaches

2.4.3 Rank-Based Approaches

2.4.4 Slice-Based Approaches

2.5 Empirical Performance Investigations

Chapter 3

The Fribourg Construction

Contents

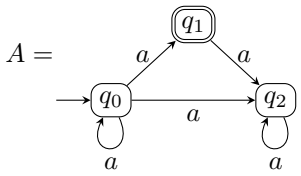


Figure 3.1: Example automaton A

The Fribourg construction draws from several ideas: the subset construction, run analysis based on reduced split trees, and Kurshan's construction [**?**] for complementing DBW. Following the classification we used in Section 2.4, it is a slice-based construction. Some of its formalisations are similar to the slice-based construction by Vardi and Wilke [12], however, the Fribourg construction has been developed independently. Furthermore, as we will see in Chapter **??**, the empirical performance of Vardi and Wilke's construction and the Fribourg construction differ considerably, in favour of the latter.

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

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

3.1 First Stage: Constructing the Upper Part

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

The structure of B'-states is determined by levels of corresponding reduced split trees. Vardi, Kähler, and Wilke refer to these levels as *slices* in their constructions [12, 2]. Hence the name slice-based approach. In the following, we will use the terms levels and slices interchangebly. A slice-based construction can work with either left-to-right or right-to-left reduced split trees. Vardi, Kähler, and Wilke use the left-to-right version in their above cited publications. In this thesis, in contrast, we will use right-to-left reduced split trees, which were also used from the beginning by the authors of the subset-tuple construction.

Figure 3.2 shows how levels of a right-to-left reduced split tree map to states of the subset-tuple construction. In essence, each node of a level is represented as a set in the state, and the order of the nodes determines the order of the sets in the tuple. [INFORMATION ABOUT ACC AND NON-ACC IS NEEDED IN THE LOWER PART BUT IMPLICIT IN THE STATES OF A]. To determine the successor of a state, say $(\{q_0, q_2\}, \{q_1\})$, one can regard this state as level of a reduced split tree, determine the next level and map this new level to a state. In the example of Figure 3.2, the successor of $(\{q_0, q_2\}, \{q_1\})$ is determined in this way to $(\{q_0\}, \{q_1\}, \{q_2\})$.

Apart from this special way of determining successor states, the construction of B'proceeds similarly as the subset construction. One small further difference is that if at the end of determining a successor for every state in B', the automaton is not complete, it must be made complete with an *accepting* sink

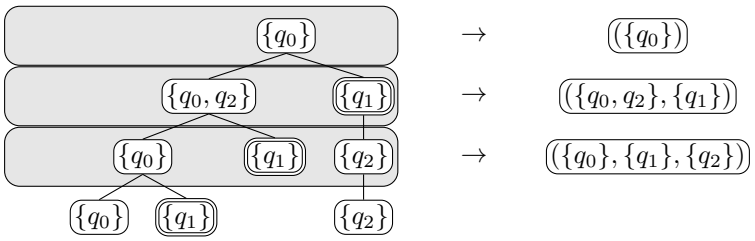


Figure 3.2: Mapping from levels of a reduced split tree, to states of the subset-tuple construction.

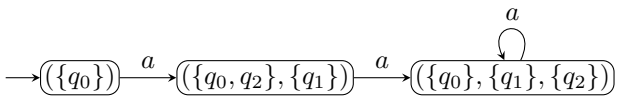


Figure 3.3: Upper part $B'$of example automaton A.

state. The steps for constructing $B'$from A can be summarised as follows.

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

For the example automaton A in Figure 3.1, we would start with $(\{q_0\})$, determine $(\{q_0, q_2\}, \{q_1\})$ as its a-successor, whose a-successor in turn we determine a $(\{q_0\}, \{q_1\}, \{q_2\})$. The a-successor of $(\{q_0\}, \{q_1\}, \{q_2\})$ is $(\{q_0\}, \{q_1\}, \{q_2\})$ again what results in a loop. Figure 3.3 shows the final upper part $B'$of A.

3.2 Second Stage: Adding the Lower Part

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

We divide our discussion of the lower part in two sections. In the following one (3.2.1), we explain the "mechanical" construction of the lower part, the steps that have to be done to arrive at the final complement automaton B. In the next section (3.3) we give the idea and intuition behind the construction and explain why it works.

3.2.1 Construction

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

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

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

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

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

Figure 3.4: Colour rules.

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

The predecessor set s_{pred} is the set of p_{pred} that in the corresponding reduced split tree is the parent node of the node corresponding to s. Figure 3.4 shows the values of $c(s)$ for all possible situations as two matrices. There is one matrix for the two cases of factor 1 above (p_{pred} has colour 2 or not) and the other two factors are laid out along the rows and columns of either matrix. Note that $c(s_{pred}) = -1$ is only present in the upper matrix, because in this case p_{pred} is a state of the upper part and cannot contain colour 2.

We will use the following notation to denote the colour of s: $\widehat{s}$ if $c(s) = -1$, s if $c(s) = 0$, $\overline{s}$ if $c(s) = 1$, and $\overline{\overline{s}}$ if $c(s) = 2$. Let us look now at a concrete example of this construction. We will add the lower part to the upper part B'in Figure 3.3, and thereby complete the complementation of the example automaton A in Figure 3.1.

First of all, we assign colour -1 all the sets of the states of B'. We might then start processing the state $(\widehat{\{q_0\}})$, let us call it p_{pred}. The resulting successor tuple, without the colours, of p_{pred} is, as in the upper part, $(\{q_0, q_2\}, \{q_1\})$. We now have to determine the colours of the sets $\{q_0, q_2\}$ and $\{q_1\}$. Since p_{pred} does not contain any 2-coloured sets, we need only to consult the upper matrix in Figure 3.4. For $\{q_1\}$, the predecessor set is $\widehat{\{q_1\}}$ with colour -1. Furthermore $\{q_1\}$ is accepting. So, the colour of $\{q_1\}$ is 2, because we end up in the first-row, second-column cell of the upper matrix ($M_1(1, 2)$). The other set, $\{q_0, q_2\}$, in turn is non-accepting, so its colour is 0 ($M_1(1, 1)$). The successor state of $(\widehat{\{q_0\}})$ is thus $(\{q_0, q_2\}, \overline{\overline{\{q_1\}}})$.

We can then continue the construction right with this new state $(\{q_0, q_2\}, \overline{\overline{\{q_1\}}})$, and call it p_{pred} in turn. The successing tuple without the colours of p_{pred} is $(\{q_0\}, \{q_1\}, \{q_2\})$. Since p_{pred} contains a set with colour 2, we have to consult the lower matrix of Figure 3.4 to determine the colours of $\{q_0\}$, $\{q_1\}$, and $\{q_2\}$. For $\{q_2\}$, we end up with colour 2 ($M_2(3, 1)$), because its predecessor set, which is $\overline{\overline{\{q_1\}}}$, has colour 2. $\{q_1\}$ gets colour 1 as it is accepting and its predecessor set, $\{q_0, q_2\}$, has colour 0 ($M_2(1, 2)$). $\{q_0\}$, which has the same predecessor set, gets colour 0, because it is non-accepting ($M_2(1, 1)$). The successor state of $(\{q_0, q_2\}, \overline{\overline{\{q_1\}}})$ is thus $(\{q_0\}, \overline{\{q_1\}}, \overline{\overline{\{q_2\}}})$.

The construction continues in this way until every state has been processed. The resulting automaton is shown in Figure 3.5. The last thing that has to be done is to make every state of the lower part that does not contain colour 2 accepting. In our example, this is only one state. The NBW B in Figure 3.5 is the complement of the NBW A in Figure 3.1, such that $L(B) = \overline{L(A)}$. This can be easily verified, since A is empty and B is universal (with regard to the single ω-word a^ω).

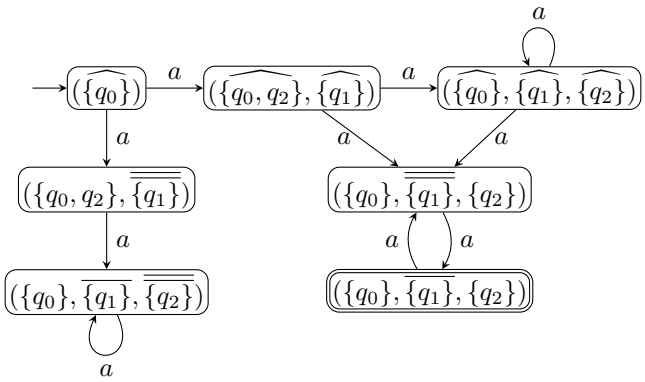


Figure 3.5: The final complement automaton B.

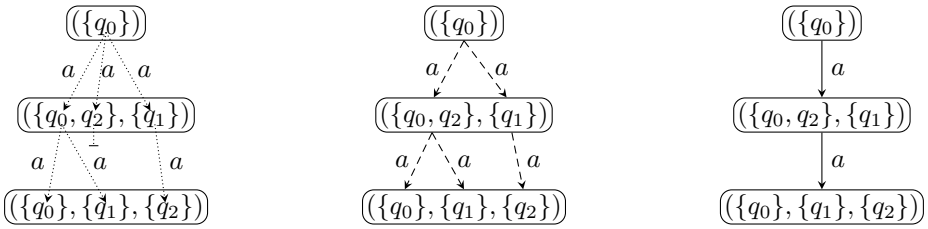


Figure 3.6: Different notions of runs.

3.2.2 Meaning and Function of the Colours

3.3 Intuition for Correctness

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

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

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

3.4 Optimisations

3.4.1 Removal of Non-Accepting States (R2C)

3.4.2 Merging of Adjacent Sets (M1)

3.4.3 Reduction of 2-Coloured Sets (M2)

Chapter 4

Performance Investigation of the Fribourg Construction

Contents

Contents

In this chapter we come to the core of this thesis, namely to test how the Fribourg construction performs on real test automata. We are interested in two things. First, how the different versions of the Fribourg construction compare to each other. That is, with which optimisations the Fribourg construction is most efficient. Second, we compare the Fribourg construction to other complementation constructions. We can refer ot the first type of investigations as the *internal* tests, and to the second one as the *external* tests.

To do these investigations, we implemented the Fribourg construction as a plugin for an ω-automata manipulation tool called GOAL. GOAL already contains implementations of the most important Büchi complementation constructions. That is, with our plugin, the Fribourg construction lives next to these other constructions in the GOAL tool, and can be easily compared to them. With the plugin in place, we then performed the actual internal and external tests. To do so, we defined a set of test data consisting of totally 22.000 automata. The performance investigation consists then in basically complementing each of these automata with the different constructions, and comparing the results. Our main performance metric is the number of generated states for the output automata. The computations, which due to the complexity of Büchi complementation are quite heavy, were executed on a high-performance computing cluster at the University of Bern, Switzerland.

In this chapter, we are going to describe each of these points, including our concrete experiment setup. The results of the tests are presented and discussed in Section **??**.

4.1 GOAL

GOAL stands for Graphical Tool for Omega-Automata and Logics and has been developed at the National University of Taiwan since 2007 [10, 11]. The tool is based on the three pillars, ω-automata, temporal logic formulas, and games. It allows to create instances of each of these types, and manipulate them in a multitude of ways. Relevant for our purposes are the ω-automata capabilities of GOAL.

With GOAL, one can create Büchi, Muller, Rabin, Streett, parity, generalised Büchi, and co-Büchi automata, either by manually defining them, or by having them randomly generated. It is then possible to perform a plethora of operations on these automata. The entirety of provided operations are too many to list, but they include containment testing, equivalence testing, minimisation, determinisation, conversions to other ω-automata types, product, intersection, and, of course, complementation.

All this is accessible by both, a graphical and a command line interface. The graphical interface is shown in Figure 4.1. Automata are displayed in the main editor window of the GUI. They can be freely edited, such as adding or removing states and transitions, and arranging the layout. There are also various layout algorithms for automatically laying out large automata. Most of the functionality provided by the graphical interface is also accessible via a command line mode. This makes it suitable for automating the execution of operations.

For storing automata, GOAL defines an own XML-based file format, called GOAL File Format, usually indicated by the file extension gff.

An important design concept of GOAL is modularity. GOAL uses the Java Plugin Framework (JPF) [1], a library for building modular and extensible Java applications. A JPF application defines so-called extension points for which extensions are provided. These extensions contain the actual functionality of the application. Extensions and extension points are bundled in plugins, the main building block of a JPF application. It is therefore possible to extend an existing JPF application by bundling a couple of

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

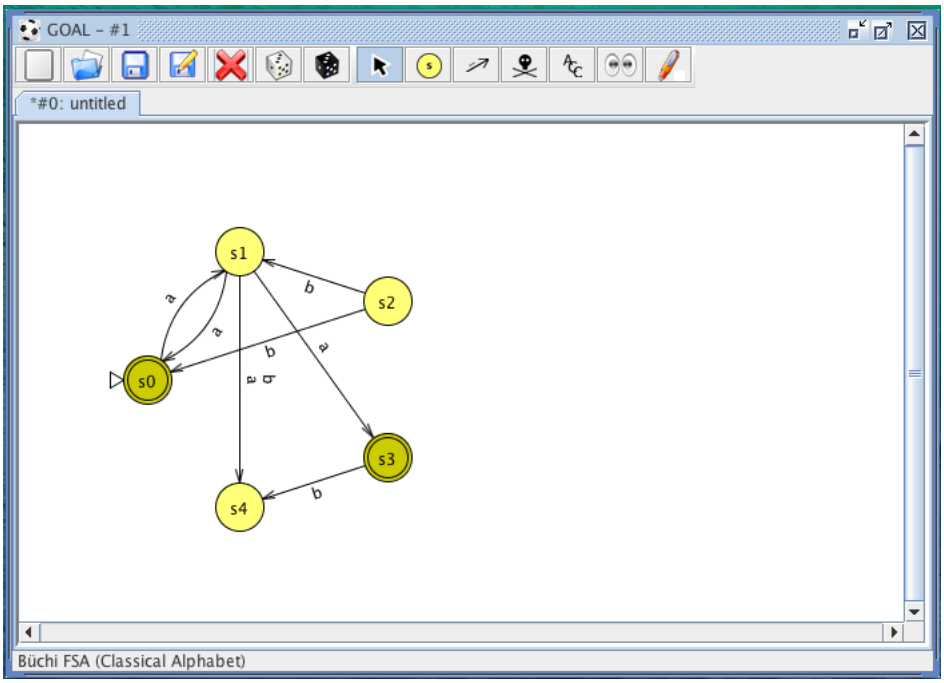


Figure 4.1: Graphical interface of GOAL.

new extensions for existing extensions points in a new plugin, and installing this plugin into the existing application. On the next start of the application, the new functionality will be included, all without requiring to recompile the existing application or to even have its source code.

GOAL provides a couple of extensions points, such as *Codec*, *Layout*, or *Complementation Construction*. An extension for *Codec*, for example, allows to add the handling of a new file format which GOAL can read from and write to. With an extension for *Layout* one can add a new layout algorithm for laying out automata in the graphical interface. And an extension to Complementation Construction allows to add a new complementation construction to GOAL. This is how we added the Fribourg construction to GOAL, as we will further explain in Section 4.2.

There are a couple of Büchi complementation constructions pre-implemented in GOAL. Table 4.1 summarises them, showing for each one its name on the graphical interface and in the command line mode, and the reference to the paper introducing it. As can be seen, the most important representants of all the four approaches (Ramsey-based, determinisation-based, rank-based, and slice-based, see Chapter **??**) are present. In addition to the listed constructions, GOAL also contains Kurshan's construction and classic complementation. These are for complementing DBW and NFA/DFA, respectively, and thus not relevant to us.

One of the constructions can be set as the default complementation construction. It is then possible to invoke this construction with the shortcut Ctrl-Alt-C. Furthermore, the default complementation constructions will be used for the containment and equivalence operations on Büchi automata, as they include complementation.

Complementation constructions in GOAL can define a set of options that can be set by the user. In the graphical interface this is done at the start of the operations via a dialog window, in the command line mode the options are specified as command line arguments. Figure 4.2 shows the options dialog of the Safra-Piterman construction. Complementation options allow to play with different configurations and variants of a construction, and we will make use of them for including the optimisations presented in Chapter **??** to our implementation of the Fribourg construction.

For most complementation constructions (all listed in Table 4.1 except the Ramsey-based construction) there is also a version for step-by-step execution. In this case, the constructions define so-called steps and stages, through which the user can iterate independently. This is a great way for understanding how a complementation construction works, and for investigating specific cases in order to potentially further improve the construction.

Table 4.1: The complementation constructions implemented in GOAL (version 2014-11-17).

Name	Command line	Reference
Ramsey-based construction	ramsey	Sistla, Vardi, Wolper (1987) [8]
Safra's construction	safra	Safra (1988) [6]
Modified Safra's construction	modfiedsafra	Althoff (2006) [1]
Muller-Schupp construction	ms	Muller, Schupp (1995) [4]
Safra-Piterman construction	piterman	Piterman (2007) [5]
Via weak alternating parity automaton	wapa	Thomas (1999) [9]
Via weak alternating automaton	waa	Kupferman, Vardi (2001) [3]
Rank-based construction	rank	Schewe (2009) [7]
Slice-based construction (preliminary)	slice -p	Vardi, Wilke (2007) [12]
Slice-based construction	slice	Kähler, Wilke (2008) [2]

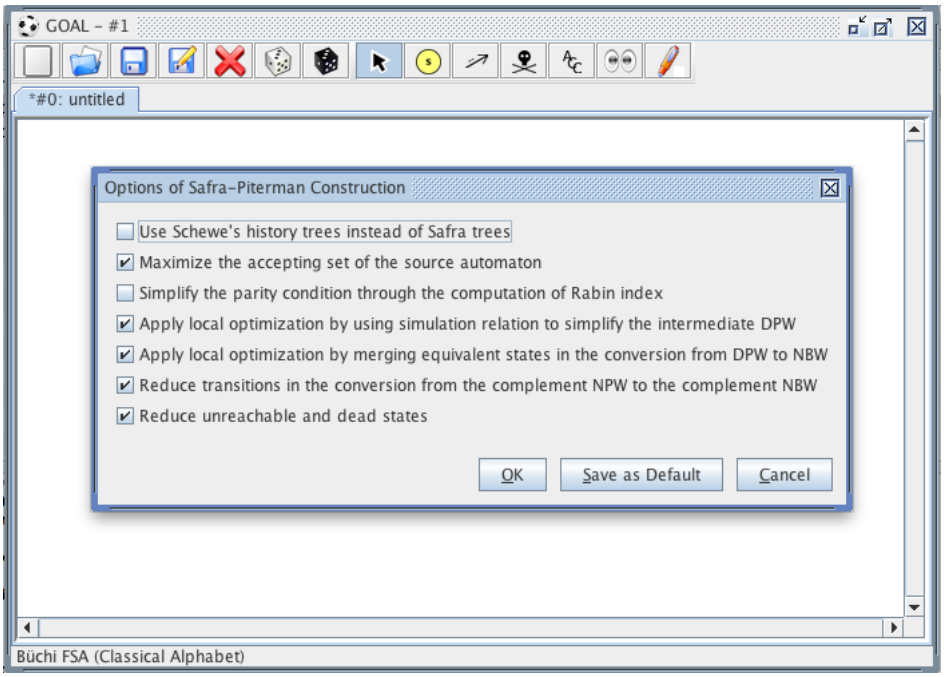


Figure 4.2: Complementation constructions in GOAL can have a set user-selectable options. Here the options of the Safra-Piterman construction.

Table 4.2: The options for the Fribourg construction.

Code	Description
m1	Component merging optimisation
m2	Single 2-coloured component optimisation
r2c	Deleting states with rightmost colour 2, if automaton is complete
c	Make input automaton complete
macc	Maximise accepting states of input automaton
r	Remove unreachable and dead states from output automaton
rr	Remove unreachable and dead states from input automaton
b	Use the "bracket notation" for state labels

4.2 Implementation of the Fribourg Construction as a GOAL Plugin

We implemented the Fribourg construction, including its optimisations, in Java as a plugin for GOAL. This means that after installing out plugin to an existing GOAL installation[2], the Fribourg construction will be an integral part of GOAL and can be used in the same way as any other pre-existing complementation construction.

4.2.1 Options for the Fribourg Construction

To keep the Fribourg construction flexible, we made use of options. The three optimisations described in Section 3.4 are presented to the user as selectable options. Additionally, we included several further options. Table 4.2 lists them all. For convenience, we use for each options a short code name, which is also used as the option name in the command line mode.

The first three items in Table 4.2, m1, m2, and r2c, correspond to the optimisations M1, M2, and R2C, described in Section 3.4. As the M2 optimisation requires M1, our implementation makes sure that the m2 option can only be selected if also the m1 option is selected. The c option, for making the input automaton complete before starting the actual construction, is intended to be used with the r2c option. In this way, the R2C optimisation can be forced to apply. This idea results from previous work that investigated whether making the input automaton complete plus the application of the R2C optimisation brings an improvement over the bare Fribourg constructoin [**?**]. The result was negative, that is, the construction performs worse with this variant on practical cases. Also note that using the c option alone, very likely decreases the performance of the construction, because the automaton is made bigger if it is not complete.

The macc and r options are common among the other complementation constructions in GOAL. The first one, macc, maximises the accepting set of the input automaton. That means, it makes as many states accepting as possible without changing the automaton's language. This should help to make the complement automaton smaller. The r options prunes unreachable and dead states from the complement automaton. Unreachable states are states that cannot be reached from the initial states, and dead states are states from where no accepting state can be reached. Clearly, all the runs containing an unreachable or dead state are not accepting, and thus these states can be removed from the automaton without changing its language. The complement automaton can in this way be made smaller. The rr option in turn removes the unreachable and dead states from the *input* automaton. That is, it makes the input automaton smaller, before the actual construction starts, what theoretically results in smaller complement automaton.

Finally, the b option affects just the display of the state labels of the complement automaton. It uses an alternative notation which uses different kinds of brackets, instead of the explicit colour number, to indicate the colours of sets. In particular, 2-coloured sets are indicated by square brackets, 1-coloured sets by round parenthesis, and 0-coloured sets by curly braces. Sets of states of the upper part of the

[2] As the plugin interfaces of GOAL have recently changed, the can be used only for GOAL versions 2014-11-17 and newer.

automaton are enclosed by circumflexes. This notation, although being very informal, has proven to be very convenient during the development of the construction.

When we developed the plugin, we aimed for a complete as possible integration with GOAL. We integrated the Friboug construction in the graphical, as well as in the command line interface. We added a step-by-step execution of the construction in the graphical interface. We provided that customised option configurations can be persistently saved, and reset to the defaults at any time. We also integrated the Fribourg construction in the GOAL preferences menu so that it can be selected as the default complementation construction. In this way, it can be invoked with a key-shortcut and it will also be used for the containment and equivalence operations. Our goal is that once the plugin is installed, the Fribourg construction is as seamlessly integrated in GOAL as all the other pre-existing complementation construction.

The complete integration allows us to publish the plugin so that it can be used by other GOAL users. At the time of this writing, the plugin is accessible at `http://goal.s3.amazonaws.com/Fribourg.tar.gz` and also over the GOAL website[3]. The installation is done by simply extracting the archive file and copying the contained folder to the plugins/ folder in the GOAL system tree. No compilation is necessary. The same plugin and the same installation procedure works FOR Linux, Mac OS X, Microsoft Windows, and other operating systems that run GOAL.

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.

4.3 Verification of the Implementation

See UBELIX/jobs/2014-11-25

Can we do a complement-equivalence test with all the 11,000 automata of size 15 in the test set?

Of course it is needed to test whether our implementation produces correct results. That is, are the output automata really the complements of the input automata? We chose doing so with an empirical approach, taking one of the pre-existing complementation constructions in GOAL as the "ground truth". We can then perform what we call complementation-equivalence tests. We take a random Büchi automaton and complement it with the ground-truth construction. We then complement the same automaton with our implementation of the Fribourg construction, and check whether the two complement automata are equivalent. Provided that the ground-truth construction is correct, we can show in this way that our construction is correct for this specific case.

We performed complementation-equivalence tests for the Fribourg construction with different option combinations. In particular, we tested the configurations m1, m1+m2, c+r2c, macc, r, rr, and the construction without any options. For each configuration we tested 1000 random automata of size 4 and with an alphabet of size 2 to 4. As the ground-truth construction we chose the Safra-Piterman construction. In all cases the complement of the Fribourg construction was equivalent to the complement of the Safra-Piterman construction.

Doubtlessly, it would be desirable to test more, and especially bigger and more diverse automata. However, by doing so one would quickly face practical problems due to long complementation times with bigger automata and larger alphabets, and high memory usage. For our current purpose, however, the tests we did are enough for us to be confident that our implementation is correct.

4.4 Test Data

For our set of sample automata, we chose to adopt the test set that has been created and used for another empirical performance comparison of Büchi complementation constructions by Tsai et al. [**?**] (the first author, Tsai, being the main author of GOAL). This test set consists of 11000 automata of size 15, and 11000 automata of size 20. Each set of 11000 automata consists of 110 groups containing 100 automata each. The 110 groups result from the cartesian product of 11 transitions densities and 10 acceptance densities.

[3] http://goal.im.ntu.edu.tw/

At this point, we have to explain the notions of transition and acceptance densitiy, as they are defined in [**?**] and also implemented in GOAL. The transition density defines the number of transitions in an automaton. In particular, if the transition density is t and the automaton has n states, then the automaton contains tn transitions for every symbol of the alphabet. In other words, the transition density is the average number of outgoing (and incoming) transitions of a state for each symbol of the alphabet. The transition densities in the test set range from 1 to 3 in steps of 0.2, that is, there are the 11 instances 1.0, 1.2, 1.4, 1.6, 1.8, 2.0, 2.2, 2.4, 2.6, 2.8, 3.0. The acceptance density in turn is the percentage of states that are accepting states. It is thus a number between 0 and 1. In the test set, the 10 acceptance density classes range from 0.1 to 1.0 in steps of 0.1.

Each of the 110 transition density and acceptance density combination groups contains thus 100 automata. These automata were generated at random with the random automata generator of GOAL. The alphabet size of all the automata is 2. According to Tsai et al. The alphabet size of all the automata is 2 According to Tsai et al. this test set generates a large class of complementation problems ranging from easy to hard [**?**]. The test set is available on the GOAL website[4]. At the time of this writing, the direct link to the data is `http://goal.im.ntu.edu.tw/wiki/lib/exe/fetch.php?media=goal:ciaa2010_automata.tar.gz`.

The reason that we chose this existing test set, instead of for example generating our own one, is that it has been previously used and is thus an established reference point. As it is commonly accepted in disciplines that rely on performance measurements via test sets (for example artificial intelligence), we also think that it is imporant that there is established test data that can be used as an objective benchmark. This is to avoid that self-made test data is biased toward the technology whose performance is to evaluate. By using the test set of Tsai et al., we are taking a step in this direction. Furthermore, the experiments conducted by Tsai et al. are similar to our ones, so there might be to notice interesting parallels or differences in the results. The same test set has also been used by an earlier performance investigation of the Fribourg construction in [**?**].

Besides the test set of the GOAL automata, we did all the tests also on the first four Michel automata:

- Michel N1: 3 states, 7 transitions, 1 accepting state
- Michel N2: 4 states, 14 transitions, 1 accepting state
- Michel N3: 5 states, 23 transitions, 1 accepting state
- Michel N4: 6 states, 34 transitions, 1 accepting state

These are the Michel automata that can be processes with our implementation and on our execution environment. Complementing Michel automata N5 and above would already by far exceed our available computing and time resources.

4.5 Internal Tests

In the internal tests we want to find out which version of the Fribourg constructionn is the most efficient one. With most efficient, we mean the one that produces on average the least number of states on the set of test automata.

As presented in Section 3.4, there are three optimisations to the Fribourg construction:

1. Removing a state if its righmost colour is 2 during the construction (only if the input automaton is complete)
2. Merge adjacent sets within a state
3. Reduce the overall number of 2-coloured sets

As in Section 3.4, we refer to the first optimisation as R2C, the second one as M1, and the third one as M2. These optimisations have the following dependencies:

1. R2C can only be applied if the input automaton is complete
2. M2 can only be applied if M1 is also applied

[4] http://goal.im.ntu.edu.tw/

Regarding the dependency of R2C, there are two possibilities. First (R2C-A), the R2C optimisation is selectively applied to the input automata which are complete, and not to the others. Second (R2C-B), all automata are made complete beforehand (by adding a sink state), and then the R2C optimisation is applied to all the automata.

Göttel [**?**] has compared R2C-B to a plain version of the Fribourg construction where no optimisations at all are applied. He used the same test data as we do. The result was that R2C-B produces on average slightly less states, but the peak number of generated states are higher than in the plain version. It will be interesting to see if we can replicate these results, and how the selective application of the R2C optimisation (R2C-A) performs compared to R2C-B.

Regarding the dependencies of the M1 and M2 optimisation, there are only two cases we can test. First, M1 alone, and second, M1 and M2 together. Assuming that the R2C optimisation adds a certain performance gain on top of an existing construction, we can then combine the better one of M1 and M1+M2 with R2C. We can already reveal at this point that M1 performs slightly better than M1+M2 on our test set, even though M1+M2 has a better theoretical worst-case complexity. This topic is further discussed in Chapter 5. Thus, the version that we will want to test is M1+R2C.

Furthermore, we can also investigate the effect of some generic optimisations on our construction. The most generic optimisations, which are included in most complementation constructions in GOAL, and also our plugin with the Fribourg construction, are:

1. Maximise the acceptance set of the input automaton (MACC)
2. Remove unreachable and dead states from the output automaton (R)

By applying these two optimisations to the best version of the Fribourg construction, we can see how far we can go with tweaking our construction, with respect to our set of test automata.

Summarising, for the internal tests we are going to carry out runs of the following 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+MACC+R

4.6 External Tests

Justification why to use only piterman, slice, and rank
UBELIX/jobs/2014-10-09:
Made test with complementing the first 10 of the size 15 test set with all constructions, and only piterman, slice, and rank (and safra) completed all of them.
See Tsai (2011) [**?**] page 5: they compared ramsey, piterman, rank, and slice. But ramsey couldn't complement any of the 11,000 automata of size 15.
Ramsey, piterman, rank, and slice are representative for the four main complementatio approaches: Ramsey-based, determinization-based, rank-based, and slice-based.

Test configurations

- Fribourg+M1+R2C
- Piterman+EQ+SIM+RO
- Rank+TR+RO
- Slice+P+RO+MADJ+EG

4.7 Execution Environment

UBELIX Linux cluster, runs Sun Grid Engine (load scheduler).
All the jobs are executed on machines belonging to one of the two following queues:

- long.q
  - hnode 47–49
  - Intel Xeon E5-2695v2 2.40GHz
  - 24 CPU cores (slots)
  - 96 GB RAM
  - → 4 GB RAM per core (slot)
  - h_cpu limit: > 200:00:00 (not sure)
- mpi.q
  - hnode 01–42
  - Intel Xeon E5-2665 2.40GHz
  - 16 CPU cores (slots)
  - 64 GB RAM
  - → 4 GB RAM per core (slot)
  - h_cpu limit: 72:00:00
  - h_rt limit: 73:00:00
- highmem.q
  - jnode 01–21
  - Intel Xeon E5-2665 2.40GHz
  - 16 CPU cores (slots)
  - 256 GB RAM
  - → 16 GB RAM per core (slot)

Tests

1. Internal on GOAL testset
2. External on GOAL testset
3. Internal on Michel automata
4. External on Michel automata
5. Completeness of GOAL testset
6. Universality of GOAL testset

The tests are successful with the following resources

Test	Queue	Slots	Job memory limit	Job CPU time limit	CPU time limit per automaton	Memory limit per automaton	Notes
1 and 2	mpi.q	4	4 GB	72:00:00	600 sec.	1 GB	rank -tr -ro has to be run on 10 partitions of the test set
3 and 4	highmem.q	4	16 GB	72:00:00	None	14 GB	piterman -eq -sim -ro out of memory on Michel N4
4	mpi.q	4	4 GB	72:00:00	None	1 GB	
5	mpi.q	4	4 GB	72:00:00	None	2 GB	universal -m piterman -eq -ro

Chapter 5

Results and Discussion

Chapter 6

Conclusions

Appendix A

Plugin Installation and Usage

Bibliography

[1] 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.

[2] 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.

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

[4] 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.

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

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

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

[8] 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.

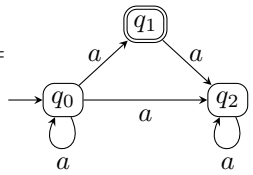
[9] W. Thomas. Complementation of Büchi Automata Revisited. In J. Karhumäki, H. Maurer, G. Păun, et al, eds., *Jewels are Forever*. pp. 109–120. Springer Berlin Heidelberg. 1999.

[10] 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.

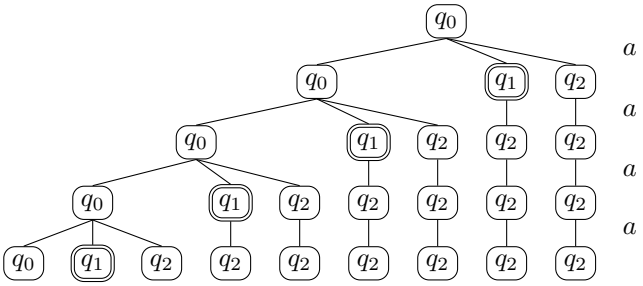
[11] Y.-K. Tsay, Y.-F. Chen, M.-H. Tsai, et al. Goal Extended: Towards a Research Tool for Omega Automata and Temporal Logic. In C. Ramakrishnan, J. Rehof, eds., *Tools and Algorithms for the Construction and Analysis of Systems*. vol. 4963 of *Lecture Notes in Computer Science*. pp. 346–350. Springer Berlin Heidelberg. 2008.

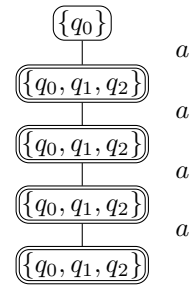
[12] M. Y. Vardi, T. Wilke. Automata: From Logics to Algorithms. In J. Flum, E. Grädel, T. Wilke, eds., *Logic and Automata: History and Perspectives*. vol. 2 of *Texts in Logic and Games*. pp. 629–736. Amsterdam University Press. 2007.

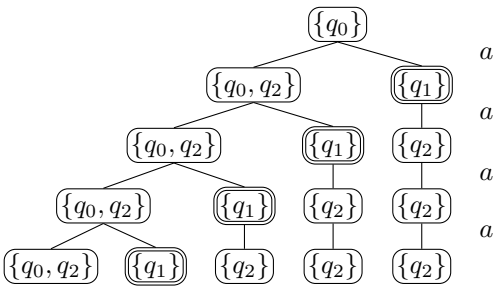
$A =$



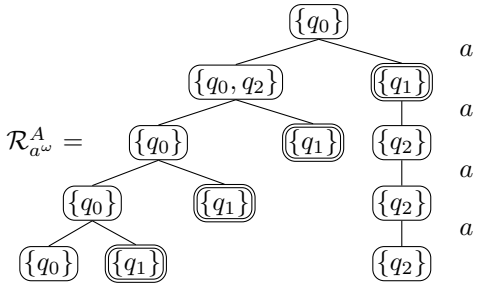
$\mathcal{T}^A_{a^\omega} =$







$\mathcal{R}^A_{a^\omega} =$



$\mathcal{G}^A_{a^\omega} =$