Performance Investigation of a Subset-Tuple Büchi Complementation Construction

Daniel Weibel

November 27, 2014

Contents

Chapter 1

Performance Investigation of the Fribourg Construction

In this chapter we come to the core of this thesis, the empirical investigation of the performance of the Fribourg construction. Our goal is to find out how the Fribourg construction performs on real automata. In particular, we are interested mainly in two things. First, how the Fribourg construction behaves under the different optimisations that we explained in Chapter **??**. Second, how the Fribourg construction compares to other complementation constructions.

1.1 The GOAL Tool

1.1.1 Overview

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 notions, and manipulate them in a multitude of ways. Relevant for our purposes are the ω-automata capabilites 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 grahical 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], 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,

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

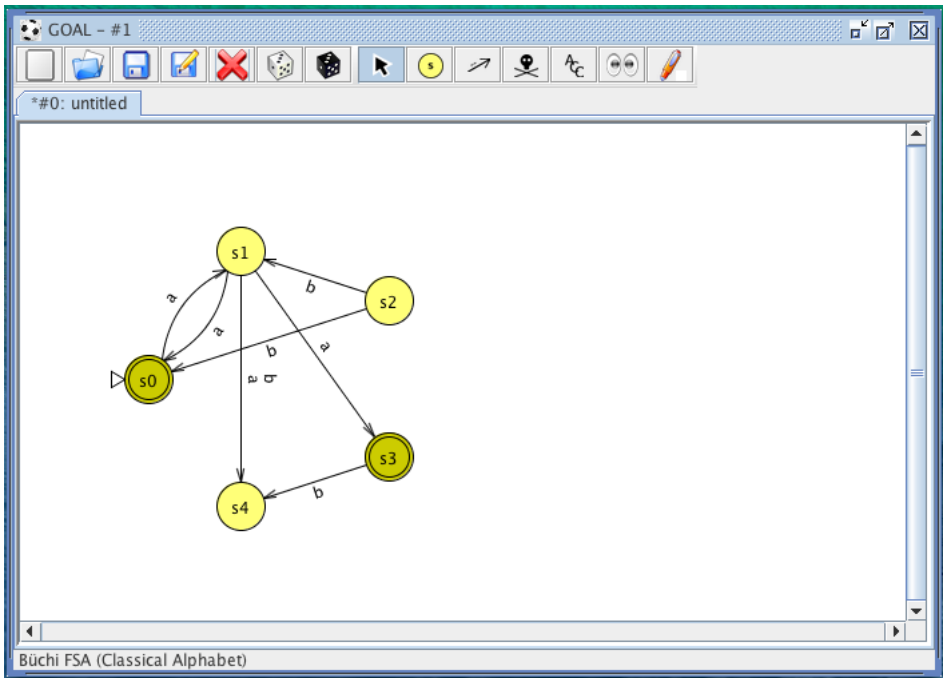


Figure 1.1: Graphical interface of GOAL.

and installing this plugin into the existing application. On the next start of the application, the new functionality will be included, all without requring 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.

1.1.2 Büchi Complementation

There are a couple of Büchi complementation constructions preimplemented 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. Furthemore, 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

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]

the optimsations 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.

1.2 Implementation of the Fribourg Construction

We implemented the Fribourg construction, including its optimsations, 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.

1.3 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

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

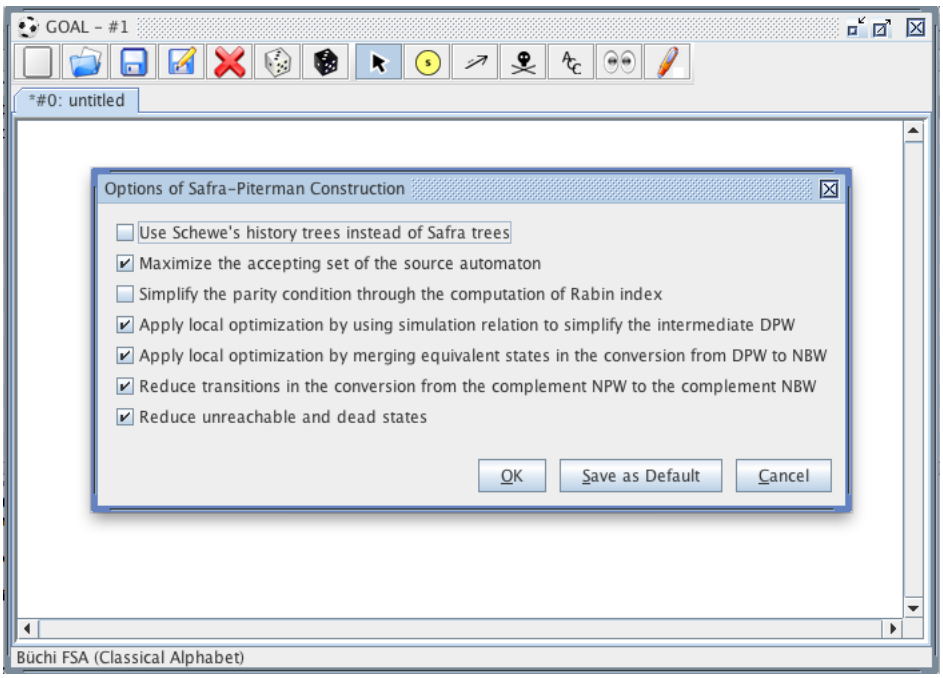


Figure 1.2: Complementation constructions in GOAL can have a set user-selectable options. Here the options of the Safra-Piterman construction.

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

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

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

1.3.1 Correctness of the Implementation

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.

Clearly, it would be appropriate 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.

1.3.2 Plugin Availability and Installation

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

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

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.

1.4 Experiment Idea

1.5 Experiment Setup

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.

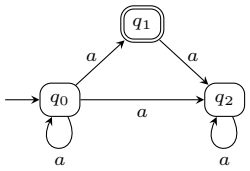
[9] W. Thomas. Complementation of Büchi Automata Revisited. In J. Karhumäki, H. Maurer, G. Păun, et al, eds., *Jewels are Forever*. pp. 109–120. Springer Berlin Heidelberg. 1999.

[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.

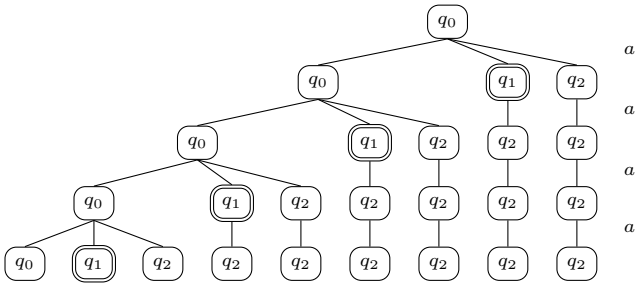
[11] Y.-K. Tsay, Y.-F. Chen, M.-H. Tsai, et al. Goal Extended: Towards a Research Tool for Omega Automata and Temporal Logic. In C. Ramakrishnan, J. Rehof, eds., *Tools and Algorithms for the Construction and Analysis of Systems*. vol. 4963 of *Lecture Notes in Computer Science*. pp. 346–350. Springer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

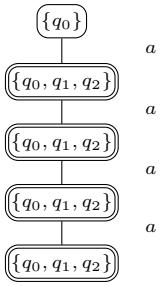
[12] M. Y. Vardi, T. Wilke. Automata: From Logics to Algorithms. In J. Flum, E. Grädel, T. Wilke, eds., *Logic and Automata: History and Perspectives*. vol. 2 of *Texts in Logic and Games*. pp. 629–736. Amsterdam University Press. 2007.

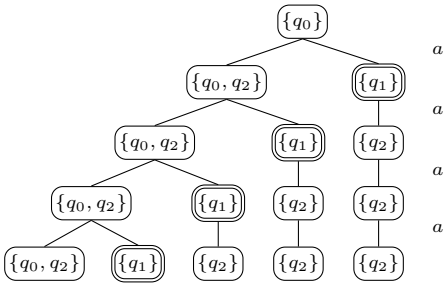
$A =$



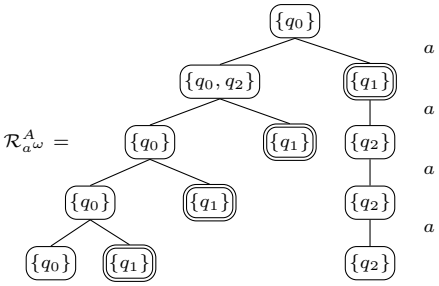
$\mathcal{T}^A_{a^\omega} =$







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



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