Performance Investigations of a Subset-Tuple Büchi Complementation Construction

Daniel Weibel

June 4, 2015

Abstract

This will be the abstract.

Acknowledgements

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

Contents

Chapter 1

Performance Investigation of the Fribourg Construction

Contents

Contents

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

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

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

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.

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],

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

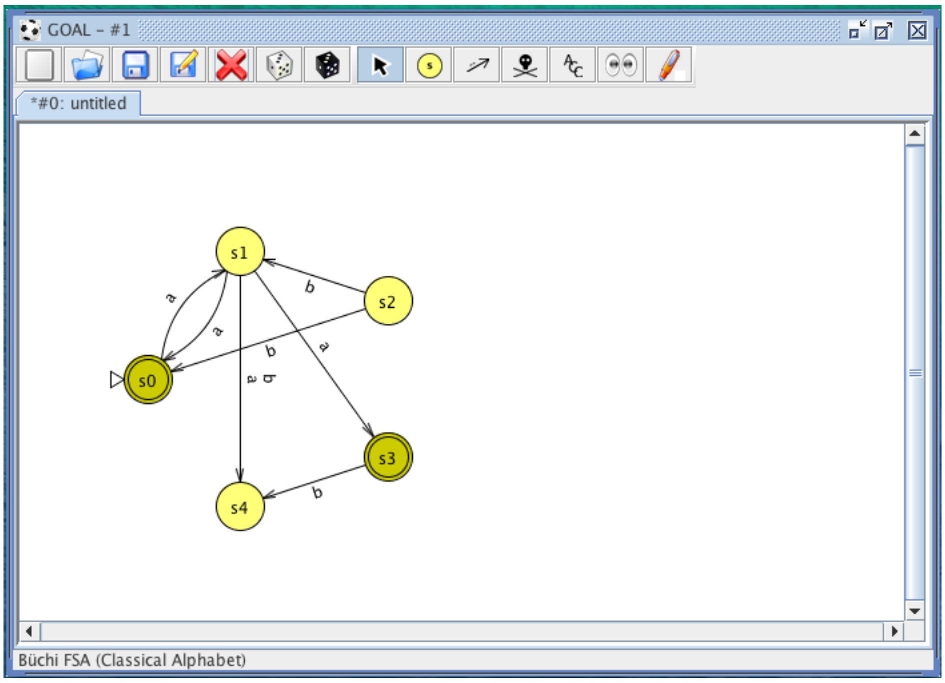


Figure 1.1: Graphical interface of GOAL.

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 1.2.

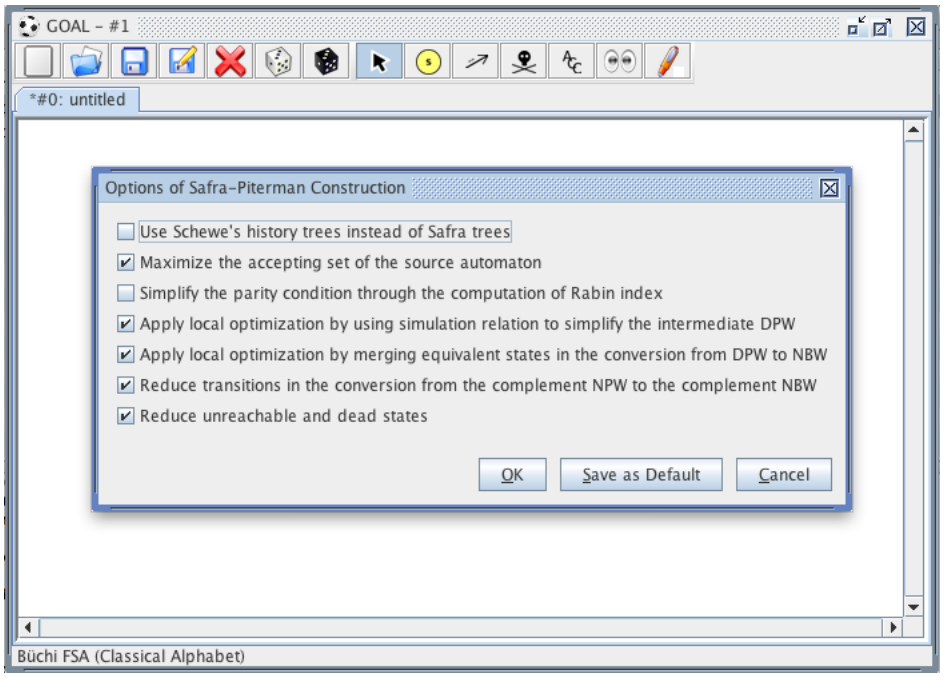
There are a couple of Büchi complementation constructions pre-implemented in GOAL. Table 1.1 summarises them, showing for each one its name on the graphical interface and in the command line mode, and the reference to the paper introducing it. As can be seen, the most important representants of all the 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 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.

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]


GOAL - #1
*#0: untitled
Options of Safra-Piterman Construction
OK
Save as Default
Cancel
Büchi FSA (Classical Alphabet)

[ ] Use Schewe's history trees instead of Safra trees
[✓] Maximize the accepting set of the source automaton
[ ] Simplify the parity condition through the computation of Rabin index
[✓] Apply local optimization by using simulation relation to simplify the intermediate DPW
[✓] Apply local optimization by merging equivalent states in the conversion from DPW to NBW
[✓] Reduce transitions in the conversion from the complement NPW to the complement NBW
[✓] Reduce unreachable and dead states

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

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.

1.2 Implementation of the Fribourg Construction as a GOAL Plugin

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

1.2.1 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 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

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

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.

1.3 Verification of the Implementation

See UBELIX/jobs/2014-11-25

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

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

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

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

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

1.4 Test Data

1.4.1 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.

1.4.2 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
- Michel N2: 4 states, 14 transitions, 1 accepting state
- Michel N3: 5 states, 23 transitions, 1 accepting state

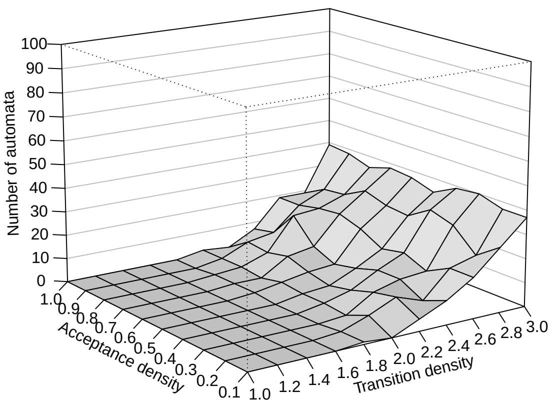
[4]http://goal.im.ntu.edu.tw/

	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
1.0	0	0	0	0	0	0	0	0	0	0
1.2	0	0	0	0	0	0	0	0	0	0
1.4	0	0	0	0	0	0	0	0	0	0
1.6	0	0	0	0	0	0	0	0	0	0
1.8	1	1	0	0	0	1	1	1	0	0
2.0	0	5	1	2	2	3	1	2	2	2
2.2	5	10	8	5	3	5	8	6	7	1
2.4	10	6	11	11	8	6	10	20	9	7
2.6	17	17	12	16	14	19	22	21	19	19
2.8	27	20	29	32	26	27	30	25	24	19
3.0	37	37	40	39	34	37	38	35	38	39

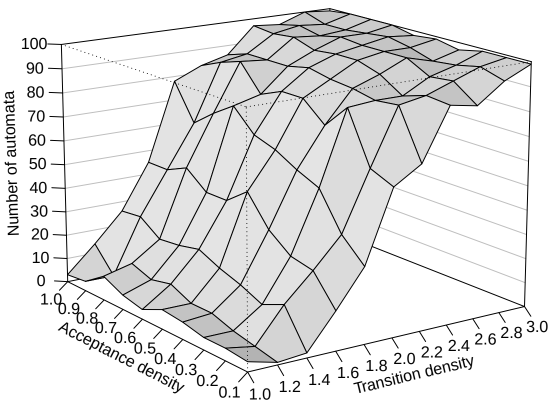
(a) Completeness

	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
1.0	4	5	5	7	8	4	6	10	4	3
1.2	1	3	5	8	8	12	10	13	4	14
1.4	2	17	13	17	20	24	22	21	27	26
1.6	16	28	30	37	49	42	42	49	45	45
1.8	31	40	55	59	64	67	76	70	63	78
2.0	60	64	85	75	83	83	79	90	87	83
2.2	67	87	86	88	89	91	89	89	89	86
2.4	88	89	86	92	95	95	94	97	96	97
2.6	86	93	92	97	97	97	98	96	98	96
2.8	94	97	95	94	97	99	98	97	97	100
3.0	99	99	99	97	99	98	100	100	100	99

(b) Universality



(c) Completeness



(d) Universality

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

- 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.

1.5 Internal Tests

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

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

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

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

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

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

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

Regarding the dependencies of the M1 and M2 optimisation, there are only two cases we can test. First, M1 alone, and second, M1 and M2 together. Assuming that the R2C optimisation adds a certain performance gain on top of an existing construction, we can then combine the better one of M1 and M1+M2 with R2C. We can already reveal at this point that M1 performs slightly better than M1+M2 on our test set, even though M1+M2 has a better theoretical worst-case complexity. This topic is further discussed in Chapter 2. Thus, the version that we will want to test is M1+R2C.

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

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

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

Summarising, for the internal tests we are going to carry out runs of the following versions of the Fribourg construction:

1. Fribourg
2. Fribourg+R2C

3. Fribourg+R2C+C
4. Fribourg+M1
5. Fribourg+M1+M2
6. Fribourg+M1+R2C
7. Fribourg+M1+R2C+MACC+R

1.6 External Tests

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

Test configurations

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

1.7 Execution Environment

UBELIX Linux cluster, runs Sun Grid Engine (load scheduler).

All the jobs are executed on machines belonging to one of the two following queues:

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

- highmem.q
  - jnode 01–21
  - Intel Xeon E5-2665 2.40GHz
  - 16 CPU cores (slots)
  - 256 GB RAM
  - → 16 GB RAM per core (slot)

Tests

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

The tests are successful with the following resources

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

Chapter 2

Results and Discussion

Contents

2.1 Internal Tests

2.1.1 GOAL Test Set

Timeouts and Memory Excesses

Timeout for each complementation task: 600 seconds CPU time. Memory limit by limiting Java heap to 1 GB.

Construction	Time	Memory
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	24	0

Table 2.1: Number of timeouts and memory excesses.

The number of effective samples is 10,939. That is, 10,939 of the 11,000 automata have been successfully complemented by all constructions, while 61 automata have failed to complement with at least one of the constructions.

It is strange that Fribourg+R has so much fewer timeouts than Fribourg

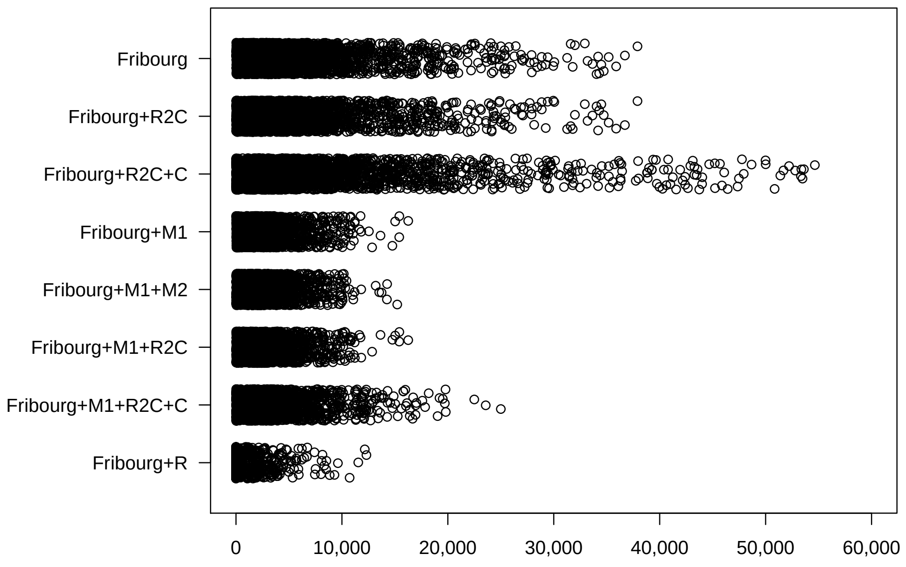


Figure 2.1: Number of states of complements.

Statistics Aggregated

Number of states of the produced complement automaton, aggregated over the whole test set.

Statistics over the execution time in CPU time of all the automata.

It is strange that Fribourg+R is so much faster than Fribourg

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: Aggregated statistics.

Construction	Mean	Min.	P25	Median	P75	Max.
Fribourg	8.53	2.49	3.30	4.89	7.26	585.99
Fribourg+R2C	6.63	2.19	2.86	4.18	6.40	219.68
Fribourg+R2C+C	8.54	2.16	2.56	3.48	6.38	582.87
Fribourg+M1	4.94	2.47	3.16	4.10	5.85	55.07
Fribourg+M1+M2	4.56	2.21	2.89	3.78	5.11	38.43
Fribourg+M1+R2C	4.44	2.22	2.83	3.59	5.27	42.48
Fribourg+M1+R2C+C	5.57	2.51	3.23	3.97	6.51	147.44
Fribourg+R	6.00	2.15	2.71	3.58	5.70	166.57

Table 2.3: Running times in seconds of the complementation tasks, measured as CPU time.

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

	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
1.0	580.3	465.0	786.4	401.2	398.4	300.3	336.7	263.3	297.4	77.3
1.2	1730.8	2050.2	1780.7	1808.5	1328.1	1298.1	1421.8	1108.0	987.4	136.4
1.4	3409.1	3251.7	2471.2	3052.9	2210.9	2255.1	1638.0	1437.3	1561.8	172.0
1.6	3493.9	3280.5	3395.7	3005.0	2215.7	1921.1	1599.5	1418.9	1401.5	170.3
1.8	2714.1	2418.7	2677.5	2199.3	2102.1	1689.1	1367.8	876.0	842.8	127.3
2.0	1778.3	1961.6	1769.2	1805.8	1401.0	1275.3	803.5	682.4	510.7	107.8
2.2	1180.9	1498.6	1257.8	1219.9	989.9	926.0	519.3	412.0	314.4	82.6
2.4	719.6	992.4	1036.3	848.8	659.2	615.9	388.6	287.8	195.4	60.8
2.6	553.0	682.8	673.3	660.3	555.1	483.2	265.0	190.2	153.2	49.1
2.8	454.6	490.5	609.6	489.8	419.2	336.3	222.7	154.1	118.9	41.1
3.0	276.4	381.4	403.5	390.1	323.8	255.3	136.2	119.5	82.2	35.0

asdflkjasdf

asdflkjf

	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
1.0	269.0	308.0	254.0	236.0	238.5	297.0	266.0	156.0	207.0	68.0
1.2	960.0	1,407.5	1,479.0	2,150.0	1,152.0	1,090.5	942.5	1,206.5	718.0	104.5
1.4	3,426.0	2,915.0	2,752.0	3,393.0	2,693.0	3,265.5	2,263.5	2,425.0	1,844.5	154.5
1.6	3,799.0	3,698.0	4,901.5	3,926.0	3,960.0	3,655.0	2,580.5	1,905.5	2,124.5	155.0
1.8	3,375.0	3,169.0	3,420.5	3,967.0	3,943.0	3,132.0	2,246.5	1,144.0	971.5	114.0
2.0	1,906.5	2,261.0	2,383.0	2,884.0	2,354.5	2,096.5	1,169.5	932.0	568.0	98.5
2.2	1,467.0	1,633.0	1,795.5	1,942.5	1,611.5	1,640.5	569.5	499.0	330.5	78.5
2.4	924.5	1,232.5	1,319.0	1,317.5	1,056.5	886.5	514.5	314.5	182.0	59.0
2.6	625.0	763.0	880.5	945.5	828.0	684.5	316.0	175.0	132.0	44.5
2.8	483.5	584.5	836.0	690.0	575.0	395.5	240.0	151.5	103.0	41.0
3.0	319.5	450.5	557.0	523.5	367.5	313.5	155.5	116.0	84.5	32.0

Version	Mean	Min	P25	Median	P75	Max
fribourg.goal	2,004.57	2	222.00	761.00	2,175.00	37,904
fribourg.m1.goal	963.17	2	177.00	482.00	1,138.00	16,260
fribourg.m1.m2.goal	958.00	2	181.00	496.00	1,156.50	15,223
fribourg.m1.r2c.goal	937.66	2	152.00	447.00	1,118.00	16,260
fribourg.m1.r2c.macc.r.goal	115.40	1	1.00	1.00	20.00	9,843
fribourg.r2c.c.goal	2,424.56	2	85.00	451.00	2,329.00	54,648
fribourg.r2c.goal	1,955.91	2	180.00	689.00	2,127.50	37,904

2.2 External Tests

2.2.1 GOAL Test Set

With `rank -tr -ro` there are 3796 uncompleted tasks (7,204 effective samples), without it only 7 (10,993 effective samples). Should the main analysis be done without rank? Because of rank, we would exclude the most difficult cases for the other constructions from the analysis.

With `piterman -eq -sim -ro`, the median is 1 and the 75th percentile 21. It' similar as when removing unreachable and dead states from the output automaton. I think this is caused by the sim optimisation, which simplifies one of the intermediate automata of the construction. Have to try the following:

- piterman -eq -sim -ro -r
- piterman -eq -ro

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.

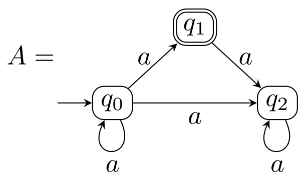
[8] A. P. Sistla, M. Y. Vardi, P. Wolper. The Complementation Problem for Büchi Automata with Applications to Temporal Logic. Theoretical Computer Science. 49(2–3):pp. 217 – 237. 1987.

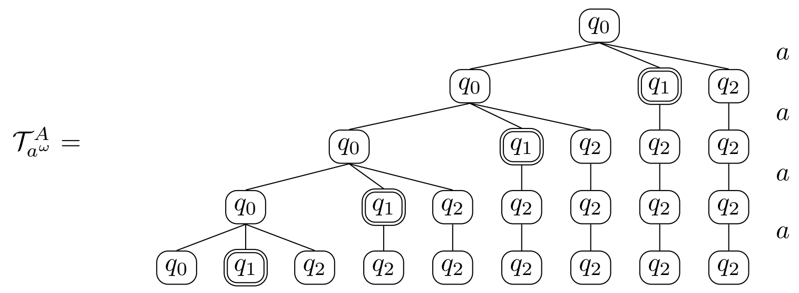
[9] W. Thomas. Complementation of Büchi Automata Revisited. In J. Karhumäki, H. Maurer, G. Păun, et al, eds., Jewels are Forever. pp. 109–120. Springer Berlin Heidelberg. 1999.

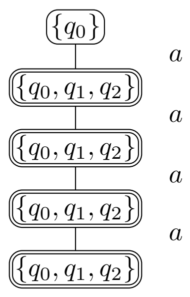
[10] Y.-K. Tsay, Y.-F. Chen, M.-H. Tsai, et al. Goal: A Graphical Tool for Manipulating Büchi Automata and Temporal Formulae. In O. Grumberg, M. Huth, eds., Tools and Algorithms for the Construction and Analysis of Systems. vol. 4424 of Lecture Notes in Computer Science. pp. 466–471. Springer Berlin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

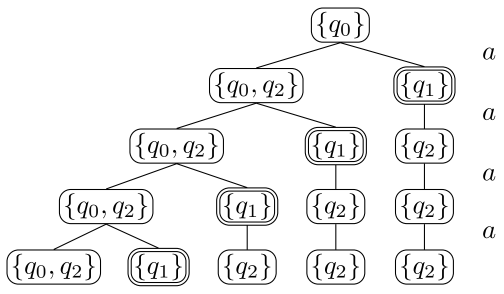
[11] Y.-K. Tsay, Y.-F. Chen, M.-H. Tsai, et al. Goal Extended: Towards a Research Tool for Omega Automata and Temporal Logic. In C. Ramakrishnan, J. Rehof, eds., Tools and Algorithms for the Construction and Analysis of Systems. vol. 4963 of Lecture Notes in Computer Science. pp. 346–350. Springer Berlin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

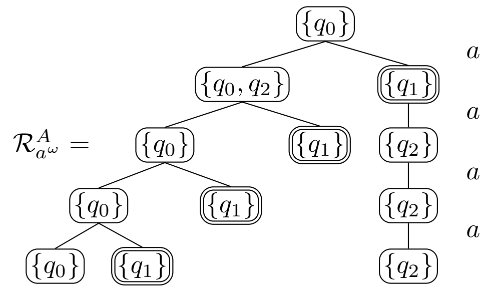
[12] M. Y. Vardi, T. Wilke. Automata: From Logics to Algorithms. In J. Flum, E. Grädel, T. Wilke, eds., Logic and Automata: History and Perspectives. vol. 2 of Texts in Logic and Games. pp. 629–736. Amsterdam University Press. 2007.


$A =$
q_0
q_1
q_2
a


$\mathcal{T}^A_{a^\omega} =$
q_0
q_1
q_2
a


$\{q_0\}$
$\{q_0, q_1, q_2\}$
a


$\{q_0\}$
$\{q_0, q_2\}$
$\{q_1\}$
$\{q_2\}$
a


$\mathcal{R}^A_{a^\omega} =$
$\{q_0\}$
$\{q_0, q_2\}$
$\{q_1\}$
$\{q_2\}$
a

$$\mathcal{G}^A_{a^\omega} = \begin{tikzcd}
q_0 \arrow[d] \arrow[dr] \arrow[drr] & q_1 & q_2 & a \\
q_0 \arrow[d] \arrow[dr] \arrow[drr] & q_1 \arrow[dr] & q_2 \arrow[d] & a \\
q_0 \arrow[d] \arrow[dr] \arrow[drr] & q_1 \arrow[dr] & q_2 \arrow[d] & a \\
q_0 \arrow[d] \arrow[dr] \arrow[drr] & q_1 \arrow[dr] & q_2 \arrow[d] & a \\
q_0 & q_1 & q_2 &
\end{tikzcd}$$