Empirical Performance Investigation of a Büchi Complementation Construction

Daniel Weibel

August 3, 2015

Abstract

This will be the abstract.

Acknowledgements

Contents

Chapter 1

The Fribourg Construction

Contents

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

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

The aim of this chapter is not to give a formally precise description of the Fribourg construction. This has been done already by its authors in [1]. The same applies for proofs of correctness and calculations of complexity. Rather, the aim of this chapter is to describe the construction in an intuitive and practically oriented way. We want to convey the concrete steps to take when, figuratively, standing with a marker in front of a whiteboard with an automaton to complement. For the formal background we refer to [1].

1.1 The Construction

In this section we explain the basic Fribourg construction without any applied optimisations. The construction consists of two sub-constructions that we call the upper-part construction and the lower-part construction. In the following we first describe the parts of the construction that are common to these two sub-construction (Section 1.1.1). Then, in Section 1.1.2 and 1.1.3, we describe the points that are specific to the upper-part construction and the lower-part construction, respectively. In these two subsections, we also illustrate the application of the construction with a simple example that we work through step by step from the initial input automaton to the final complement automaton.

1.1.1 Basics

Below we first give a high-level view of the construction, and then explain how the construction generates new states. The generation of states is the heart of the construction, and common to the two sub-constructions that we explain in detail in the subsequent two subsections.

High-Level View

As mentioned, the Fribourg construction consists of two sub-constructions, the *upper-part construction* and the *lower-part construction*. These two sub-constructions are chained together, such that the output of the upper-part construction becomes the input of the lower-part construction. Figure 1.1 illustrates this in a functional view of the Fribourg construction.

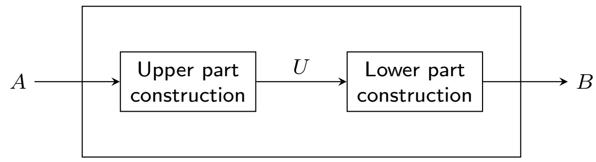


Figure 1.1: Functional view of the Fribourg construction with an automaton A as input and its complement B as output.

The automaton A to be complemented is the input for the upper-part construction. The upper-part construction builds up a new automaton by starting with an initial state and adding state by state. The output of the upper part constructio is an intermediate automaton U. This intermediate automaton U will be the *upper part* of the final complement automaton. The lower-part construction takes U as

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

input and adds further states and transitions to it until finally outputting an automaton B which is the complement of the input automaton A. We say that the lower-part construction attaches the *lower part* of the final complement automaton to the upper part. This terminology comes from the fact that it is convenient to draw the states of the lower part below the states of the upper part [1]. This also explains why the two sub-constructions are calleed upper-part construction and lower-part construction.

Generation of States

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

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

The sequence of components of each state of the Fribourg construction is imposed by a slice (that is, a level) of a reduced split tree. We will explain the process of going from an existing state over the slices of a reduced split tree to a successor state further below. For now, we look at how a slice defines the structure of a state. Figure 1.2 shows a reduced split tree (on the left), and for each slice (framed by a shaded box), the state that this slice would define (on the right).

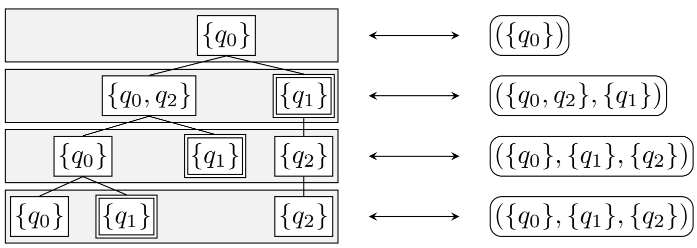


Figure 1.2: Relation between slices of a reduced split tree (left) and states of the output automaton of the Fribourg construction (right). The sequence of nodes in a slice determines the sequence of components in a state.

As can be seen in Figure 1.2, each node of a slice gives rise to a component in the corresponding state. To this end, it does not matter whether the nodes in the slices are siblings or whather their familial relations are. It is important to note, however, that the order of the nodes in a slice must be preserved in the order of the components in a state. It is similarly possible to interpret a state as a slice of a reduced split tree (indicated by the double-ended arrows in Figure 1.2). In this case, it is not possible to deduce the ancestors of the resulting nodes, however, this is not necessary for the construction.

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

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

The state that results from thew new slice is the successor of the existing state for the given alphabet symbol. If, for example, we want to define the successor on the symbol a (a-successor) of the state $(\{q_0\})$, then as the first step we would create a slice representation of it that looks like the first slice from the top in Figure 1.2. Then, we create based on the input automaton the subsequent slice for symbol a, that might look like the second slice from the top in Figure 1.2. Interpreting this slice as a state results in $(\{q_0\}2, \{q_1\})$, which thus is the a-successor of $\{q_0\}$.

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

1.1.2 Upper-Part Construction

In this subsection, we recapitulate the upper-part construction, and demontrate its application with an example. The example is based on the input automaton in Figure 1.3.

Description

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

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

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

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

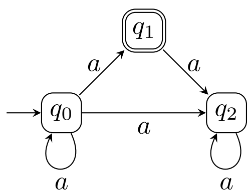


Figure 1.3: Non-deterministic Büchi automaton used for the examples in this section.

Example

We demonstrate the application of the upper part construction by applying it to the example automaton A in Figure 1.3. Note that this automaton has an alphabet size of one, and thus can run only on a single ω-word, namely a^ω. Furthermore, the automaton does not accept this word, which means that it does not accept *any* word and is thus empty. We selected this special automaton to keep our example handy and small. However, the construction works in exactly the same way for non-empty automata with larger alphabets.

The steps of the upper-part construction on the automaton A are showin in Figure 1.4 (a) to (d). Below, we walk through these steps one by one.

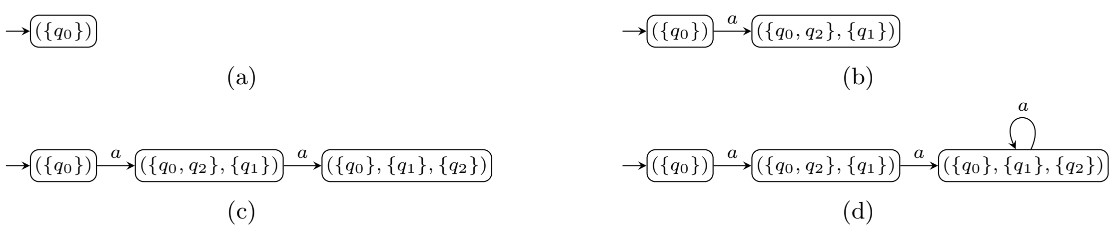
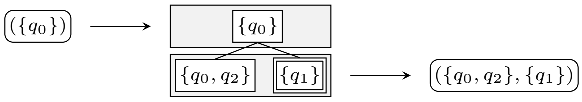


Figure 1.4: Steps of the upper-part construction applied to the example input automaton in Figure 1.3.

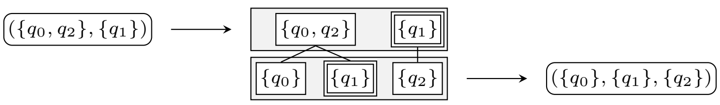
The first step in Figure 1.4 (a) is to start with the initial state. As can be seen, this state $(\{q_0\})$ contains only one component with the initial state of the input automaton A. This state will be the initial state of the final complement automaton B.

The next step Figure 1.4 (b) is to add the a-successor of the inital state $(\{q_0\})$. This is done by the state generation procedure that we described in the last section. In particular, this means: (1) interpret the state $(\{q_0\})$ as a slice of a reduced split tree, (2) based on the input automaton A, determine the subsequent slice for symbol a, and (3) interpret this new slice as a state. This new state is the a-successor of $(\{q_0\})$. We can illustrate these steps in the following way:



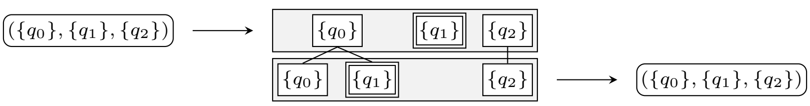
The state $(\{q_0\})$ results in a slice with only one node containing $\{q_0\}$. For creating the next slice, we determine the set of states of A that can be reached from $\{q_0\}$ on symbol a, which is $\{q_0, q_1, q_2\}$. According to the rules of reduced split trees, this set is split into a non-accepting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

The next step in Figure 1.4 (c) is to determine the a-successor the newly created state $(\{q_0, q_2\}, \{q_1\})$. Applying the same procedure as above, results in the following:



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

The last step in Figure 1.4 (d) consists in determining the a-successor of $(\{q_0\}, \{q_1\}, \{q_2\})$. The state generation procedure in this case looks as follows:



The state $(\{q_0\}, \{q_1\}, \{q_2\})$ results in a slice with three nodes. Again, we have to process these nodes from right to left. The node with $\{q_2\}$ has a non-accepting $\{q_2\}$ as its only child. Regarding the node with $\{q_1\}$, the reachable set in A is $\{q_2\}$ However, since q_2 already exists in the slice, it is omitted, and since this empties the set, no child of this node is added to the new slice. Regarding the last node with

$\{q_0\}$, the set of states reachable from $\{q_0\}$ is $\{q_0, q_1, q_2\}$, but again, due to the omission of q_2 and the splitting into an accepting and a non-accepting set, this results in the accepting right child $\{q_1\}$ and the non-accepting left child $\{q_0\}$. The state corresponding to this slice is $(\{q_0\}, \{q_1\}, \{q_2\})$, which is identical to the state we are currently processing. Consequently, we only add a transition from $(\{q_0\}, \{q_1\}, \{q_2\})$ back to itself.

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

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

1.1.3 Lower-Part Construction

Description

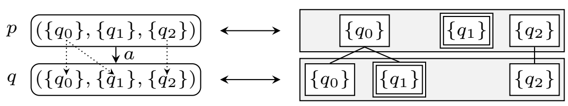
The lower-part construction takes the intermediate automaton resulting from the upper-part construction as input, and regards these states as the unprocessed states to be processed. This results in additional successors for the states of the upper part, and finally in the entire lower growing out of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

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

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

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

Regarding the first point, a component is accepting if it corresponds to an accepting node in the slice that defines it. Regarding the second point, each component (except the component of the intial state) has exactly one predecessor component. The predecessor component of a component c is the parent of c in a slice-representation of the two states. It can be imagined as merging the edges between two slices into the two corresponding states. These virtual edges then link each component of the successor state with its predecessor component in the predecessor state. For example, consider the following two states p and q, which are created by the state generation procedure on the example input automaton A in Figure 1.3:



Our goal is to identify the predecessor components of the components of state q. The slice representation of q and its predecessor state p on the right reveals the relation between the components of the two states, and we see that the predecessor components of $\{q_0\}$, $\{q_1\}$, and $\{q_2\}$ are $\{q_0\}$, $\{q_0\}$, and $\{q_2\}$, respectively. This is indicated by the dotted lines between the two states. For the example later in this section, we will re-use this notaton for indicating predecessor component relations.

Having explained the terms accepting component and predecessor component, we can now turn to the actual rules that determine the colour of a given component. One remark is however still necessary. The colouring function needs a way to recognise components that belong to the upper part of the output automaton. A way to guarantee this to preliminarily assign the colour -1 to all the components of the upper-part states. Figure 1.5 shows the rules that determine the colour of a component c.

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

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

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

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

Figure 1.5: Rules for determining the colour of a component c (in bold). The identifier c_{pred} denotes the predecessor component of c.

As can be seen in Figure 1.5 the colour rules are divided into two cases. Case A applies if the predecessor state does not contain any components with the colour 2, and Case B applies if the predecessor state contains one or more components with the colour 2. In each of the cases, the colour of a component c is unambiguously determined by the colour of the predecessor component c_{pred}, and by the fact whether c is an accepting component or not.

Note that in Case A, the possible colours of the predecessor component includes -1, but not 2. The missing of 2 is because obviously the colour of the predecessor component cannot be 2, because case A applies only if the predecessor stat contains *no* components with colour 2. The colour of the predecessor component can however be -1, if the predecessor state is an upper-part state. In case B, on the other hand, the colour -1 is absent, because if the predecessor state contains a 2-coloured component, it cannot be an upper-part state.

Having explained how the colours of components are determined, we now give some insight into the meaning of the colours. The purpose of the colours is to signalise the presence or absence of certain runs of the input automaton (input-runs). We divide these input-runs into *dangerous* and *safe* runs. Dangerous input-runs have visited at least one accepting input-state since entering the lower part (in othe terms, they made a right-turn). These runs are dangerous in the sense that they might become accepting if they keep visiting accepting input states. Safe runs, on the other hand, are runs that have not yet visited any accepting input-states since enetering the lower part. Input-runs that stay safe forever are non-accepting runs in the input automaton. The purpose of the distinction of safe and dangerous input-runs is in the end to recognise whether *all* input-runs on a specific words are safe, that is, non-accepting in the input automaton. Because in this case, the complement automaton must accept this words. In particular, the meaning of the three colours 0, 1, and 2 is as follows:

Colour 2 Signalises the presence of dangerous input-runs, that is, runs that have visited at least one accepting input-state. Colour 2 appears as soon as runs that have previously been safe visit an accepting component (see Figure 1.5 (a) lines 1 and 2). However, once a component has colour 2, it passes on this colour to all its successor components, no matter if they are accepting or non-accepting (see Figure 1.5 (b) line 3). This means that a path of 2-coloured components can only stop if at some point a component has no successor. In this case, we say that the colour 2 *disappears*.

Colour 1 Means basically the same as colour 2, namely the presence of dangerous runs. The reason that colour 1 exists is a caveat that can arise if we would assign colour 2 to *every* component whose path just made a right-turn. In this case, it could happen that constantly new "families" of 2-coloured components appear in a sequence of states, which at the smae time all disappear again. However, because of the constant appearance of new 2-coloured components, the disappearance of the existing 2-coloured components would not noticed, because the corresponding states always contain 2-coloured components. For this not to happen, if the predecessor state already contains 2-coloured components, then all the components that result from a right-turn get colour 1 instead of colour 2 (see Figure 1.5 (b) line 1). We say that these are dangerous runs that are “on hold”. That means that at first they only pass on their colour 1 to all their successors (see Figure 1.5 (b) line 2). However, as soon as all the other 2-coloured components in their states disappear, the become “active” in the sense that their successors now get colour 2 instead of colour 1 (see Figure 1.5 (a) line 3).

Colour 0 Signalises that all input-runs are safe. That means, these input-runs have not yet visited an accepting input-state since entering the lower part. If these runs stay safe indefinitely, then they are non-accepting in the input automaton, which is actually the crucial information that is needed for the complementation construction.

In this way, each component of the lower part gets one of the colours 0, 1, or 2. At the end of the construction, these colours determine which states of the lower part become accepting states. The rule is as follows:

Each state of the lower part containing *no* 2-coloured components is made accepting

The rationale behind this rule is that visiting such states infinitely often guarantees that the corresponding input-runs are non-accepting. With the definition of the acceptance set, the lower-part construction is finished. The resulting non-deterministic Büchi automaton, consisting of the upper part and the attachd lower part, constitutes the complement of the input automaton.

Example

Below, we demonstrate the application of the lower-part construction with an example. To this end, we continue the example that we left off after the upper-part construction in the last section. We will use the following notation for specifying the colour of a component c:

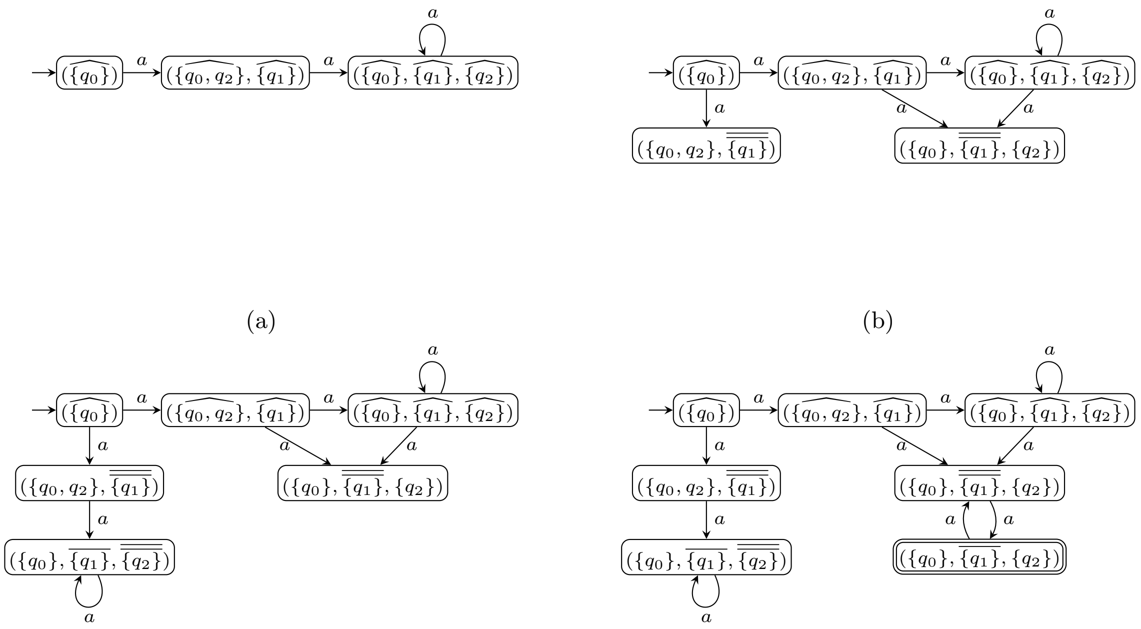
- $\widehat{c}$: colour -1
- $\{q\}$: colour 0
- $\overline{\{q\}}$: colour 1
- $\overline{\overline{\{q\}}}$: colour 2

Figure 1.6 shows some of the steps of the construction of the lower part. In Figure 1.6 (a), we start with the upper part that we previously constructed in the last section. The only difference is that we assigned colour -1 to all of the components of the upper part.

In Figure 1.6 (b), we created the a-successors of the three states of the upper part. The structure of these new states, apart from the colours, is determined by the same method that we used for the upper part. The only difference in the construction of the lower part is that each component is assigned a colour. For both new states, the predecessor states do not contain any 2-coloured components, thus we only need to consider the colour rules in Figure 1.5 (a). Regarding the colour of the predecessor components, they all have colour -1, thus we have to use the rule in Figure 1.5 (a) line 1 for all the new components. In this way, the components $\{q_0, q_2\}$, $\{q_0\}$, and $\{q_2\}$ are assigned colour 0, because they are non-accepting, and component $\{q_1\}$ gets colour 2, because it is accepting.

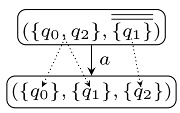
Note how we have to keep track for each component of the lower part whether it is accepting or non-accepting, and which is its predecessor component in the predecessor state.

In Figure 1.6 (c), we added the a-successor to the state $(\{q_0, q_2\}, \overline{\overline{\{q_1\}}})$. Disregarding the colours, this state has the form $(\{q_0\}, \{q_1\}, \{q_2\})$. Its predecessor state $(\{q_0, q_2\}, \overline{\overline{\{q_1\}}})$ contains a 2-coloured component, thus we have to use the colour rules in Figure 1.5 (b). Now we need to know which are the predecessor components of the components in $(\{q_0\}, \{q_1\}, \{q_2\})$. This information is contained in the two slices of the reduced split tree that were used to determine the structure of the new state. For the case of our two states, the successor relation of their components is as follows:



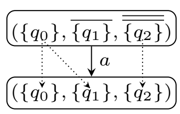
(a) (b) (c) (d)

Figure 1.6: Selected steps of the construction of the lower part, starting with the upper part



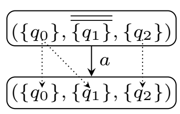
The predecessor component of $\{q_0\}$ is $\{q_0, q_2\}$, which has colour 0. Thus, we have to use the rule in Figure 1.5 (b) line 1, and $\{q_0\}$ gets the colour 0, because it is non-accepting. The predecessor component of $\{q_1\}$ is also $\{q_0, q_2\}$, and we have to use the same rule. However, since $\{q_1\}$ is accepting, it gets colour 1. Note how the use of colour 1 here prevents the introduction of a further 2-coloured component before an already existing 2-coloured component has disappeared. The predecesor component of $\{q_2\}$ is the 2-coloured $\overline{\overline{\{q_1\}}}$, and thus, according to the rule in Figure 1.5 (b) line 3, $\{q_2\}$ also gets colour 2.

Next, still in Figure 1.6 (c), we create in turn the successor state of $(\{q_0\}, \overline{\{q_1\}}, \overline{\overline{\{q_2\}}})$. The structure of the components stays the same for the successor. The succesor relation of the two states is as follows:



The result is that $\{q_0\}$ gets colour 0, $\{q_1\}$ gets colour 1, and $\{q_2\}$ gets colour 2. Thus, the successor state is identical to the current state, and we add loop.

Figure 1.6 (d) includes the remaining for arriving at the final complement automaton. First, we created the a-successor of the state $(\{q_0\}, \overline{\overline{\{q_1\}}}, \{q_2\})$. The complement successor relation of this state with its successor is as follows:



According to the rules in Figure 1.5 (b), component $\{q_0\}$ gets colour 0, $\{q_1\}$ gets colour 1, and $\{q_2\}$ gets colour 0. This results in a new state, since the colours are different from the ones in the current state. Interesting here is that we have the case that a 2-coloured component "disappears". This is because the component $\overline{\overline{\{q_1\}}}$ has no successor component in the successor state. For the a-successor of the new state $(\{q_0\}, \overline{\{q_1\}}, \{q_2\})$, this means in turn that we have to use the rules in Figure 1.5 (a), what results in the already existing state $(\{q_0\}, \overline{\overline{\{q_1\}}}, \{q_2\})$.

At this point, all the states in the automaton have been processed, and the construction is therefore completed. The only thing that remains to be done is to determine the accepting states of the automaton. The rule is that each state of the lower part that does not contain any 2-coloured component is an accepting state. In our automaton this applies only to the state $(\{q_0\}, \overline{\{q_1\}}, \{q_2\})$, which is thus the only accepting state of the automaton.

It can be easily seen that the automaton in Figure 1.6 accepts the word a^ω, which is not accepted by the input automaton in Figure **??**. Thus, the result of our construction is a correct complement of the input automaton.

A loose upper bound for the worst-case state complexity of the entire construction (including the construction of both, the upper and the lower part) has been calculated to be in $O((1.59n)^n)$, where n is the number of states of the input automaton [1]. This result is subject to refinement in ongoing research by the authors of the Fribourg construction. As mentioned, this publication ([1]) also contains a formal desciption and a proof of correctness of the construction.

1.2 Optimisations

There are several optimisations to the basic Fribourg construction, which all have the goal to reduce the size of the output automaton. These optimisations are like "add-ons" and can be added to the basic construction and combined in different ways.

In this section, we describe three of these optimisations. For easier reference, we define an abbreviation for each optimisation: R2C, M1, and M2. These optimisations will also be subject to the empirical performance investigation of the Fribourg construction, starting from the next chapter, and there we will use the same abbreviations.

1.2.1 R2C: Remove States with Rightmost 2-Coloured Components

The R2C optimisation can be summarised as follows:

If the input automaton is complete, then states of the lower part that have a rightmost component with colour 2 can be omitted.

This means that if during the construction of the lower part we determine a state that has a 2-coloured rightmost component, then we do not need to add this state to the automaton. This consequently also omits all the potential successors of this state from the automaton.

The reason for this is the following fact. If the input automaton is complete, then the rightmost component of an output-state *always* has at least one successor component (because of the completeness of the input automaton). However, this applies only to the rightmost component, as the successors of the other components might be omitted, due to the right-to-left precedence in right-to-left reduced split trees.

If the colour of the rightmost component is 2, then the successor component inherits this colour 2 (Figure 1.5 (b) line 3). This means inductively that all the successors of a state with a 2-coloured rightmost component will have a 2-coloured rightmost component. Or in other words, there is a 2-coloured component that will never disappear. Since states containing a 2-coloured component are non-accepting, these states form a cycle without a single accepting state, and this cycle can be removed without changing the language of the automaton.

Note, however, that the omission of these states can only be done if the input automaton is complete.

1.2.2 M1: Merge Components

The M1 optimisation allows the merging of adjacent components depending on their colour. With the merging of adjacent components, we mean the replacement of these components with their union (if regarding components as sets). The three merging rules are as follows.

1. Two adjacent 1-coloured components can be merged to a single 1-coloured component
2. Two adjacent 2-coloured components can be merged to a single 2-coloured component
3. A 2-coloured component adjacent to a 1-coloured component (in this order from left to right) can be merged to a single 2-coloured component

These mergings can be done recursively. This means that the result of a merging can again be subject to further merging, until nothing more can be merged. For example, the state $(\{q_0\}, \overline{\overline{\{q_1\}}}, \overline{\overline{\{q_2\}}}, \overline{\{q_3\}}, \overline{\{q_4\}}, \overline{\overline{\{q_5\}}})$ can be transformed to $(\{q_0\}, \overline{\overline{\{q_1, q_2, q_3, q_4, q_5\}}}$, by the recursively applying the above three merging rules.

Consequent applying of these mergings reduces the maximal number of states in the output automaton. An upper bound on the number of states in the lower part of the automaton has been calculated to be in $O((1.195n)^n)$, where n is the number of states of the input automaton [1]. This is considerably lower than the $O((1.59n)^n)$ worst-case state growth of the Fribourg construction without the M1 optimisation. A formal description and a proof of correctness of the M1 optimisation can be found in [1] (Section 4.E).

1.2.3 M2:

The M2 optimisation is the most involved of the three optimisations. Furthermore, it can only be applied together with the M1 optimisation. The main point of the M2 optimisation can be summarised as follows:

> Every state of the lower part contains *at most* one 2-coloured component.

The purpose of this restriction is to further reduce the maximal number of states the construction can generate (that is, to reduce the worst-case state complexity of the construction).

- When determining the colours of the components of a new state, the components must be processed from right to left
- The first component that according to the rules in Figure 1.5 deserves colour 2 gets colour 2 as usual
  - If this component has a sibling[2] to its left, then this sibling gets colour 2 as well
- From the point on where colour 2 has been assigned to a component (and possibly its sibling), all the further components that according to Figure 1.5 deserve colour 2 get colour 1 instead of colour 2

This leaves the new state in a situation where at most two components have colour 2, namely the first one that deserves it from the right, and its possible sibling. In the case that there was a sibling, the application of the M1 optimisation (which is necessary for the application of the M2 optimisation) merges these two components to a single 2-coloured component. This makes the state in any case having at most one 2-coloured component.

This modification of the construction requires further care to be taken when the only 2-coloured component of a state "disappears", that is, has no successor components in the state under construction. In this case, one of the 1-coloured components of this state is made accepting. In more detail, the points to consider are as follows.

- Disappearance of a 2-coloured component: uf the predecessor of a new state has a 2-coloured component, and if after the assignment of colours to the components of this new state, as described above, the new state has no 2-coloured component, then we say that the 2-coloured component of the predecesor state disappeared.
- When the disappearance of a 2-coloured component is detected, and if the state under construction contains at least one 1-coloured component, then the following is done:
  - The position where the successor component of the disappeared component *would* be is determined
  - Select the first 1-coloured component that is to the left of the first a 0-coloured component that is to the left of this position

[2] With sibling we mean having the same predecessor component.

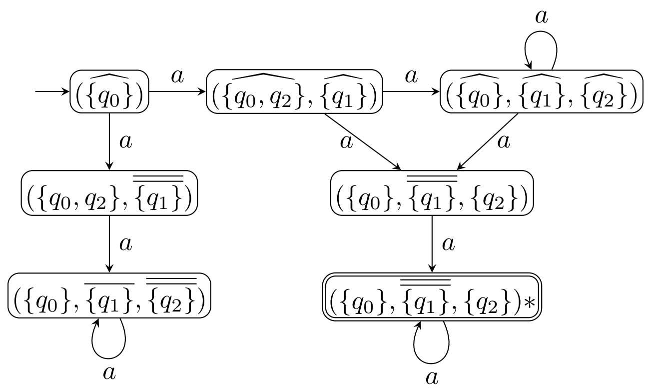


Figure 1.7: Result of applying the M2 optimisation to the complementation of the example automaton from Figure ??.

  - ∗ If the search arrives at the leftmost component of the state, then continue it starting from the rightmost component
  - ∗ If there is exactly one 1-coloured component in the state, then directly select this component
- – Change the colour of the selected component from 1 to 2
- – Mark the state with a special mark (for example a $*$)
  - ∗ This mark distinguished states. That is, if two states have the same components with the same colours, but one has a mark and the other not, then they are two different states.

The result of this procedure is that the state has a single 2-coloured component again. However, as mentioned, this state must be accepting, even though it contains a 2-coloured component. It is the purpose of the mark to distinguish this state from other states with a 2-coloured component. The last thing we have to do is to change the acceptance condition to include all states of the lower part containing no 2-coloured components, plus all states having a mark.

In this case, the difference is the marked state $(\{q_0\}, \overline{\overline{\{q_1\}}}, \{q_2\})*$. It exists, because the 2-coloured component $\overline{\overline{\{q_1\}}}$ of the predecessor state disappears. According to the colour rules, the $\overline{\{q_1\}}$ in the new state then gets colour 1. This is where the construction without the M2 optimisation would left off. With the M2 optimisation, however, we have to select one of the 1-coloured components of the new state and change its colour to colour 2. There is only one 1-coloured component in the state, $\overline{\{q_1\}}$, and so we change its colour from 1 to 2. We also mark this state with a star. The successor of this marked state happens to be identical with itself (because the component $\overline{\overline{\{q_1\}}}$ again disappears), so we add a loop. In the end, the marked state is made accepting.

In [1] (Section 4.H) a very loose upper bound on the number of states of the lower part of $O((0.86n)^n)$ is proposed. This is a significant reduction from the upper bound of $O((1.195n)^n)$ that s achieved with the M1 optimisation. Furthermore, concrete calculations on the M2 optimisation suggest that the real upper bound might be as low as $O((0.76n)^n)$. This coincides with the established lower bound for Büchi complementation by Yan [55] and would make the construction optimal in the sense of worst-case state complexity. The actual complexity of the M2 optimisation is subject to further research by the authors of the Fribourg construction.

[1] Section 4.H

1.3 General Remarks

Appendix A

Plugin Installation and Usage

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

Appendix B

Median Complement Sizes of the GOAL Test Set

Bla bla bla

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

(a) Fribourg

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

(b) Fribourg+R2C

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

(c) Fribourg+R2C+C

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

(d) Fribourg+M1

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

(e) Fribourg+M1+M2

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

(f) Fribourg+M1+R2C

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

(g) Fribourg+M1+R2C+C

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

(h) Fribourg+R

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

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

(a) Piterman+EQ+RO

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

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

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

Appendix C

Execution Times

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

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

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

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

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

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

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

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

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

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

Bibliography

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

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

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

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

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

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

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

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

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

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

[11] 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.

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

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

[14] 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.

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

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

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

[18] 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.

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

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

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] J.-P. Pecuchet. On the complementation of Büchi automata. *Theoretical Computer Science.* 47(0):pp. 95 – 98. 1986.

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

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

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

[30] 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.

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

[32] 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.

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

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

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

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

[37] 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.

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

[39] 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.

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

[41] 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.

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

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

[44] 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.

[45] 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.

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

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

[48] 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.

[49] 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.

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

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

[52] 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.

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

[54] 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.

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