Performance Investigation of a Subset-Tuple Büchi Complementation Construction

Daniel Weibel

November 9, 2014

Contents

Chapter 1

The Büchi Complementation Problem

1.1 Preliminaries

1.1.1 Büchi Automata

Büchi automata have been introduced in 1962 by Büchi [2] in order to show the decidability of monadic second order logic; over the successor structure of the natural numbers [1].

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 [25] (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 [24].

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 [2]. 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 [24][18]. 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 [4][16]. In the case of Büchi automata, this inequivalence is the main cause that Büchi complementation problem is such a hard problem [15] and until today regarded as unsolved.

1.1.2 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) [13], Rabin (Rabin automata, 1969) [17], Streett (Streett automata, 1982) [21], and Mostowski (parity automata, 1985) [12].

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 [9]. 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 [9], which will be used for certain Büchi complementation construction, as we will see in Section1.4. 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* [10]. Another proof given in [22]. 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 1.4.2.

It turned out that also all the other types of the just introduced ω-automata, non-deterministic and determinstic, are equivalent among each other [18][7][6][9][22]. 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 1.1

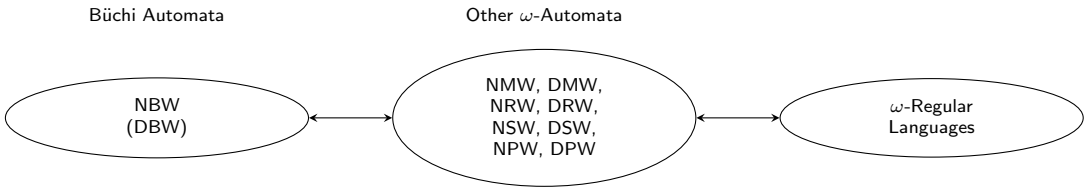


Figure 1.1: Non-deterministic Büchi automata (NBW) are expressively equivalent to Muller, Rabin, Streett, and parity automata (both deterministic and non-deterministic), and to the ω-regular languages. Deterministic Büchi automata (DBW) are less expressive than NBW.

1.1.3 Complementation of Büchi Automata

Büchi automata are closed under complementation. This result has been proved by Büchi himself when he introduced Büchi automata in [2]. 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 [22] (p. 15).

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

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

To illustrate this, we can briefly look at the complementation of the ordinary finite state automata on finite words (FA). FA are also closed under complementation [4] (p. 133). A DFA can be complemented by simply switching its accepting and non-accepting states [4] (p. 133). Now,

since NFA and DFA are equivalent [4] (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 [8], that can complement a DBW to a NBW in polynomial time. The size of the complement NBW is furthermore at most the double of the size of the input DBW.

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

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

1.1.4 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 [2] 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 [20]. 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 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 $n!$ [11]. 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 [26]. 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 $2^{O(n\,log\,n)}$ [26].

Michel's lower bound remained valid for almost two decades until in 2006 Yan showed a new lower bound of $(0.76n)^n$ [26]. 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) [19]. 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 [24] p. 8.

1.2 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

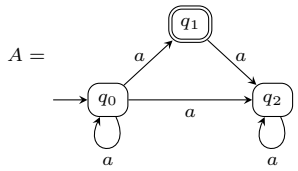


Figure 1.2: Example NBW A that will be used in different places throughout this thesis. The alphabet of A consists of the single symbol a, consequently, A can only process the single ω-word a^ω. This word is rejected by A, so the automaton is empty.

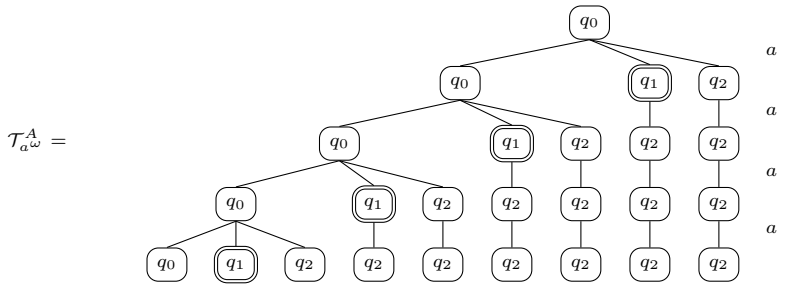


Figure 1.3: Automaton A and the first five levels of the run tree of the runs of A on the word a^ω.

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 **??**.

1.2.1 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 1.8 shows the first few levels of the run tree of the example automaton A from Figure 1.2 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 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

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

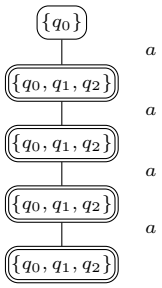


Figure 1.4: Automaton A and the first five levels of the run tree of the runs of A on the word a^ω.

with the set of A-states it includes. Figure 1.4 shows this effect with our example automaton from Figure 1.2 and the word a^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 1.4 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 1.4). 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 1.4 and 1.8, the reason for this problem becomes apparent. If we look for example at the second level of the subset-construction tree we can deduce that there must be an A-run that visits the accepting A-state q_1. Let us call this run r_{q_1}. However, at the third level, we cannot say anything about r_{q_1} anymore, whether it visits one of the non-accepting states or again q_1 on the third level, or whether it even ended at the second level. In turn, what we know on the third level in our example is that there is again an A-run, r'_{q_1}, that visits q_1. However, whether r'_{q_1} is r_{q_1}, and in turn the future of r'_{q_1} cannot be deduced. In our example we end up with the situation that there are infintiely many visits to q_1 in the unifed B-run, but we don't know if the reaon for this are one or more A-runs that visit q_1 infinitely often, or infinitely man A-runs where each one visits q_1 only finitely often (the way it is in our example). In the first case, it would be correct ot accept the B-run, in the second case however it would be wrong as the input automaton A does not accept the word. The subset construction does not distinguish these two cases and hence the determinised automaton B may accept words that the input automaton A rejects. In general, the language of an output DBW of the subset construction is a superset of the language of the input NBW.

This raises the question how the subset construction can be minimally modified such that the output automaton is equivalent to the input automaton. One solution is to not mix accepting and non-accepting A-states in the B-states. That is, instead of creating one B-state that contains all the A-states, as in the subset construction, one creates two B-states where one contains the accepting A-states and the other the non-accepting A-states. Such a construction has been formalised in [23]. The output automaton B is then not deterministic, but it is equivalent to A. The type of run analysis trees that correspond to this refined subset construction are split trees. Figure 1.9 shows the first five levels of the split tree of our example automaton A on the word a^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

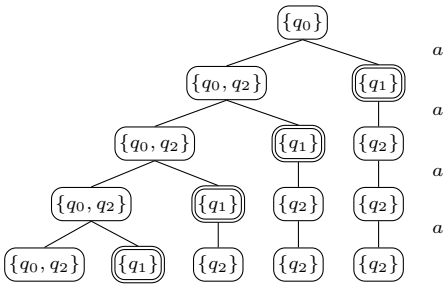


Figure 1.5: Automaton A and the first five levels of the split tree of the runs of A on the word a^ω.

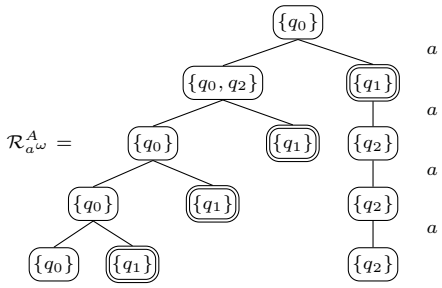


Figure 1.6: Automaton A and the first five levels of the reduced split tree of the runs of A on the word a^ω.

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 [14], cf. [**?**].

1.2.2 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 1.6 shows the reduced split tree corresponding to the split tree in Figure 1.9. As can be seen, only one occurrence of each A-state on each level is kept, the other are discarded. To allow this, however, the order of the accepting and non-accepting siblings in the tree matters. Either the accepting child is always put to the right of the non-accepting child (as in our example in Figure 1.6, or vice versa. We call the former variant a right-to-left reduced split tree, and the latter a left-to-right reduced split tree. In this thesis, we will mainly adopt the right-to-left version.

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

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

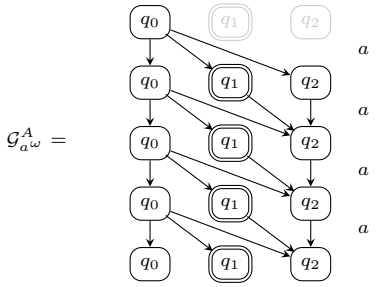


Figure 1.7: Automaton A and the first five levels of the run DAG of the runs of A on the word a^ω.

- Put the accepting child to the right of the non-accepting child on level $n+1$

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

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

We mentioned that the greedy run somehow represents the deleted runs. More precisely, the relation is as follows and has been proved in [25]: if any of the deleted runs is a prefix of a run that is Büchi-accepted (that is, an infinite run visiting infinitely many accepting A-states), then the greedy run is so too. That means that if the greedy cannot be expanded to a Büchi-accepting run, then none of the deleted runs could be either. Conversely, if any of the deleted runs could become Büchi-accepting, then the greedy run can so too. So, the greedy run is sufficient to indicate the existence or non-existence of a Büchi-accepting run with this prefix, and it is safe to delete all the other runs.

1.2.3 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 1.12 shows the first five rows of the run DAG of the example automaton in Figure 1.2 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.

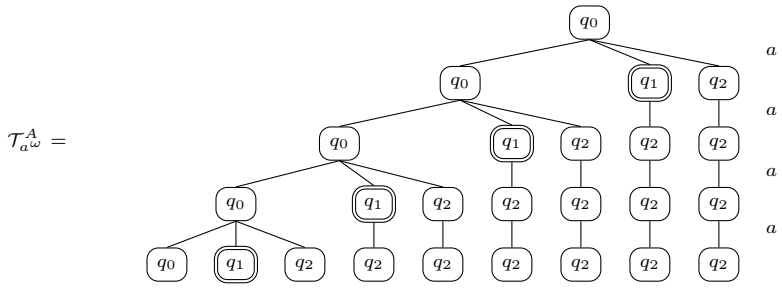


Figure 1.8: Automaton A and the first five levels of the run tree of the runs of A on the word a^ω.

1.3 Run Analysis

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

1.3.1 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 1.8 shows an example automaton A and the first five levels of the run tree for the word $w = a^\omega$ (infinite repetitions of the symbol a). Each branch from the root to one of the leaves represents a possible way for reading the first four positions of w. On the right, as a label for all the edges on the corresponding level, is the symbol that causes the depicted transitions.

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

We define by the width of a tree the maximum number of vertices occurring at any level [14]. 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.

1.3.2 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.

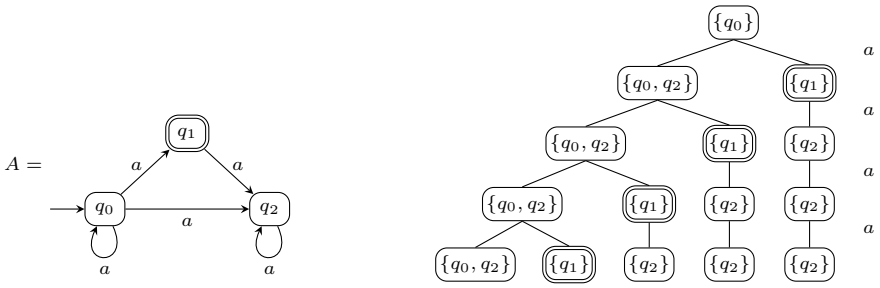


Figure 1.9: Automaton A and the first five levels of the split tree of the runs of A on the word a^ω.

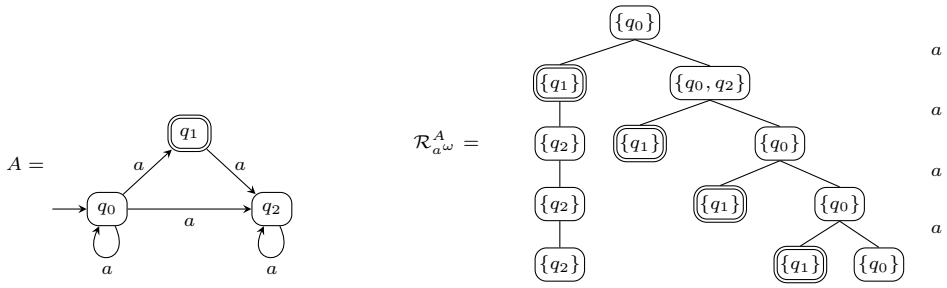


Figure 1.10: Automaton A and the first five levels of the left-to-right reduced split tree of the runs of A on the word a^ω.

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 [23]. Generally, the idea of treating accepting and non-accepting states separately is important in the run analyis of Büchi automata.

1.3.3 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 1.9 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 [5], cf. [3]. 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 [14]. Formal definitions os split trees can be found in [5][3].

1.3.4 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.

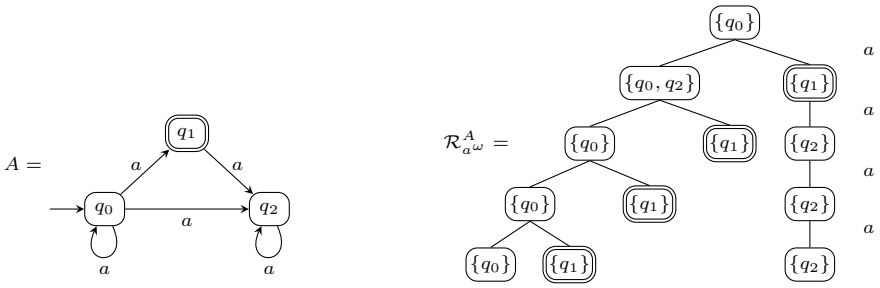


Figure 1.11: Automaton A and the first five levels of the left-to-right reduced split tree of the runs of A on the word a^ω.

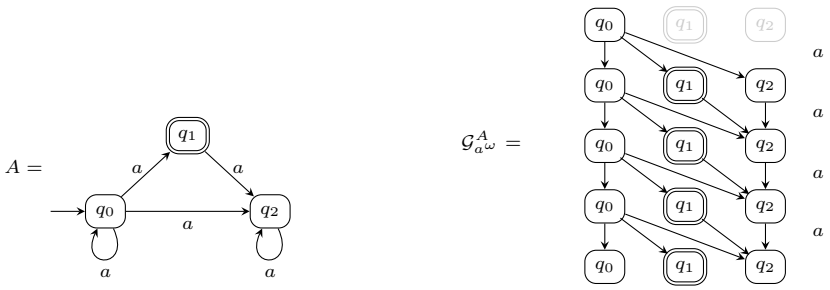


Figure 1.12: Automaton A and the first five levels of the run DAG of the runs of A on the word a^ω.

1.3.5 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 1.12 shows the run DAG of automaton A on the word $w = a^\omega$.

1.4 Review of Büchi Complementation Constructions

1.4.1 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 [1].

1.4.2 Determinisation-Based Approaches

1.4.3 Rank-Based Approaches

1.4.4 Slice-Based Approaches

1.5 Empirical Performance Investigations

Bibliography

[1] 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.

[2] 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.

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

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

[5] 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.

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

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

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

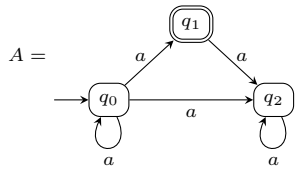
[9] C. Löding. Optimal Bounds for Transformations of ω-Automata. In C. Rangan, V. Raman, R. Ramanujam, eds., *Foundations of Software Technology and Theoretical Computer Science*. vol. 1738 of *Lecture Notes in Computer Science*. pp. 97–109. Springer Berlin Heidelberg. 1999.

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

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

[12] 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.

[13] 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.



[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

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

[20] 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.

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

[22] 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.

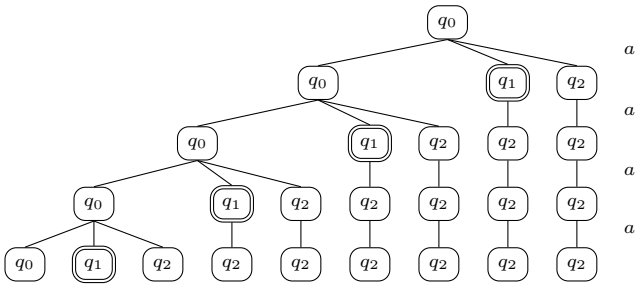
[23] U. Ultes-Nitsche. A Power-Set Construction for Reducing Büchi Automata to Non-Determinism Degree Two. *Information Processing Letters*. 101(3):pp. 107 – 111. 2007.

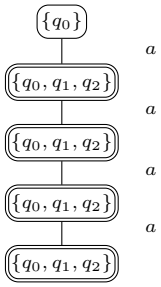
[24] M. Vardi. An automata-theoretic approach to linear temporal logic. In F. Moller, G. Birtwistle, eds., *Logics for Concurrency*. vol. 1043 of *Lecture Notes in Computer Science*. pp. 238–266. Springer Berlin Heidelberg. 1996.

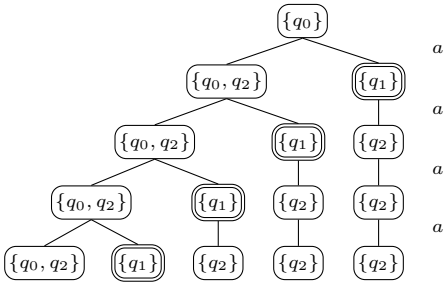
[25] M. Y. Vardi, T. Wilke. Automata: From Logics to Algorithms. In J. Flum, E. Grädel, T. Wilke, eds., *Logic and Automata: History and Perspectives*. vol. 2 of *Texts in Logic and Games*. pp. 629–736. Amsterdam University Press. 2007.

[26] 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.

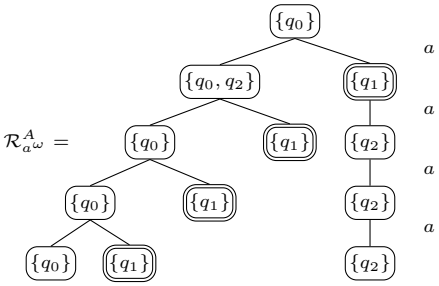
$\mathcal{T}^{A}_{a^\omega} =$







$\mathcal{R}^{A}_{a^\omega} =$



$\mathcal{G}^{A}_{a^\omega} =$