Performance Investigations of a Subset-Tuple Büchi Complementation Construction

Daniel Weibel

June 9, 2015

Abstract

This will be the abstract.

Acknowledgements

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

Contents

1 Performance Investigation of the Fribourg Construction **2**
1.1 Implementation of the Construction . . . 3
1.1.1 GOAL . . . 3
1.1.2 The GOAL Plugin . . . 5
1.1.3 Options for the Fribourg Construction . . . 5
1.1.4 Verification of the Implementation . . . 7
1.2 Test Data . . . 8
1.2.1 GOAL Test Set . . . 8
1.2.2 Michel Automata . . . 8
1.3 Experimental Setup . . . 10
1.3.1 Internal Tests . . . 10
1.3.2 External Tests . . . 11
1.3.3 Time and Memory Limits . . . 12
1.3.4 Execution Environment . . . 12

2 Results and Discussion **14**
2.1 Internal Tests . . . 15
2.1.1 GOAL Test Set . . . 15
2.1.2 Michel Automata . . . 17
2.2 External Tests . . . 17
2.2.1 GOAL Test Set . . . 17
2.2.2 Michel Automata . . . 18

Chapter 1

Performance Investigation of the Fribourg Construction

Contents

1.1 Implementation of the Construction 3
1.1.1 GOAL 3
1.1.2 The GOAL Plugin 5
1.1.3 Options for the Fribourg Construction 5
1.1.4 Verification of the Implementation 7
1.2 Test Data 8
1.2.1 GOAL Test Set 8
1.2.2 Michel Automata 8
1.3 Experimental Setup 10
1.3.1 Internal Tests 10
1.3.2 External Tests 11
1.3.3 Time and Memory Limits 12
1.3.4 Execution Environment 12

Contents

1.1 Implementation of the Construction 3
1.1.1 GOAL 3
1.1.2 The GOAL Plugin 5
1.1.3 Options for the Fribourg Construction 5
1.1.4 Verification of the Implementation 7
1.2 Test Data 8
1.2.1 GOAL Test Set 8
1.2.2 Michel Automata 8
1.3 Experimental Setup 10
1.3.1 Internal Tests 10
1.3.2 External Tests 11
1.3.3 Time and Memory Limits 12
1.3.4 Execution Environment 12

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

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

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

1.1 Implementation of the Construction

1.1.1 GOAL

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

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

All this is accessible by both, a graphical and a command line interface. The graphical interface is shown in Figure 1.1. Automata are displayed in the main editor window of the GUI. They can be freely edited, such as adding or removing states and transitions, and arranging the layout. There are also various layout algorithms for automatically laying out large automata. Most of the functionality provided by the graphical interface is also accessible via a command line mode. This makes it suitable for automating the execution of operations.

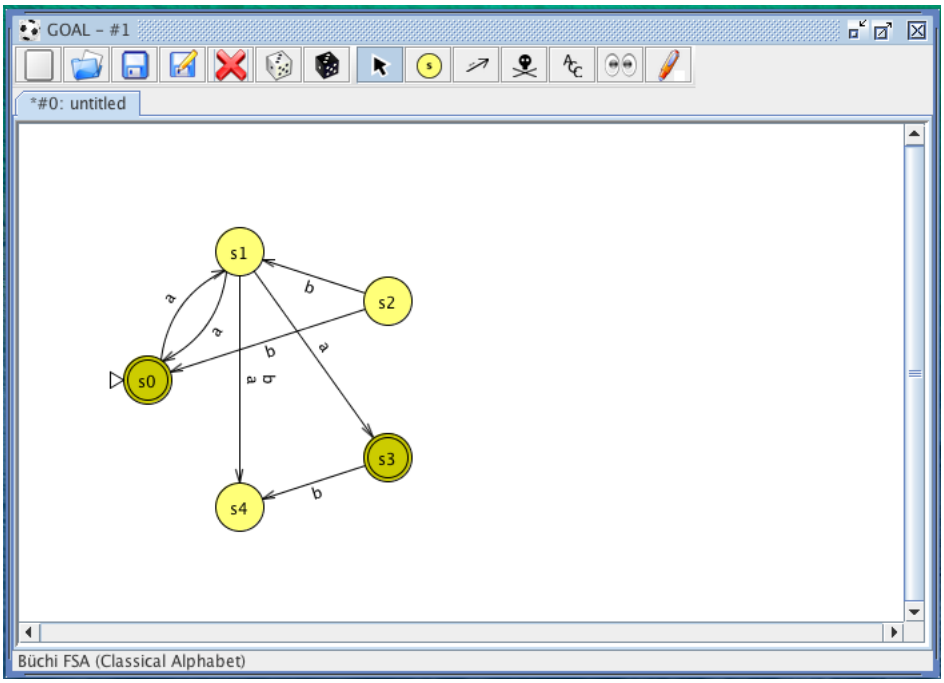


Figure 1.1: Graphical interface of GOAL.

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

An important design concept of GOAL is modularity. GOAL uses the Java Plugin Framework (JPF) [1], a library for building modular and extensible Java applications. A JPF application defines so-called extension points for which extensions are provided. These extensions contain the actual functionality of the application. Extensions and extension points are bundled in plugins, the main building block of a JPF application. It is therefore possible to extend an existing JPF application by bundling a couple of new extensions for existing extensions points in a new plugin, and installing this plugin into the existing application. On the next start of the application, the new functionality will be included, all without requiring to recompile the existing application or to even have its source code.

GOAL provides a couple of extensions points, such as *Codec*, *Layout*, or *Complementation Construction*. An extension for *Codec*, for example, allows to add the handling of a new file format which GOAL can read from and write to. With an extension for *Layout* one can add a new layout algorithm for laying out automata in the graphical interface. And an extension to Complementation Construction allows to add a new complementation construction to GOAL. This is how we added the Fribourg construction to GOAL, as we will further explain in Section 1.1.2.

There are a couple of Büchi complementation constructions pre-implemented in GOAL. Table 1.1 summarises them, showing for each one its name on the graphical interface and in the command line mode, and the reference to the paper introducing it. As can be seen, the most important representants of all the four approaches (Ramsey-based, determinisation-based, rank-based, and slice-based, see Chapter **??**) are present. In addition to the listed constructions, GOAL also contains Kurshan's construction and classic complementation. These are for complementing DBW and NFA/DFA, respectively, and thus not relevant to us.

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

Complementation constructions in GOAL can define a set of options that can be set by the user. In the graphical interface this is done at the start of the operations via a dialog window, in the command line

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

Table 1.1: The complementation constructions implemented in GOAL (version 2014-11-17).

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

mode the options are specified as command line arguments. Figure 1.2 shows the options dialog of the Safra-Piterman construction. Complementation options allow to play with different configurations and variants of a construction, and we will make use of them for including the optimisations presented in Chapter **??** to our implementation of the Fribourg construction.

For most complementation constructions (all listed in Table 1.1 except the Ramsey-based construction) there is also a version for step-by-step execution. In this case, the constructions define so-called steps and stages, through which the user can iterate independently. This is a great way for understanding how a complementation construction works, and for investigating specific cases in order to potentially further improve the construction.

1.1.2 The GOAL Plugin

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

1.1.3 Options for the Fribourg Construction

To keep the Fribourg construction flexible, we made use of options. The three optimisations described in Section **??** are presented to the user as selectable options. Additionally, we included several further options. Table 1.2 lists them all. For convenience, we use for each options a short code name, which is also used as the option name in the command line mode.

The first three items in Table 1.2, m1, m2, and r2c, correspond to the optimisations M1, M2, and R2C, described in Section **??**. As the M2 optimisation requires M1, our implementation makes sure that the m2 option can only be selected if also the m1 option is selected. The `c` option, for making the input automaton complete before starting the actual construction, is intended to be used with the `r2c` option. In this way, the R2C optimisation can be forced to apply. This idea results from previous work that investigated whether making the input automaton complete plus the application of the R2C optimisation brings an improvement over the bare Fribourg constructoin [**?**]. The result was negative, that is, the construction performs worse with this variant on practical cases. Also note that using the `c` option alone, very likely decreases the performance of the construction, because the automaton is made bigger if it is not complete.

The `macc` and `r` options are common among the other complementation constructions in GOAL. The first one, `macc`, maximises the accepting set of the input automaton. That means, it makes as many states accepting as possible without changing the automaton's language. This should help to make the

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

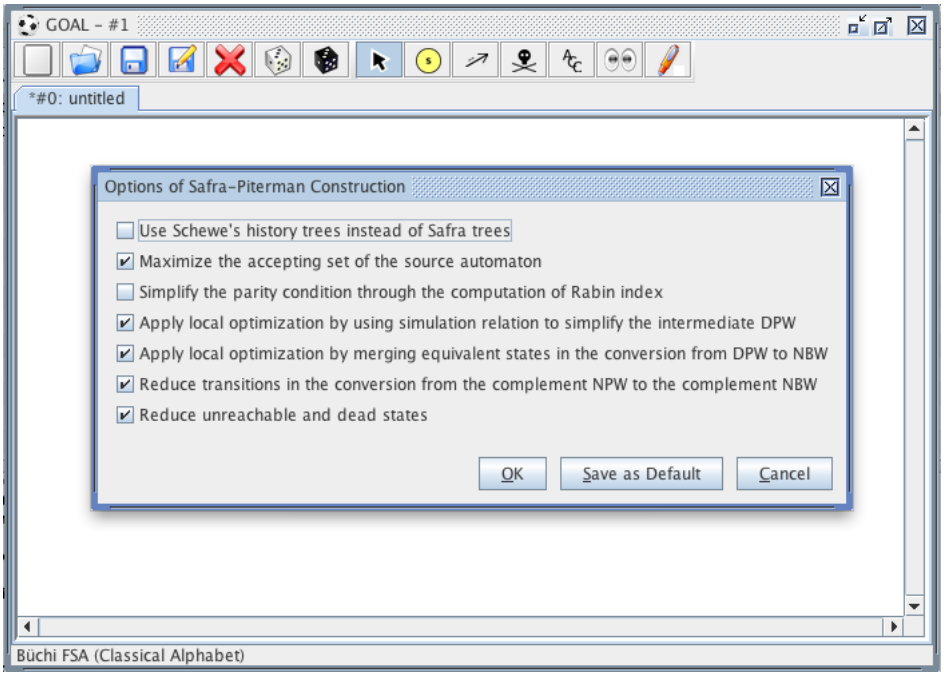


Figure 1.2: Complementation constructions in GOAL can have a set user-selectable options. Here the options of the Safra-Piterman construction.

Table 1.2: The options for the Fribourg construction.

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

complement automaton smaller. The r options prunes unreachable and dead states from the complement automaton. Unreachable states are states that cannot be reached from the initial states, and dead states are states from where no accepting state can be reached. Clearly, all the runs containing an unreachable or dead state are not accepting, and thus these states can be removed from the automaton without changing its language. The complement automaton can in this way be made smaller. The rr option in turn removes the unreachable and dead states from the input automaton. That is, it makes the input automaton smaller, before the actual construction starts, what theoretically results in smaller complement automaton.

Finally, the b option affects just the display of the state labels of the complement automaton. It uses an alternative notation which uses different kinds of brackets, instead of the explicit colour number, to indicate the colours of sets. In particular, 2-coloured sets are indicated by square brackets, 1-coloured sets by round parenthesis, and 0-coloured sets by curly braces. Sets of states of the upper part of the automaton are enclosed by circumflexes. This notation, although being very informal, has proven to be very convenient during the development of the construction.

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

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

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

1.1.4 Verification of the Implementation

See UBELIX/jobs/2014-11-25

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

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

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

Doubtlessly, it would be desirable to test more, and especially bigger and more diverse automata. How-

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

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

1.2 Test Data

1.2.1 GOAL Test Set

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

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

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

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

We tested each of the 11,000 automata for completeness and universality. As GOAL provides no commands for testing completeness and universality, we created the additional GOAL plugin ch.unifr.goal.util that implements these two operations and makes them accessible over the command line interface. The results of our tests are the following.

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

We furthermore analysed how these two properties are distributed over the 110 classes of transition/acceptance density combinations. The results follow below in table form and as a three-dimensional visualisation.

1.2.2 Michel Automata

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

- Michel N1: 3 states, 7 transitions, 1 accepting state

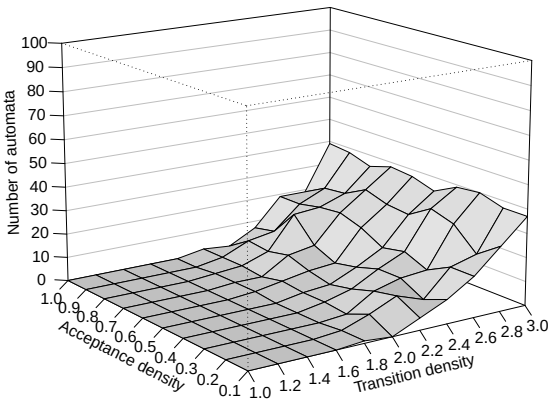
[4] http://goal.im.ntu.edu.tw/

0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	1	1	1	0	0
0	5	1	2	2	3	1	2	2	2
5	10	8	5	3	5	8	6	7	1
10	6	11	11	8	6	10	20	9	7
17	17	12	16	14	19	22	21	19	19
27	20	29	32	26	27	30	25	24	19
37	37	40	39	34	37	38	35	38	39

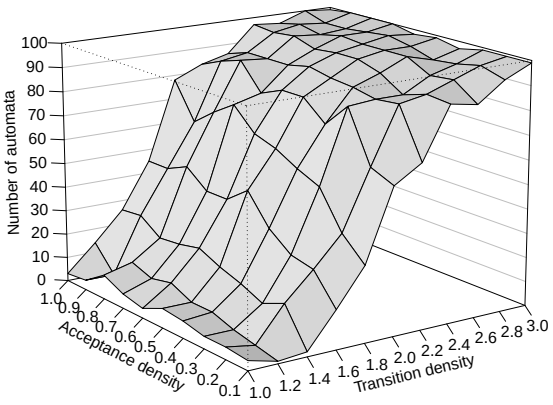
(a) Completeness

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

(b) Universality



(c) Completeness



(d) Universality

Figure 1.3: Completeness and universality in the GOAL test set.

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

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

1.3 Experimental Setup

1.3.1 Internal Tests

In the internal tests our aim is to compare different versions of the Fribourg construction. A specific version of the Fribourg construction is composed of a combination of options. Options can be the

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

- R2C: if the input automaton is complete, remove all states whose rightmost colour is 2
- M1: merge certain adjacent sets within a state
- M2: reduce 2-coloured sets (requires M1)

Furthermore, our GOAL plugin includes, among others, the following options:

- C: make the input automaton complete (by adding a sink state)
- R: remove unreachable and dead states from the output automaton

The versions of the Fribourg construction that we chose for our internal tests consist of combinations of these five options. According to the nature of the two test sets (the GOAL test set and the Michel automata), we chose two different sets of versions for the two test sets. We are now first going to describe the setup for the GOAL test set and then the one for the Michel test set.

GOAL Test Set

For the tests on the GOAL test set, we chose the following eight versions of the Fribourg construction:

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

The first version is the plain Fribourg construction without any optimisations or options. The next two versions are devoted to investigate the R2C optimisation. Version 2 applies the optimsation only to the automata which happen to be complete (and as we have seen in Section 1.2.1 these are 9%). Version 3, on the other hand, makes all input automata preliminarily complete by adding a sink state, so that the R2C optimisation can be applied to *all* automata. The question here is, does it pay off to increase the size of the automata by one (for adding the sink state) but then being able to apply R2C, or is it better to not add the extra state but then not applying R2C neither?

Similar investigations about the R2C optimisation of the Fribourg construction have been made by Göttel [**?**]. In particular, he compared Version 1 in our above listing with Version 3. His results were that the mean complement sizes of Version 3 are higher than in Version 1. He evaluated, however, only the mean values, but looking closely at the results suggests that the median values could be in favour of Version 3. Therefore, we decided to reinvestigate this question. Indeed, in our own results the median complement sizes of Version 3 are considerably lower than the ones of Version 1 and Version 2. We will further elaborate on this point in Chapter 2.

Versions 4 and 5 in our above listing are for investigating the M1 and M2 optimisations. As M2 requires M1, there are only these two possible combinations. Versions 6 and 7 then enhance the "better" one of Version 4 and 5 with R2C and its alternative R2C+C. As we will see in Chapter 2, the better one of

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

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

Michel Test Set

The versions we tested for the Michel test set are the following:

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

The rationale for choosing these versions is basically the same as for the GOAL test set with the following differences. First, Michel automata are complete, thus it is not necessary to apply the C option. Rather, R2C will automatically apply to all automata. Second, the aim of Version 5 is again to enhance the better one of Versions 3 and 4 with R2C. However, contrarily to the GOAL test set Fribourg+M1+M2 is better than Fribourg+M1 for the Michel automata. That is why Version 5 is Fribourg+M1+M2+R2C.

1.3.2 External Tests

In the so called external tests we compare the best version of the Fribourg construction with different complementation constructions, again for both the GOAL test set and the Michel test set. The concrete constructions we compared for the external tests are the following.

1. Piterman+EQ+RO
2. Slice+P+RO+MADJ+EG
3. Rank+TR+RO
4. Fribourg+M1+R2C (GOAL test set) and Fribourg+M1+M2+R2C (Michel test set)

For the alternative constructions, we chose the Piterman, Slice, and Rank construction. These constructions are representative for three of the four main complementation approaches, determinization-based, slice-based, and rank-based. The fourth complementation approach is Ramsey-based and there is an implementation of the Ramsey construction in GOAL. However, in preliminary tests, we realised that this construction is not performant enough to be used on our test sets within our time and memory constraints. The authors of GOAL came to a similar conclusion when they made similar experiments on the GOAL test set [**?**]. In their case, the Ramsey construction could not complete any of the 11,000 automata within the set time and memory constraints, and they went on to do the result analysis without the Ramysey construction.

The same applies to all the other constructions that are implemented in GOAL. We did preliminary tests with all of them and saw that only the three mentioned constructions, Piterman, Slice, and Rank, can be resonably used on the GOAL test set[5].

The three chosen constructions also have optimisations in their implementations in GOAL. To have a fair comparison to the best version of the Fribourg construction, which also uses optimisations, we activated the optimisation of these constructions as well. In particular, we chose those optimisations which are set as default in the GOAL GUI for each construction.

We use Fribourg+M1+R2C for the GOAL test set and Fribourg+M1+M2+R2C for the Michel test set. This is because these two versions are the most performant ones for the respective test sets.

[5]The Safra construction would also have been possible, but the Safra construction is similar to the Piterman construction.

1.3.3 Time and Memory Limits

We defined a time limit of 600 seconds CPU time and a memory limit of 1 GB per complementation task in the GOAL test set. That means, if the complementation of a single automaton is not finished after 600 seconds CPU time or uses more than 1 GB memory, the task is aborted and marked as a *timeout* or *memory excess*.

These limits correspond to the ones used by the experiments of the GOAL authors [**?**]. However, from their paper it is not clear if their time limit is in CPU time or wallclock time. But since they used different computing nodes, our results regarding the number of timeouts will anyway differ from theirs.

The reason for these limits is simply the restricted amount of time and memory resources that we have available for the experiments. In an ideal world, we would let every complementation task run to its end, no matter how long it takes and how much memory it uses. This would give a perfectly unbiased picture of the results. By setting time and memory limits, we basically exclude the most difficult automata from the experiment. However, as mentioned, the practical reasons of limited time and computing power force as to make this compromise.

The timeout and memory limit of 1 GB apply just to the automata of the GOAL test set. For the Michel test set we did not set a timeout because we wanted each one of the four automata to finish. The longest complementation task of a Michel automaton consequently durated 109,810 seconds which is about 28 hours. We set a very high memory limit of 14 GB for the Michel test set to avoid memory excesses as we wanted each automaton to successfully complete. The number of 14 GB is determined by the physically available memory on the used computing nodes. All Michel automata successfully completed with this amount of memory.

The timeout was implemented by the means of the ulimit Bash builtin[6], which allows to set a maximum time after which running processes are killed. The memory limit was implemented by setting the maximum size of the Java heap, which can be done by the -Xmx option to the Java Virtual Machine (JVM). The heap is the main memory area of Java and the place where all the objects reside. Note that since our memory limit defines actually the size of the Java heap, the total amount of memory used by the process is higher than our limit, as Java has some other memory areas, for example for the JVM itself. However, this is a rather constant amount of memory and independent from the current automaton, so it does not disturb the relative comparisons of the results.

The presence of aborted complementation task require the consideration of the so called effective samples in the result analysis, as introduced in the experiment paper of the GOAL authors. The effective samples are those automata which have been successfully completed by *all* constructions that are to be compared to each other. Imagine two constructions A and B where A is successful complementing all the automata, whereas B has timeouts or memory excesses at 100 of the automata. If we would now take, for example, the median complement sizes of the two result sets without first extracting the effective samples, then B is likely be assessed as too good relative to A, because B's results do not include the 100 automata at which it failed, and which are thus likely to have large complement sizes with B. The same 100 automata would however be included in the results of A. Therefore, all the result analysis of the experiments with the GOAL test sets, that we present in Chapter 2, are based on the effective samples of the result sets.

1.3.4 Execution Environment

We executed the experiments on a high performance computing (HPC) computer cluster called UBELIX at the University of Bern[7]. This cluster consists of different types of Linux-driven HPC computing nodes, and is managed by Oracle Grid Engine[8] (formerly known as Sun Grid Engine) version 6.2. Oracle Grid Engine is a so called load scheduler that is responsible for automatically distributing computing tasks to computing nodes.

The basic workflow of working on the cluster is to prepare a so called job and specify the resources that the job requires for running (time, memory, number of CPU cores, and so on). Then the job can be submitted to the grid engine. The grid engine first puts incoming jobs in a queue and then automatically dispatches them to suitable computing nodes as soon as the required capacity is available.

[6] http://linux.die.net/man/1/bash
[7] http://ubelix.unibe.ch
[8] http://www.oracle.com/us/products/tools/oracle-grid-engine-075549.html

A job in our case is the execution of a construction (or version of the Fribourg construction) over an entire test set. Thus, we were running 22 jobs in total. We arranged that all jobs were run on identical nodes with the following specifications:

- Processor: Intel Xeon E5-2665 2.40GHz (64 bit)
- CPU cores: 16
- Operating System: Red Hat Enterprise Linux 6.6

Since we use the execution time (CPU time) as a secondary metrics, next to the complement sizes, it is important that all complementation tasks are executed on the same type of hardware. GOAL is multithreaded and thus the jobs may use multiple CPU cores. The number of CPU cores a job may use is not restricted (up to the number of available cores on the node, in our case 16), but our observation is that the jobs use a rather small number of cores (2–3). Note that the measurement of the execution time is not affected by the number of cores a process uses, as the CPU times are measured separately on each core and then added together.

Chapter 2

Results and Discussion

Contents

2.1 Internal Tests 15
2.1.1 GOAL Test Set 15
2.1.2 Michel Automata 17
2.2 External Tests 17
2.2.1 GOAL Test Set 17
2.2.2 Michel Automata 18

2.1 Internal Tests

2.1.1 GOAL Test Set

First of all, let us how many unsuccessful complementation tasks there were with the internal tests on the GOAL test set. Table 2.1 shows the number of timeouts and memory excesses for each of the eight tested versions of the Fribourg construction.

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

Table 2.1: Number of timeouts and memory excesses in the internal tests on the GOAL test set.

As we can see in the table, there were no memory excesses, that is, none of the 11,000 complementation tasks required more than 1 GB memory (actually, Java heap size). On the other hand, there is quite a number of timeouts. If we extract the effective samples from these result sets, we get a set of 10,939 automata. That means that these 10,939 automata have been successfully completed by all the versions of the Fribourg construction, whereas the remaining 61 automata (0.55%) provoked a timeout in at least one of the versions.

Our main analysis is now about the sizes of the complements of these 10,939 automata. With size we mean the number of states of an automaton. A stripchart as the one in Figure 2.1 is a good way to get a first glance of the complement sizes. Each horizontal strip contains 10,939 dots, each one corresponding to a produced complement automaton. The x-position of a dot indicates the size of the corresponding complement automaton.

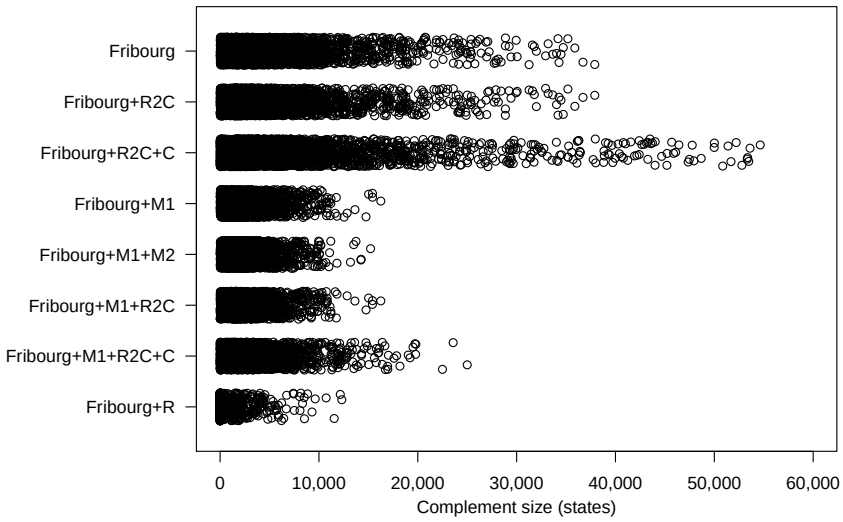


Figure 2.1: Stripchart with the complement sizes of the 10,939 effective samples of the GOAL test set.

The first thing to note is that the distribution of complement sizes is extremely right-skewed (also known as positive-skewed). The peak close to the left end of the x-axis and there is a long tail toward the right. This means that most of the complements are very small and the frequency decreases with increasing complement size. Finally, there are some few complements which are very large. This distribution implies

that the mean is generally higher than the median, because the mean is "dragged" to the right by the few very large complements.

Next, we can compare the distributions of the individual version to each other. While Fribourg and Fribourg+R2C have a similarly long tail, Fribourg+R2C+C has a considerably longer one. This clearly is the effect of increasing the size of 91% of the automata by adding a sink state in order to make them complete. Next, Fribourg+M1, Fribourg+M1+M2, and Fribourg+M1+R2C have similarly long tails that are significantly shorter than the ones of the previous versions. This indicates that the M1 optimisation is effective in reducing the size of otherwise large complements. Fribourg+M1+R2C+C, as expected, again increases the size of the tail due to the C option. Finally, Fribourg+R has a very short tail and an even higher concentration of very small complements. This is because the complements are pruned by removing their unreachable and dead states, and thus at least the complements of the 61.8% universal input automata are reduced to empty automata of size 1.

Having a first impression of the distribution of complement sizes, we can look at some statistics characterising this distribution. Table 2.2 shows the mean complement size along with the classic five-number summary consisting of minimum value, 25th percentile, median, 75th percentile and maximum value for each version of the Fribourg construction. As expected, the mean is always higher than the median. Generally, the median is a more robust metrics than the mean, because it is not affected by the value of outliers. This applies specifically to our distribution where some few very large complements might significantly increase the man whereas they leave the median unaffected. Therefore, the median will be our main metric, and the subsequent analyses will be based on the median.

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

Table 2.2: Statistics of the complement sizes of the 10,939 effective samples of the GOAL test set.

Going through the median values in Table 2.2 we encounter some surprises. To begin with, as expected, there is a decrease from Fribourg (761) to Fribourg+R2C (689). Then, however, there is a significant drop to 451 with Fribourg+R2C+C. This is a surprise insofar as by looking at Figure 2.1, Fribourg+R2C+C seems to have the worst performance at a first glance. Indeed it also has the highest mean, which is due to the group of extremely large complements. The median, however, is very low, even lower than the one of Fribourg+M1 with its significantly shorter tail in Figure 2.1. Also the 25th percentile of Fribourg+R2C+C is with 85 one of the lowest. Going to the other side of the median, however, the 75th percentile (2,329) is the largest of all versions. A possible characterisation of this phenomenon is that the C option (together with R2C) makes small complements smaller, and large complements larger. The diminishment of small complements is far-reaching enough that the median is affected by it and decreased significantly.

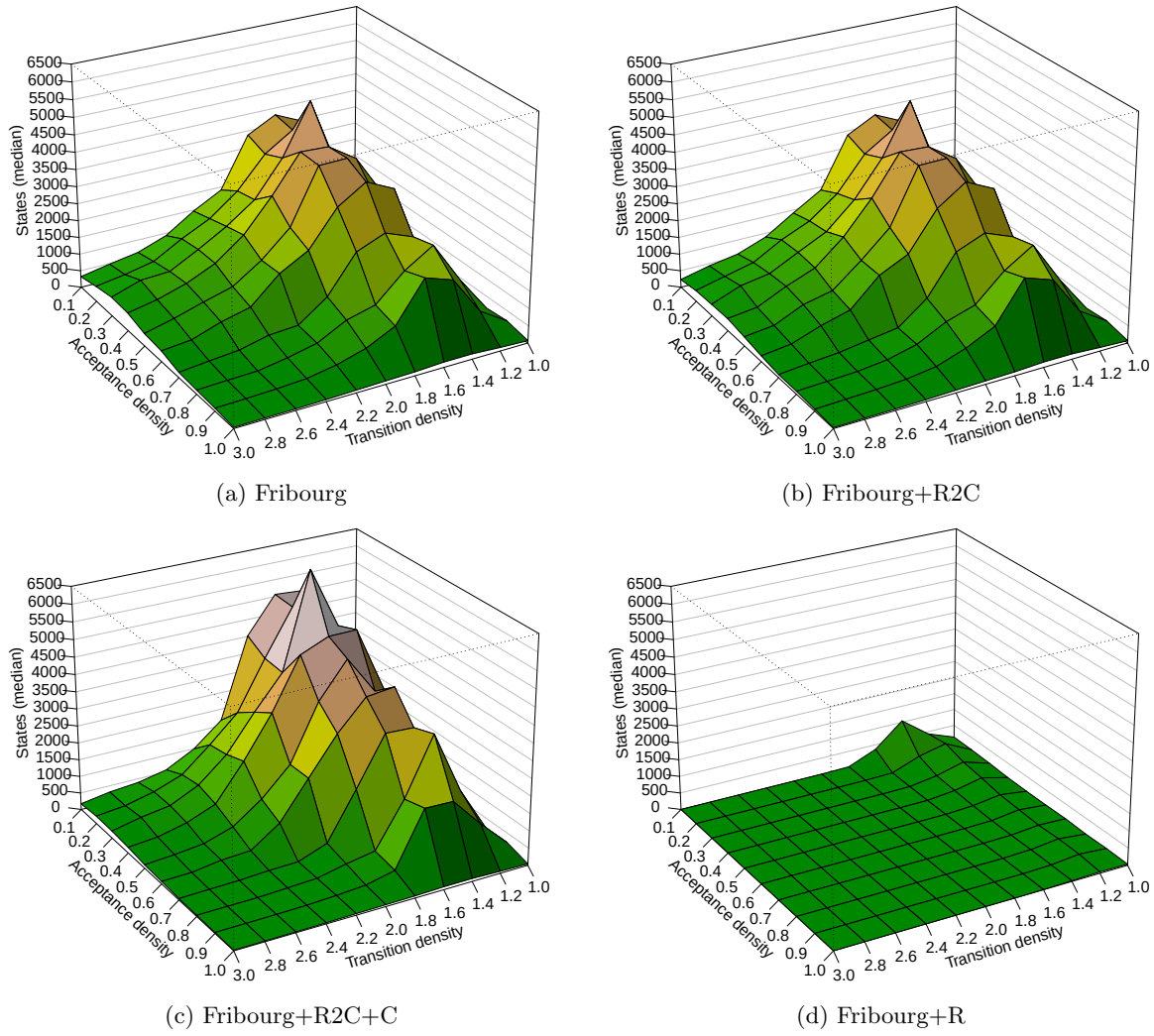
The next thing we see in Table 2.2 is that the median of Fribourg+M1 (482) is slightly lower than the median of Fribourg+M1+M2 (496). The same applies to the 25th and 75th percentile. This backs up our statement from Section **??** that Fribourg+M1 performs better on the GOAL teset set than Fribourg+M1+M2. The difference is rather small (the median increase is 2.9%), but it is enough to consider Fribourg+M1 as the better of the two versions, and to combine it therefore with R2C and R2C+C.

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

complements are made larger.

Finally, the last row in Table 2.2 with Fribourg+R shows the extent of unreachable and dead states that the Fribourg construction produces. The median is 1, and a further analysis reveals that the single-state complements go up to the 61st percentile. That is, 61% of the complements have a size of 1. This is likely to correspond to the 61.8% of universal automata in the GOAL test set whose complements can be reduced to empty automata with a single state.

Statistics Split Up by dt/da classes



(a) Fribourg

(b) Fribourg+R2C

(c) Fribourg+R2C+C

(d) Fribourg+R

Figure 2.2: Median states

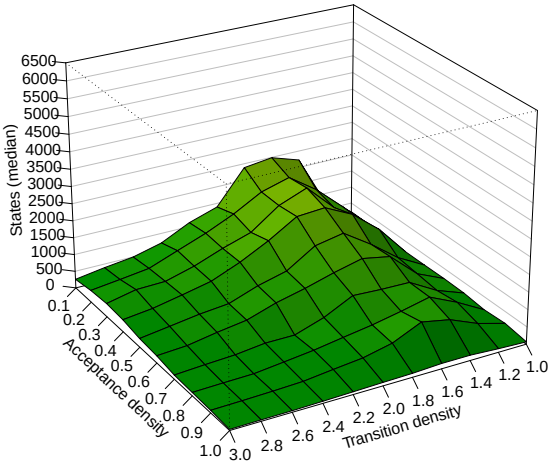
Difficulty Classes of the GOAL Test Set

2.1.2 Michel Automata

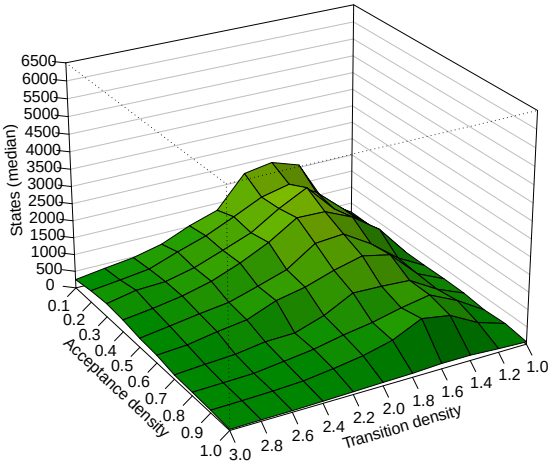
Michel automata are complete, thus the C options makes no sense.

For the Michel automata Fribourg+M1+M2 is better than Fribourg+M1. Fribourg+M1+R2C is again much better than Fribourg+M1 and even Fribourg+M1+M2. Thus, we have to test Fribourg+M1+M2+R2C.

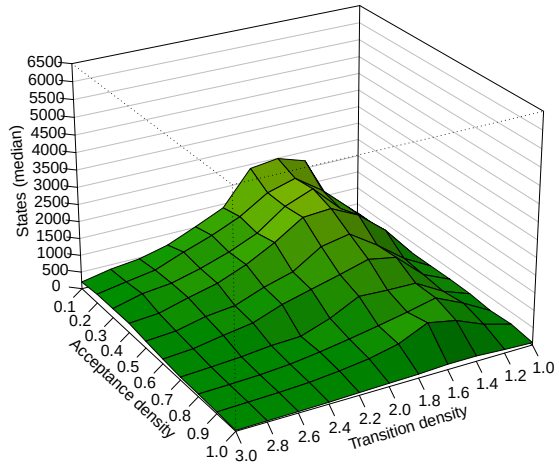
- Add Fribourg+M1+M2+R2C
- Add Fribourg+M1+M2+R2C+R
- Add Fribourg+R



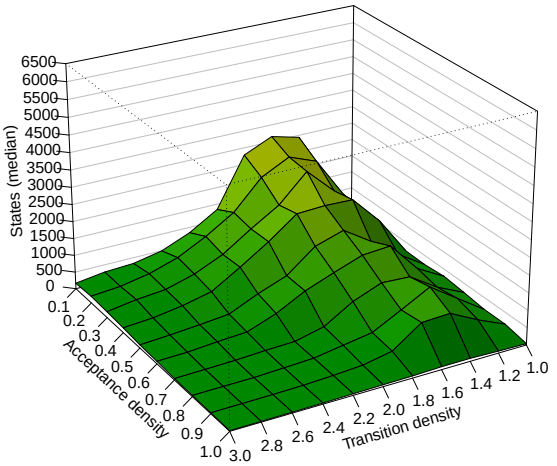
(a) Fribourg+M1



(b) Fribourg+M1+M2



(c) Fribourg+M1+R2C



(d) Fribourg+M1+R2C+C

2.2 External Tests

2.2.1 GOAL Test Set

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

Table 2.3: Number of timeouts and memory excesses.

- With Rank
  - 7,204 effective samples (3,796 uncompleted tasks, 34.51%)
- Without Rank
  - 10,998 effective samples (2 uncompleted tasks, 0.02%)

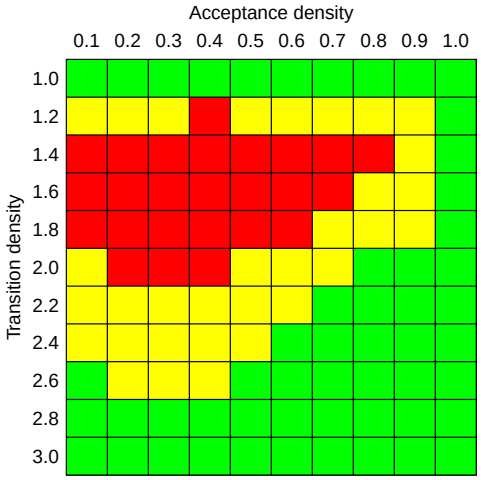


Figure 2.4: Average of the median number of produced states of all the constructions: green (easy): $\leq$ 500; yellow (medium): $\leq$ 1450; red (hard): $>$ 1450.

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

Table 2.4: Aggregated statistics of complement sizes of the 7,204 effective samples.

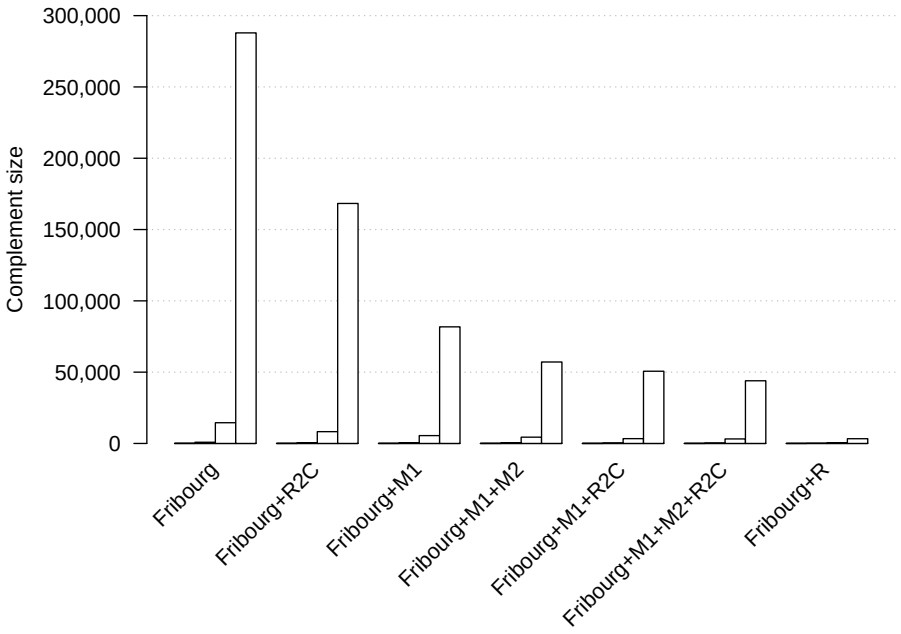
With Rank

Without Rank

2.2.2 Michel Automata

Construction	Michel 1	Michel 2	Michel 3	Michel 4
Fribourg	57	843	14,535	287,907
Fribourg+R2C	33	467	8,271	168,291
Fribourg+M1	44	448	5,506	81,765
Fribourg+M1+M2	42	402	4,404	57,116
Fribourg+M1+R2C	28	275	3,369	50,649
Fribourg+M1+M2+R2C	28	269	3,168	43,957
Fribourg+R	18	95	528	3,315

(a) Complement sizes of the first four Michel automata.



(b) Complement sizes of the first four Michel visualised as a barplot. The bars 1 to 4 of each group correspond to Michel automata 1 to 4.

Figure 2.5: Complement sizes of Michel automata.

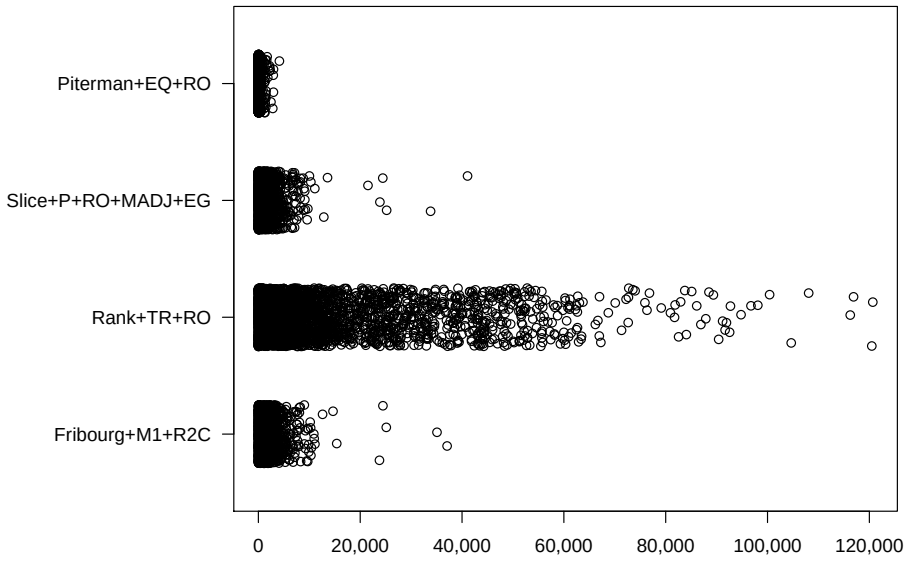


Figure 2.6: Complement sizes of the 7,204 effective samples.

Construction	Mean	Min.	P25	Median	P75	Max.
Piterman+EQ+RO	2.97	2.22	2.58	2.78	3.03	42.93
Slice+P+RO+MADJ+EG	3.66	2.21	2.69	3.22	4.07	36.67
Rank+TR+RO	16.04	2.28	2.76	3.71	9.31	443.33
Fribourg+M1+R2C	4.02	2.22	2.69	3.10	4.37	410.37

Table 2.5: Aggregated statistics of the running times of the 7,204 effective samples.

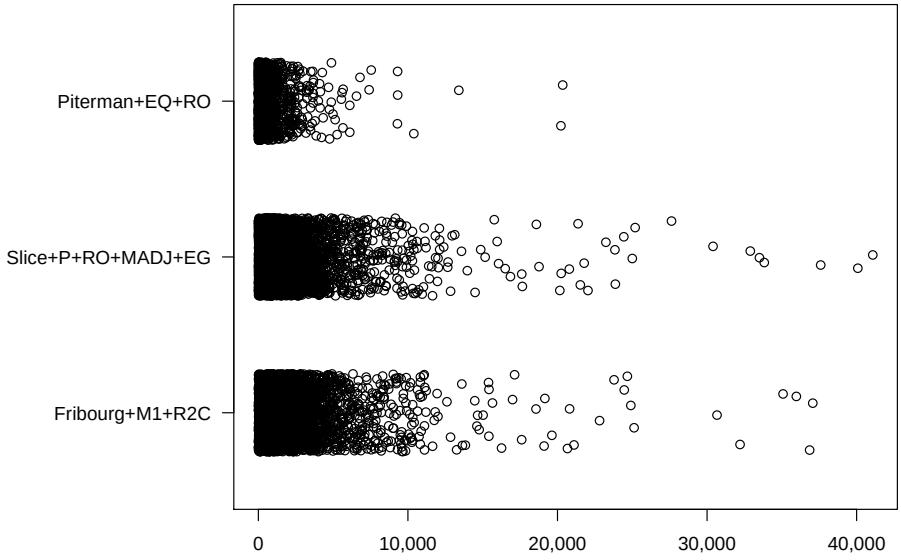


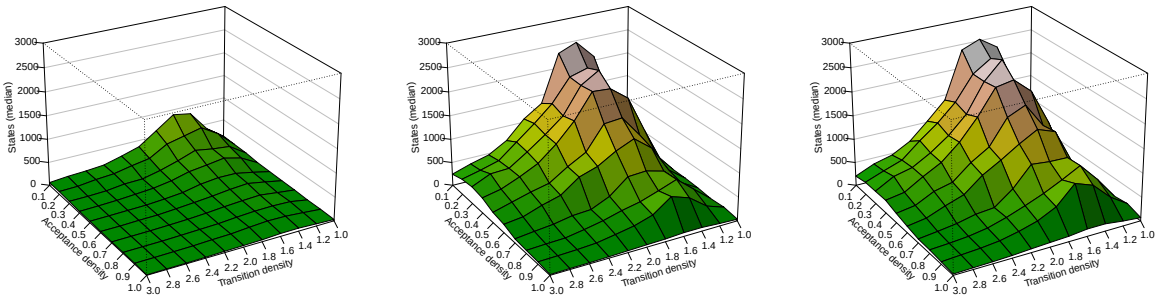
Figure 2.7: Complement sizes of the 10,998 effective samples.

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

Table 2.6: Aggregated statistics of complement sizes of the 10,998 effective samples without Rank.

Construction	Mean	Min.	P25	Median	P75	Max.
Piterman+EQ+RO	3.61	2.22	2.66	2.90	3.38	365.68
Slice+P+RO+MADJ+EG	4.31	2.21	2.94	3.73	5.00	42.37
Fribourg+M1+R2C	4.74	2.22	2.83	3.61	5.31	410.37

Table 2.7: Aggregated statistics of the running times of the 10,998 effective samples without Rank.

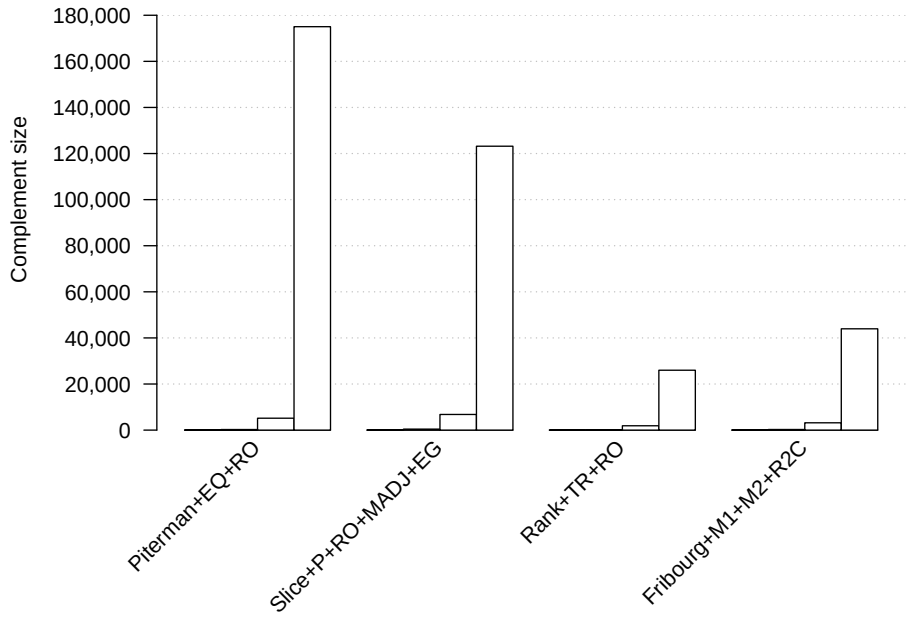


(a) Piterman+EQ+RO (b) Slice+P+RO+MADJ+EG (c) Fribourg+M1+R2C

Figure 2.8: Median complement sizes.

Construction	Michel 1	Michel 2	Michel 3	Michel 4
Piterman+EQ+RO	23	251	5,167	175,041
Slice+P+RO+MADJ+EG	35	431	6,786	123,180
Rank+TR+RO	23	181	1,884	25,985
Fribourg+M1+M2+R2C	28	269	3,168	43,957

(a) Complement sizes of the first four Michel automata.



(b) Complement sizes of the first four Michel visualised as a barplot. The bars 1 to 4 of each group correspond to Michel automata 1 to 4.

Figure 2.9: Complement sizes of Michel automata.

Construction	Michel 1	Michel 2	Michel 3	Michel 4
Piterman+EQ+RO	23	251	5,167	175,041
Slice+P+RO+MADJ+EG	35	431	6,786	123,180
Rank+TR+RO	23	181	1,884	25,985
Fribourg+M1+M2+R2C	28	269	3,168	43,957

Table 2.8: Execution times for the first four Michel automata.

Bibliography

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

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

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

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

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

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

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

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

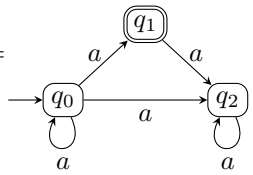
[9] W. Thomas. Complementation of Büchi Automata Revisited. In J. Karhumäki, H. Maurer, G. Păun, et al, eds., Jewels are Forever. pp. 109–120. Springer Berlin Heidelberg. 1999.

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

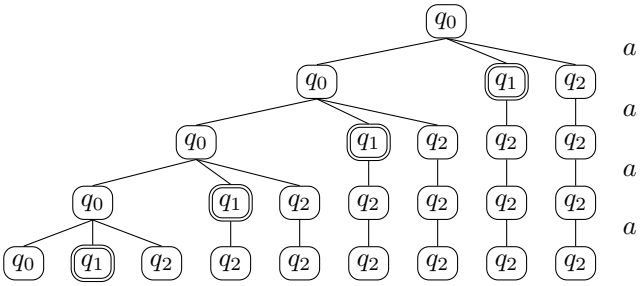
[11] Y.-K. Tsay, Y.-F. Chen, M.-H. Tsai, et al. Goal Extended: Towards a Research Tool for Omega Automata and Temporal Logic. In C. Ramakrishnan, J. Rehof, eds., Tools and Algorithms for the Construction and Analysis of Systems. vol. 4963 of Lecture Notes in Computer Science. pp. 346–350. Springer Berlin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

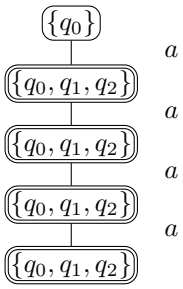
[12] M. Y. Vardi, T. Wilke. Automata: From Logics to Algorithms. In J. Flum, E. Grädel, T. Wilke, eds., Logic and Automata: History and Perspectives. vol. 2 of Texts in Logic and Games. pp. 629–736. Amsterdam University Press. 2007.

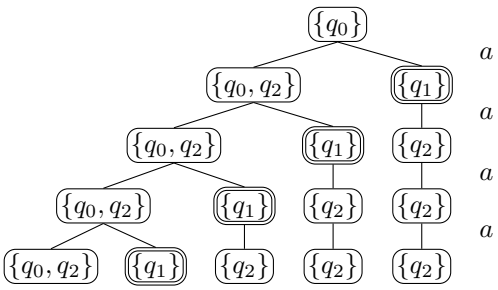
$A =$



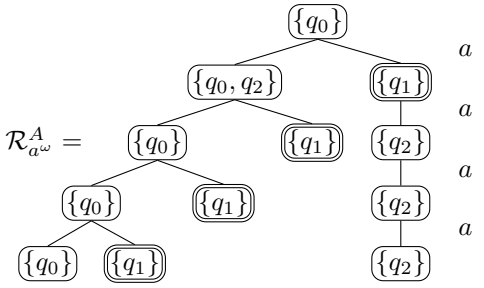
$\mathcal{T}^A_{a^\omega} =$







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



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