ELSEVIER
Available online at www.sciencedirect.com
ScienceDirect
Theoretical Computer Science 363 (2006) 224–233
Theoretical Computer Science
www.elsevier.com/locate/tcs

Observations on determinization of Büchi automata

Christoph Schulte Althoff, Wolfgang Thomas, Nico Wallmeier*

RWTH Aachen, Lehrstuhl Informatik 7, 52056 Aachen, Germany

Abstract

The two determinization procedures of Safra and Muller–Schupp for Büchi automata are compared, based on an implementation in a program called OmegaDet.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Determinization; Rabin; Muller–Schupp

1. Introduction

A central result in the theory of ω-automata is McNaughton's Theorem [6]. In its original formulation it says that a non-deterministic Büchi automaton can be converted into a deterministic Muller automaton. Many constructions have been proposed to show this determinization result (cf. [14,10] for hints to the literature). In most cases, the target automaton is a deterministic Rabin automaton, which can be considered as a special form of Muller automaton. It is well-known that the blow-up in number of states has to be greater than in the classical subset construction; there is a lower bound of $2^{O(n \log n)}$ for the number of states of a deterministic Rabin automaton, given a Büchi automaton with n states ([7,5]).

Safra [11] was the first to find a construction which matches this lower bound. It seems now the standard way of showing Büchi automata determinization. But there is a second construction, again matching the lower bound, due to Muller and Schupp [9]. Teaching experience of the second author indicates that the Muller–Schupp proof can be explained more easily (the reader can judge him/herself below). The two constructions are sufficiently different to justify a closer comparison. This is the aim of present paper.

The study is based on an implementation of the two algorithms in a program called OmegaDet. As it turns out, such an implementation is necessary not only for a serious experimental performance comparison of the two procedures but also for a better conceptual understanding of their behavior. The two algorithms are too involved to be analyzed by hand if one is interested in studying say a dozen examples. We report here on some insights we obtained in this investigation, both regarding a better understanding of the characteristics (and the similarities) of the two algorithms and of their performance. We observe that (apart from some peripheral cases), the Safra algorithm uses stronger abstractions than the one of Muller–Schupp and thus yields smaller automata. (Maybe this is a reason for the difficulties the Safra

* Corresponding author.
E-mail addresses: althoff@i7.informatik.rwth-aachen.de (C.S. Althoff), thomas@i7.informatik.rwth-aachen.de (W. Thomas), wallmeier@i7.informatik.rwth-aachen.de (N. Wallmeier).

0304-3975/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2006.07.026

algorithm poses for exposition in lectures.) Moreover, our experiments led to an improvement of the Muller–Schupp procedure which can reduce the state space of the target automaton.

The paper is structured as follows. In the subsequent Section 2, we present the determinization procedures, with an emphasis on singling out those points where they coincide and where they differ. In this exposition, we use the insights from our experiments; we start with an explanation of the Muller–Schupp algorithm and more briefly discuss the Safra construction. We do not give any details about the correctness proofs. In Section 3, we give a brief introduction to (the user's view of) the program OmegaDet, in which also the subset based construction of Hayashi and Miyano [8] for co-Büchi automata determinization is included. We report on observations obtained in case studies and suggest the above mentioned improvement of the Muller–Schupp algorithm. In Section 4, we comment on the context of our work as well as on perspectives for extensions.

2. The algorithms of Safra and Muller–Schupp

We consider Büchi automata in the format $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ where Q is the finite set of states, Σ the input alphabet, q_0 the initial state, $\Delta \subseteq Q \times \Sigma \times Q$ the transition relation, and $F \subseteq Q$ the set of "final states". The automaton $\mathcal{A}$ accepts the ω-word $\alpha \in \Sigma^\omega$ if a run $\rho \in Q^\omega$ exists (defined in the standard way) with infinitely many occurrences of states in F. In other words, one may consider the *run tree* t_α of $\mathcal{A}$ on α, which has a root (considered to be at "level 0") labelled q_0, and displays level by level the states reached after the α-prefixes $\alpha(0) \ldots \alpha(i-1)$ for $i = 0, 1, 2, \ldots$. Formally, a vertex on level i labelled p has the successor nodes labelled $q_1, \ldots, q_k$ if for p and the letter $a = \alpha(i)$ the transitions $(p, a, q_1), \ldots, (p, a, q_k)$ are applicable. We assume the tree to be sibling ordered, with reference to an ordering of the set of states. The *run dag* r_α of $\mathcal{A}$ on α is obtained inductively from t_α, level by level, by deleting a vertex v labelled q if on the same level a vertex u labelled q appears to the left; in this case an edge is added from the parent vertex of v to u. Clearly, the input word α is accepted by $\mathcal{A}$ iff in the run dag r_α there is an infinite path from the root on which an F-state occurs infinitely often (henceforth we just speak of a "successful path").

A deterministic Rabin automaton (we say "Rabin automaton" to be short) has the format $\mathcal{A} = (Q, \Sigma, q_0, \delta, \Omega)$ where Q, Σ, q_0 are as for Büchi automata, $\delta : Q \times \Sigma \to Q$ is the transition function, and $\Omega = ((E_1, F_1), \ldots, (E_k, F_k))$ a list of "accepting pairs" with $E_j, F_j \subset Q$. The automaton accepts the input word $\alpha \in \Sigma^\omega$ if for the unique run $\rho \in Q^\omega$ of $\mathcal{A}$ on α an index j exists such that some F_j-state is visited infinitely often in ρ but each E_j-state only finitely often.

The starting point for the transformation of a Büchi automaton $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ into an equivalent Rabin automaton (i.e., recognizing the same ω-language) is to use a finite abstraction of the infinity of the finite prefixes of run trees. If the Büchi automaton has scanned the prefix $\alpha(0) \ldots \alpha(i-1)$ of the input, the run tree up to level i is built up. Taking the run dag instead, one observes that a structure of finite width suffices (since each state can occur at most once on each level). The main point of the transformation of $\mathcal{A}$ into a finite deterministic automaton is to invent a finite number of representations of the infinitely many possible run dag prefixes, in a way that the existence of an infinite path with infinitely many F-states can still be detected. For this, it is necessary to separate the different threads of the run tree (or run dag) for recording of the occurrence of final states. Both algorithms, the procedures by Safra and Muller–Schupp, use tree structures for this purpose. A node in such a tree provides the information which states are presently visited in certain threads of the run tree; in particular, the root records the totality of presently reachable states (as in the classical subset construction). Also both procedures adopt the convention that a state is kept only at its leftmost occurrence on a tree level, thus inheriting the rule mentioned above for constructing run dags.

2.1. Muller–Schupp trees

Let us first introduce the tree structures used by Muller and Schupp, called Muller–Schupp trees in the present paper. A *Muller–Schupp tree* is a finite sibling-ordered strictly binary tree (i.e., each vertex except the leaves has precisely two sons), whose vertices are named with positive natural numbers and additionally are labelled by two items: a subset of Q and a color from the set {red, yellow, green}. Since by construction the set of states at a parent node is the disjoint union of the sets at the two sons, it would suffice to keep state-sets as labels only for the leaves; however, for easier readability of the trees we prefer to use the state-set labelling throughout.

The Muller–Schupp trees can be introduced in three stages, starting from the computation tree t_α of the given Büchi automaton on some input word α. The first step consists in partitioning the sons of a vertex v into two classes, those which carry a final state and those which carry a non-final state. The former are collected in a set and declared as the label of the left son of v, the latter (non-final) ones form the label of the right son (of course, one of the two sons can vanish). Call the resulting tree with at most binary branching the "acceptance tree" t_α^1. It is easy to verify that

t_α has a successful path iff t_α^1 has a path branching left infinitely often

If we keep only the occurrences of a state q which occur leftmost on the respective level of the tree t_α^1, we obtain the tree t_α^2, and note that the equivalence above holds also for t_α^2 instead of t_α^1.

The tree t_α^2 grows in a deterministic fashion level by level, given α. Note that each level has at most as many entries as there are states in $\mathcal{A}$, so t_α^2 is of bounded width. The idea is to take as states of the deterministic Rabin automaton compressed versions of the t_α^2-prefixes, level by level: a path segment from a left son, respectively a right son, or from the root, to the next branching point v is contracted into v. Then a strictly binary tree is obtained. The states of such a path segment are forgotten except some information about the presence of final states, given by three different colors which the remaining vertex v can have: red, yellow, and green. It is clear that the number of such trees is finite.

The update step upon processing an input letter a (corresponding to the extension of t_α^2 by one level) is performed by attaching sons to the leaves according to the subset construction, starting from the state set at each leaf. Of course, no son is introduced to a leaf if from none of its states a continuation of the run via a is possible. This case leads to the deletion of the whole path back to the last branching point. Vertex names which are freed by this can be reused, however not in the same update step. In the remaining case a left son, a right son, or both are introduced, depending on whether there are only final states, only non-final states, or both in the resulting state set. When a final state is encountered, the vertex carrying it is colored green. By the cancellation policy (to keep only the leftmost occurrence of a state) and the path compression procedure it can happen that path segments (i.e., strictly speaking, tree vertices) are merged into a single vertex (again setting free the name of the spared vertex). In this case the parent vertex may receive final states from a son with which it is merged; we say then that it "receives a new final state".

A vertex is colored red if the path segment it represents has no final state, yellow if it has a final state but did not receive this state in the last step, and it is colored green if it received a final state in the last step, either at a leaf via the subset construction or by a merge step, for example with a vertex previously colored yellow.

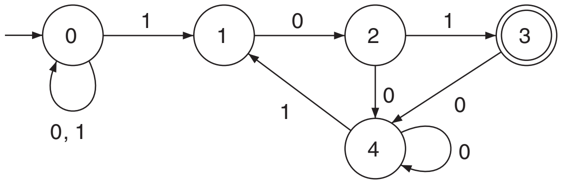
Using this update procedure, it turns out that t_α^2 has a path branching left infinitely often iff in the sequence of corresponding Muller–Schupp trees some vertex v stays forever from some point onwards and has the color green again and again. This is captured by a Rabin acceptance condition of pairs (E_i, F_i) where i ranges over the finite reservoir of vertex names: E_i contains those trees where i is missing, and F_i has those trees where i occurs colored green.

Formally, the update for a tree t and input letter a is carried out as follows:

Update of Muller–Schupp tree

(1) Copy the given tree t, changing all colors green to yellow.
(2) Apply the subset construction (via letter a) to each leaf, add left and right son carrying the reached final, respectively, non-final states; color these sons green and red, respectively.
(3) Keep only the leftmost occurrence of each state.
(4) Delete the vertices which are only on paths leading to leaves whose value is the empty set.
(5) As long as there exists a vertex of degree one merge this vertex with its successor, inheriting the color green if this successor was colored green or yellow.
(6) Proceeding from the leaves, label each parent by the union of the two state sets from the labelling of the two sons.

Example 1. Let $\mathcal{A}_0$ be the Büchi automaton



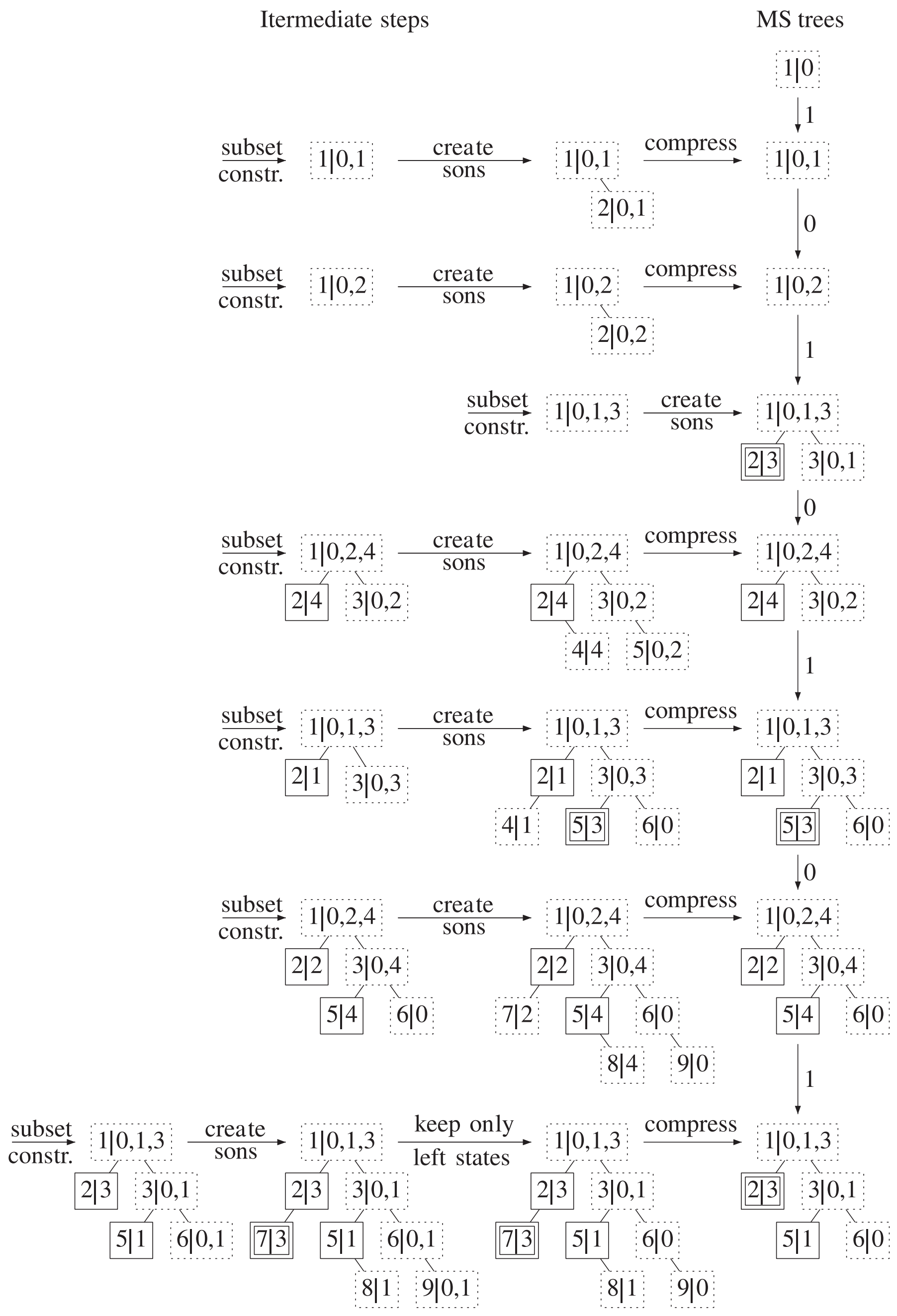


Fig. 1. MS-trees (with intermediate steps) for input 1010101 on automaton $\mathcal{A}_0$.

which accepts the 0–1-sequences which have the segment 11 only finitely often but 101 infinitely often. The run dag of $\mathcal{A}_0$ on the input word 1010101 . . . is given on the left of Fig. 2. In Fig. 1 we present the Muller–Schupp trees (MS-trees) for this input word on the right, with intermediate results of the computation to its left. A vertex is named by a number; following the stroke we list the states belonging to its label. In a Muller–Schupp tree, a vertex colored red is given as a dashed rectangle, colored yellow as a simple rectangle, and colored green as a double-line rectangle.

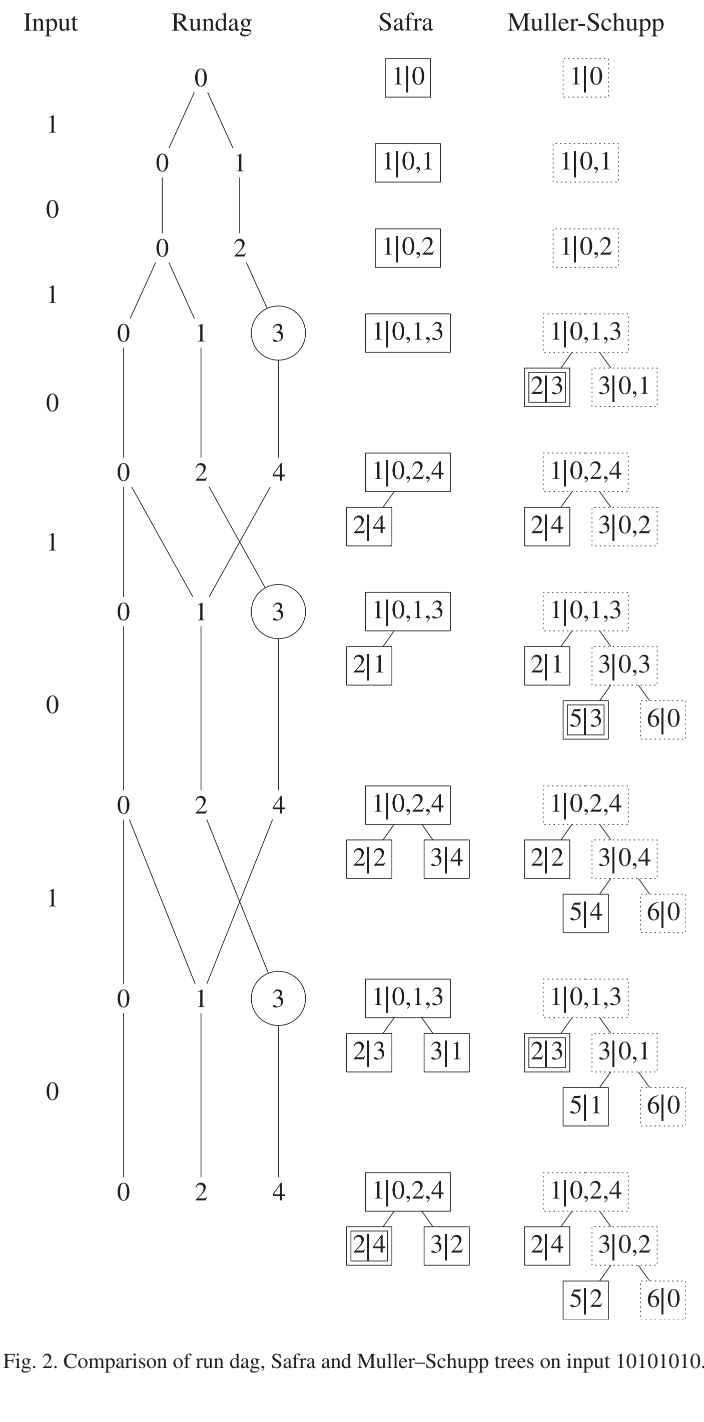


Fig. 2. Comparison of run dag, Safra and Muller–Schupp trees on input 10101010.

2.2. *Safra trees*

The Safra trees are more succinct in the sense that they suppress as much as possible the record of non-final states. Starting from the update step of the Muller–Schupp algorithm (which of course was not the way these algorithms were invented), the Safra construction introduces a technical simplification when the subset construction is applied: here only the left son (containing the final states reached) of a vertex v is kept in the tree, no right son for the non-final states is introduced. When from these non-final states at later stages final states are reached, new son vertices of v are created successively in the Safra tree; in this situation more than binary branching may occur. In a Muller–Schupp tree the intermediate vertices with non-final state-sets amount to a binary encoding of the Safra trees. However, due to different coloring policies, the embedding of a Safra tree into the corresponding Muller–Schupp tree cannot in general be lifted to an embedding of the Safra state space into the Muller–Schupp state space. In particular, a Safra automaton can also be larger than the corresponding Muller–Schupp automaton; cf. the remark at the end of Section 3. A didactic advantage of the Muller–Schupp trees is that they convey more directly the structure of the computation tree of the given Büchi automaton.

The difference of colorings reflects different recordings of visits of final states. The Muller–Schupp procedure uses the coloring policy to signal "new visits to final states". The Safra algorithm marks a node by color green according to the so-called breakpoint construction, signalling that all states of the node can be reached via a *past* visit to a final state. Similarly as for Muller–Schupp trees, a run is accepting if some vertex stays indefinitely from some point onwards and is colored green again and again. Formally, the update for a Safra tree t and input letter a is carried out as follows (following to [11]):

Update of Safra tree

(1) Copy the given tree t, removing all colors green.
(2) For all nodes v of the tree create a new youngest son containing the final states of node v if there exists such ones.
(3) Apply the subset construction (via letter a) to all nodes.
(4) Keep only the leftmost occurrence of each state.
(5) Remove all nodes with empty labels.
(6) For every node v whose label is equal to the union of the labels of its sons, remove all the descendants of v and color v green.

Example 2. Let $\mathcal{A}_0$ be the same Büchi automaton as in Example 1. In Fig. 2 we present the run dag for the input word 10101010 . . ., right to it the run of Safra trees and then the run of Muller–Schupp trees. We skip the intermediate stages in the construction process between two successive trees. We use the same notation as in Example 1. Similarly, the vertices of a Safra tree receiving the mark green are displayed in double-line rectangles.

3. Implementation, experiments, conclusions

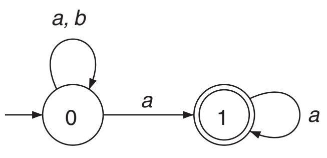
OmegaDet is a program written in C++ which offers implementations of four determinization procedures for Büchi automata:

- the Safra construction;
- the Muller–Schupp construction;
- an optimized Muller–Schupp version (which is presented later in this section);
- the Hayashi–Miyano construction (which can be applied to co-Büchi automata)

The fourth option is included since many examples we considered turned out to be co-Büchi automata. This happens, for example, in cases where the accepting loops consist of a single state only. In this case the Büchi condition holds iff from some time onwards only final states are encountered (co-Büchi acceptance). Other examples were not co-Büchi automata as such but could be transformed to this shape by declaring more states (than previously) as final, without changing the accepted language. Thus, a useful preprocessing consists in successively declaring as final all states from which only final states are reachable, which does not change the language but might lead to a co-Büchi automaton.

The Hayashi–Miyano construction is a simple refinement of the subset construction, needing only $2^{O(n)}$ states (essentially two subsets) and will be preferable if a large automaton can be presented as a co-Büchi automaton. An implementation exists also in the LASH package at Liège (see [8,1,13]).

The program OmegaDet asks the user to supply a Büchi automaton as a text file. The format of text file is best explained by an example. Consider the following Büchi automaton $\mathcal{A}_1$ which accepts all ω-words with only finitely many b.



```
2
ab
1
0 a 0
0 b 0
0 a 1
1 a 1
```

The first line of the text file is the number n of states of the Büchi automaton, whose state set is then assumed to be $\{0, \dots, n-1\}$. The initial state is 0. The second line contains the used alphabet Σ as a string of single ASCII symbols. Each symbol of the string is used as a single letter. Afterwards the final states are numerated in the third line separated by spaces. Then the transitions of the automaton are listed.

The user can choose the desired algorithms in a menu. He has the possibility to either simulate a run interactively or to compute the deterministic Rabin automaton. In the latter case, the program reports progress after every 200 computed states. The output is then delivered in four parts:

- the number of states;
- the list of Safra trees, respectively Muller–Schupp trees, each introduced by a name si, respectively, ki (for $i = 0, 1, \dots$) together with the word (the first in the canonical ordering of words) via which the state is reached, and then a display of the tree (to be explained below);
- the transition table, using the state names si, respectively ki;
- the list of accepting pairs and number of accepting pairs.

For the display of trees, a textual representation is used which indicates the sons of some node by the subsequent lines, marked as sons by indentation after a pointer symbol +->. So brother nodes are listed with same indentation. The colors red, yellow, green are presented as the symbols $-$, 0, $+$, following each vertex in Muller–Schupp trees; in Safra trees a mark ! is attached to a vertex if it has color green.

As an example of the output we list the automaton according to Safra generated by the example above.

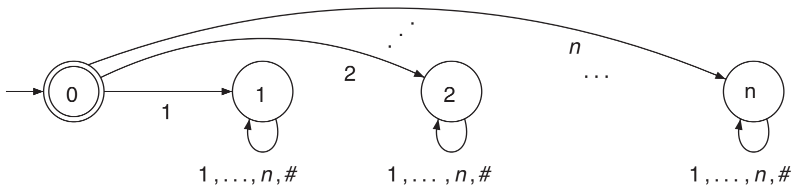
```
Deterministic Rabin automaton
according to Safra:

4 States:
s0:
    [1|0]

s1: a
    [1|0,1]

s2: aa
    [1|0,1]
     +-> [2|1]

s3: aaa
    [1|0,1]
     +-> [2|1]!
```

```
Transition table:
     a   b
s0   s1  s0
s1   s2  s0
s2   s3  s0
s3   s3  s0

Acceptance pairs:
for vertex 2 (sizes 2,1):
({s0,s1},{s3})

Overall: 1 pair with non-empty
acceptance set
```

We used the program for various Büchi automata, among them also the automata suggested by Michel [7] for showing the $2^{O(n \log n)}$ lower bound for complementation of Büchi automata (see also [10,15]). Recall that the Büchi automaton $\mathcal{M}_n$ considered in [7] has states $0, \dots, n$, the input alphabet $\{1, \dots, n, \sharp\}$ and the following transition graph:

Table 1
Comparison of Safra, Muller–Schupp and optimized Muller–Schupp algorithms

	Safra		Muller–Schupp		Opt. Muller–Schupp	
	States	Pairs	States	Pairs	States	Pairs
$\mathcal{M}_1$ (2 states)	7	1	9	5	9	5
$\mathcal{M}_2$ (3 states)	33	2	4058	8	262	7
$\mathcal{M}_3$ (4 states)	385	5	4,823,543	11	23,225	9
$\mathcal{M}_4$ (5 states)	13,601	7	Memory exceeded		3,656,802	11
$\mathcal{M}_5$ (6 states)	1,059,057	9	Memory exceeded		Memory exceeded	



While we could compute the Rabin automaton according to Safra's construction easily up to automaton $\mathcal{M}_5$ (where a million states are reached for the first time), the Rabin automaton following Muller–Schupp has about 5 million states already for $\mathcal{M}_3$, and the system ran out of memory for $\mathcal{M}_4$ (see Table 1). The reader can obtain an impression of the program OmegaDet by calling `http://www-i7.informatik.rwth-aachen.de/d/research/omegadet.html`; there the program itself, as well as the text files for the automata $\mathcal{M}_1, \ldots, \mathcal{M}_5$ and the output files are listed.

It was already mentioned in the previous section that the Muller–Schupp trees tend to be larger due to the inclusion of vertices whose labels are formed from non-final states. A more serious effect, however, is the procedure for introducing new vertices as sons of the leaves and the naming scheme pursued here. Many trees can be generated which have the same structure of vertex labels (and colors), whereas a difference occurs in the vertex naming. An idea to spare vertex names is to add new sons only for leaves which contain final states as well as non-final states (and to apply immediate merging with the parent if only final or only non-final states are encountered). This leads to a more restrictive way of introducing vertex names.

Formally, we proceed as follows in this modification of the Muller–Schupp algorithm:

Optimized update of Muller–Schupp tree

(1) Copy the given tree t, changing all colors green to yellow.
(2) Apply the subset construction (via letter a) to each leaf.
(3) Keep only the leftmost occurrence of each state.
(4) For all leaves which contain final states as well as non-final states add left and right son carrying the reached final, respectively non-final states; color these sons green and red, respectively.
(5) Color all leaves containing only final states green.
(6) Items (4), (5) and (6) of original algorithm.

Indeed, this modification can spare many states, as seen in the table above (fourth column). In Fig. 2, the optimization is visible in the last four listed Muller–Schupp trees: the vertex names 5,6 are changed there to 4,5. Nevertheless, the number of the Muller–Schupp trees grows still much faster than the number of Safra trees.

More case studies are reported in [12]. A simple general statement relating the numbers of states of the two constructions is not obvious, since we observed several cases where the Safra construction gives a slightly larger automaton than the Muller–Schupp procedure. This happens, for example, for the automaton $\mathcal{A}_1$ considered above, where the Safra construction yields four states and the Muller–Schupp construction (the normal one as well as the optimized one) only two states. The delayed signalization of "success" (by an extended initialization) yields different Safra trees, distinguished by different colorings, for a unique Muller–Schupp tree. A possibility to spare states in the Safra construction, which appears in the exposition of [10], is to exchange the application of the powerset construction and the creation of sons. In the example just mentioned this spares one state, however still leads to a larger Safra automaton than the Muller–Schupp one.

4. Outlook

We have presented the determinization procedures of Safra and Muller–Schupp, working out their similarities and their differences. Remarks on the implementation focussed on the input and output format and the explanation of simple case studies. In Michel's example we found a drastic difference between the two procedures, giving an advantage to Safra's algorithm. We explained this effect and suggested an improvement of the Muller–Schupp algorithm, giving (in some cases) much smaller automata.

As a result of our experimental studies we found a tighter connection between the two algorithms than expected, and we found the superiority of the Safra procedure for building small automata.

Despite much research, the determinization algorithms have so far not reached the stage of application examples of any serious scale, in definite contrast to the many implementations based on Büchi automata and alternating automata, applied mostly in model-checking. On the other hand, determinization is a necessary prerequisite in certain domains, for example in the solution of games with regular winning conditions (where a presentation by non-deterministic automata does not suffice). It seems that before reaching real practice, determinization still needs more investigation, not only with respect to the procedures as such (as done in this paper) but also concerning the phases of "preprocessing" and "postprocessing". The preprocessing phase would involve an analysis of the given Büchi automaton, possibly its reduction e.g., with methods of [2] and the test whether it can as well be represented as a co-Büchi automaton. In the latter case, the simpler determinization procedure of Hayashi and Miyano [8] can be applied (see e.g., [1]). The postprocessing would deal with the reduction or even minimization of deterministic Rabin automata, and it would have to deal with a parameter which we did not consider in the present paper: the complexity of the acceptance condition (number of accepting pairs in a Rabin automaton). So the present paper just addresses a single aspect in the general problem of implementing Büchi automaton determinization.

Some of these aspects are considered by Klein and Baier in [4]. There they have presented some improvements of the Safra algorithm and developed a postprocessing step by computing a bisimulation quotient.

Acknowledgments

We thank Detlef Kähler for his presentation of [3] in our seminar and remarks on a preliminary version of the paper; we relied on his work at several places. The second author acknowledges fruitful discussions with Christof Löding while preparing a lecture on the Muller–Schupp algorithm.

References

[1] B. Boigelot, S. Jodogne, P. Wolper, An effective decision procedure for linear arithmetic with integer and real variables, ACM Trans. Comput. Logic 6 (3) (2005) 614–633.
[2] K. Etessami, T. Wilke, R.A. Schuller, Fair simulation relations, parity games, and state space reduction for Büchi automata, in: Proc. of ICALP 2001, Lecture Notes in Computer Science, Vol. 2076, Springer, Berlin, 2001, pp. 694–707.
[3] D. Kähler, Determinisierung von ω-automaten, Diplomarbeit, Christian-Albrechts-Universität zu Kiel, Juni 2001.
[4] J. Klein, C. Baier, Experiments with deterministic ω-automata for formulas of linear temporal logic, in: Proc. 10th Internat. Conf. on the Implementation and Application of Automata, CIAA 2005, Lecture Notes in Computer Science, Vol. 3845, Springer, Berlin, 2005.
[5] C. Löding, Optimal bounds for the transformation of omega-automata, in: Proc. 19th Conf. on Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science, Vol. 1738, Springer, Berlin, 1999, pp. 97–109.
[6] R. McNaughton, Testing and generating infinite sequences by a finite automaton, Inform. and Control 9 (5) (1966) 521–530.
[7] M. Michel, Complementation is more difficult with automata on infinite words, cNET, Paris, 1988.
[8] S. Miyano, T. Hayashi, Alternating finite automata on ω-words, Theoret. Comput. Sci. 32 (1984) 321–330.
[9] 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, Theoret. Comput. Sci. 141 (1&2) (1995) 69–107.
[10] D. Perrin, J.-E. Pin, Infinite Words: Automata, Semigroups, Logic and Games, Pure and Applied Mathematics, Vol. 141, Elsevier, Amsterdam, 2004.
[11] S. Safra, On the complexity of ω-automata, in: Proc. 29th Symp. on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1988, pp. 319–327.

[12] C. Schulte-Althoff, Construction of deterministic ω-automata: a comparative analysis of the algorithms by safra and muller/schupp, Diplomarbeit, RWTH Aachen, April 2005.
[13] The Liège Automata-based Symbolic Handler (LASH), available at ⟨http://www.montefiore.ulg.ac.be/∼boigelot/research/lash/⟩.
[14] W. Thomas, Automata on infinite objects, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Vol. B, Formal models and semantics, Elsevier, Amsterdam, 1990, pp. 133–191 (Chapter 4).
[15] W. Thomas, Languages, Automata and logic, in: Handbook of Formal Language Theory, Vol. III, Springer, Berlin, 1997, pp. 389–455.