

MASARYK UNIVERSITY  
FACULTY OF INFORMATICS



# **Transformation of Nondeterministic Büchi Automata to Slim Automata**

BACHELOR'S THESIS

**Pavel Šimovec**

Brno, Fall 2020



*Replace this page with a copy of the official signed thesis assignment and a copy of the Statement of an Author.*



## Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Pavel Šimovec

**Advisor:** RNDr. František Blahoudek, Ph.D.; doc. RNDr. Jan Strejček, Ph.D.



## Acknowledgements

ack

# **Abstract**

abstract



## Keywords

keyword1, keyword2, ...



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Preliminaries</b>	<b>3</b>
2.1	Büchi Automaton . . . . .	3
2.2	Generalized Büchi Automaton . . . . .	4
2.3	Breakpoint Automaton . . . . .	4
2.4	Slim Automata Construction [separate chapter?] . . . .	5
2.5	Slim Automaton Construction Generalized to TGBA . .	5
<b>3</b>	<b>Implementation</b>	<b>7</b>
3.1	Technologies/Tools . . . . .	7
3.1.1	Seminator . . . . .	7
3.1.2	Spot . . . . .	7
3.2	Create Slim Automata Using Seminator . . . . .	7
3.3	Implementation of Slim Automata inside Seminator . .	8
3.4	Testing and Verification . . . . .	9
3.5	How to Install Seminator . . . . .	9
3.6	Future of Implementation . . . . .	9
<b>4</b>	<b>Evaluation</b>	<b>11</b>
4.1	Optimized . . . . .	11
4.2	weak/strong . . . . .	11
4.2.1	Comparisons among Unoptimized options . . .	11
4.2.2	Comparisons among Post-Optimized options . .	11
4.3	via tba/no change . . . . .	11
4.3.1	Comparisons among Unoptimized options . . .	11
4.3.2	Comparisons among Post-Optimized options . .	11
4.4	Slim automata produced seminator ePMC . . . . .	11
4.5	Compare with semi-deterministic Automata . . . . .	12
4.6	To Put Somewhere in This Chapter . . . . .	12
<b>5</b>	<b>Conclusion</b>	<b>13</b>



## List of Figures



# 1 Introduction

... slim automata are specially constructed Büchi automata...





## 2 Preliminaries

In this chapter we define a Büchi automaton and its generalized version. Then we continue with breakpoint algorithm. It allows us to introduce slim automata by its construction, which builds on the breakpoint one. Finally we generalize slim automation construction to work with generalized Büchi automata.

We will need to know that on *alphabet* is a set of letters, an  $\omega$ -word  $w \in \Sigma^\omega$  is an infinite sequence of letters, and a *language*  $L \subseteq \Sigma^\omega$  is a set of  $\omega$ -words.

### 2.1 Büchi Automaton

A Büchi automaton is a theoretical finite-state machine used to define  $\omega$ -languages. It decides which infinitely long words ( $\omega$ -words) belong to its language.

A *transition-based Büchi automaton* (TBA) is a tuple  $\mathcal{A} \stackrel{\text{def}}{=} (\Sigma, Q, q_i, \Delta, \Gamma)$ , where

- $\Sigma$  is a non-empty finite *alphabet*,
- $Q$  is a non-empty finite set of *states*,
- $q_i \in Q$  is the initial state of  $\mathcal{A}$ .
- We write the set of *transitions* as  $\Delta \subseteq Q \times \Sigma \times Q$ . Intuitively, a transition  $(s, a, t)$  directionally connects the states  $s$  and  $t$  with the letter  $a$ .
- $\Gamma \subseteq \Delta$  is a set of *accepting transitions*.

A *run*  $r$  of  $\mathcal{A}$  is an infinite sequence of transitions  $r \stackrel{\text{def}}{=} t_0 t_1 \dots \in \Delta^\omega$ , where  $t_i = (s_i, a_i, s_{i+1})$ , such that  $q_0 = q_i$ . A run of  $\mathcal{A}$  is *accepting* iff it contains infinitely many accepting transitions.

Finally, we define the *language*  $L(\mathcal{A}) \subseteq \Sigma^\omega$  recognized by the automaton  $\mathcal{A}$ . An  $\omega$ -word  $w \in \Sigma^\omega$  belongs to  $L(\mathcal{A})$  iff there exists an accepting run of  $\mathcal{A}$  over the word  $w$ .

## 2.2 Generalized Büchi Automaton

A *transition-based Generalized Büchi automaton* (TGBA) is a tuple  $\mathcal{A} \stackrel{\text{def}}{=} (\Sigma, Q, q_i, \Delta, G)$ , where  $\emptyset \subseteq G \subseteq 2^\Delta$  contains sets of accepting conditions and the rest is defined as for TBA. A run of  $\mathcal{A}$  is *accepting* iff it contains infinitely many accepting transitions *for each*  $\Gamma \in G$ . TBA can be seen as a special case of TGBA with  $|G| = 1$

## 2.3 Breakpoint Automaton

We want to define *slim GFM<sup>1</sup> Büchi automaton* (slim automaton) through its construction which is based on breakpoint construction.

**Construction** Let us fix a Büchi Automaton  $\mathcal{A} \stackrel{\text{def}}{=} (\Sigma, Q, q_i, \Delta, \Gamma)$ . We want to construct a deterministic automaton  $\mathcal{D}$  such that  $L(\mathcal{D}) \subseteq L(\mathcal{A})$ . We denote  $3^Q$  by the set  $\{(S, S') \mid S' \subsetneq S \subseteq Q\}$  and  $3_+^Q$  by  $\{(S, S') \mid S' \subseteq S \subseteq Q\}$ . We define the notation for the transitions and accepting transitions as  $\delta, \gamma : 2^Q \times \Sigma \rightarrow 2^Q$  with

$$\delta : (S, a) \stackrel{\text{def}}{=} \{q' \in Q \mid \exists q \in S. (q, a, q') \in \Delta\} \text{ and}$$

$$\gamma : (S, a) \stackrel{\text{def}}{=} \{q' \in Q \mid \exists q \in S. (q, a, q') \in \Gamma\}$$

(? Let us note that  $\delta$  and  $\gamma$  are deterministic transitions.)

We define the raw breakpoint transition  $\rho_\Gamma : 3^Q \times \Sigma \rightarrow 3_+^Q$  as

$$\rho_\Gamma((S, S'), a) \stackrel{\text{def}}{=} (\delta(S, a), \delta(S', a) \cup \gamma(S, a))$$

We follow the set of reachable states (first set) and the states that are reachable while passing at least one of the accepting transitions (second set). The transitions of the breakpoint automaton  $\mathcal{D}$  follow  $\rho$  with an exception: they reset the second set to the empty set when it equals the first; the resetting transitions are accepting. The breakpoint automaton  $\mathcal{D} \stackrel{\text{def}}{=} (\Sigma, 3^Q, (q_i, \emptyset), \delta_D, \gamma_D)$  is defined such that, when  $\rho : ((S, S'), a) = (R, R')$ , then there are three cases:

1. if  $R = \emptyset$ , then  $\delta_B((S, S'))$  is undefined (or, if a complete automaton is preferred, maps to a rejecting sink),

---

1. Good for Markov decision processes [+zdroj]

2. else, if  $R \neq R'$ , then  $\delta_B((S, S'), a) = (R, R')$  is a non-accepting transition, and  $\gamma_d((S, S'), a)$  is undefined.
3. otherwise  $\delta_D((S, S')) = \gamma_D((S, S'), a) = (R, \emptyset)$  is an accepting transition.

??????? On the other hand, semi-deterministic automata decide superset of such language. We are going to define a few more transitions on top of breakpoint construction which allow us to construct slim automata that decide exactly the class of GFM languages.

## 2.4 Slim Automata Construction [separate chapter?]

Breakpoint automata constructed as presented in the previous section are not always equivalent to the input automaton.

In this section we define transitions  $\gamma_w, \gamma_p : 3^Q \times \Sigma \rightarrow 3^Q$  that promote the second set of a breakpoint construction to the first set as follows.

1. if  $\delta_S(S', a) = \gamma_S(S, a) = \emptyset$ , then  $\gamma_p((S, S'), a)$  and  $\gamma_w((S, S'), a)$  are undefined, and
2. otherwise  $\gamma_p : ((S, S'), a) = (\delta(S', a) \cup \gamma(S, a), \emptyset)$  and  $\gamma_w : ((S, S'), a) = (\delta(S', a), \emptyset)$

$\mathcal{S} \stackrel{def}{=} (\Sigma, 3^Q, (q_i, \emptyset), \Delta_S, \Gamma_S)$  is slim, when  $\Delta_S = \Delta_D \cup \Gamma_p$  is set of transitions generated by  $\delta_D$  and  $\gamma_p$ , and  $\Gamma_S = \Gamma_D \cup \Gamma_p$  is set of accepting transitions, that is generated by  $\gamma_D$  and  $\gamma_p$ .  $L(\mathcal{S}) = L(\mathcal{A})$  (proof in text with original definition)

Alternatively, similarly defined using  $\gamma_w$  instead of  $\gamma_p$ , automaton  $\mathcal{W} \stackrel{def}{=} (\Sigma, 3^Q, (q_i, \emptyset), \Delta_w, \Gamma_w)$  is slim a and  $L(\mathcal{S}) = L(\mathcal{A})$ . (no proof yet)

## 2.5 Slim Automaton Construction Generalized to TGBA

We want to construct a slim automaton from TGBA  $\mathcal{T} \stackrel{def}{=} (\Sigma, Q, q_i, \Delta, G)$ . One possibility is to *degeneralize*  $\mathcal{T}$  and to use previously mentioned

## 2. PRELIMINARIES

---

algorithm in section 2.3. Another way is to extend slim automaton construction to TGBA.

**extended slim construction** We need to make sure we go infinitely many times through each accepting subset  $g \in G$ . To achieve this, we will go through each subset one by one, using original algorithm. We will keep track of  $levels \stackrel{def}{=} \{0, 1, \dots, |G| - 1\}$  in the names of states. Let  $|G|$  be number of  $levels$  and  $i \in N, i < |G|$  the current level. At each level  $i$ , we look at  $i$ th subset of  $G$ . We use same steps as in classic breakpoint construction, but on each accepting transition the new state will be leveled up to  $(i + 1) \bmod |G|$ , otherwise the target state has the same level. Our new automaton simulates  $\mathcal{T}$ , as it accepts a word iff it cycles through all levels. If  $|G| = 0$ , we return a trivially accepting automaton

We can use the core of previous construction and just to extend it with levels. Let  $up(x) \stackrel{def}{=} (x + 1) \bmod |G|$

$$P := 3^Q \times levels \text{ (?nepotrebuju and } P_+ := 3_+^Q \times levels)$$

We define  $\gamma_i$  similarly like  $\gamma$ , we just use  $\Gamma_i$  instead of  $\Gamma$  and it allows us to easily define the raw generalized breakpoint transitions  $\rho_{\Gamma_i}$ : similarly as  $\rho_{\Gamma}$  using  $\gamma_i$  instead of  $\gamma$ .

The generalized breakpoint automaton  $\mathcal{D} = (\Sigma, 3^Q \times \mathcal{N}, (q_i, \emptyset, 0))$  is defined such that, when  $\delta_R: ((S, S', i), a) \rightarrow (R, R', j)$ , then there are three cases:

1. if  $R = \emptyset$ , then  $\delta_B((S, S', i))$  is undefined,
2. else, if  $R \neq R'$ , then  $\delta_B: ((S, S', i), a) = (R, R', i)$  is a non-accepting transition,
3. otherwise  $\delta_B, \gamma_B: \delta_B((S, S', i), a) = (R, \emptyset, up(i))$ .

$$\gamma_p: P \times \Sigma \rightarrow P$$

1. if  $\delta(S', a) = \gamma_i(S, a) = \emptyset$ , then  $\gamma_p((S, S', i), a)$  is undefined, and
2. otherwise  $\gamma_p: ((S, S', i), a) = (\delta(S', a) \cup \gamma_i(S, a), \emptyset, up(i))$ .

$\mathcal{S} \stackrel{def}{=} (\Sigma, P, (q_i, \emptyset, 0), \Delta_p, \Gamma_p)$  is slim, when  $\Delta_p$  is set of transitions generated by  $\delta_b$  and  $\gamma_p$ , and  $\Gamma_p$  is set of accepting transitions, that is generated by  $\gamma_b$  and  $\gamma_p$ .

## 3 Implementation

I have implemented the generalized construction of slim automata in both weak and strong version (2.4,2.5). I have also added option to create breakpoint automata (2.3).

### 3.1 Technologies/Tools

The implementation is inside seminator, which is implemented in C++17 builds on Spot library[24-mklokocka].

#### 3.1.1 Seminator

[zdroj mklokocka thesis] Seminator is a Linux command-line tool which can be run with the seminator command.

The tool expects the input automaton in the Hanoi Omega-Automata(HOA) format [25-mklokocka] on the standard input stream, but it can also read the input automaton from a file.

[EXAMPLE OF sDBA created by seminator?]

#### 3.1.2 Spot

ltl2tgba ...

autfilt ...

ltlcross ...

### 3.2 Create Slim Automata Using Seminator

By default, seminator creates sDBA. To create a slim automaton we need to add --slim option.

**Options** By default, --slim tries all reasonable combinations of options, optimizes the output and chooses an automaton with the smallest number of states.

### 3. IMPLEMENTATION

---

There are several options to specify how we construct the automata.

For example `seminator --slim --strong --optimizations=0 --via-tgba` generates output according to algorithm in 2.5. (Using `--via-tba` converts input to tba first) With automaton

[TBA (or TGBA?) INPUT EXAMPLE]

```
$ cat automaton.hoa | ./seminator --slim --strong --via-tgba --optimizations=0
```

we will get output

[SLIM OUTPUT EXAMPLE]

--slim to generate slim automata by, defaults to unoptimized, "strong"

slim algorithm

--weak use "weak"-slim algorithm instead

--best try weak and strong, optimize outputs with spot and choose the one with smaller automaton [delete and use as default]

(add --strong to generate just automata just by strong slim algorithm) [not implemented yet] neither --weak nor --strong specified try both, optimize and choose smaller result

--via-tba transform input automaton to tba (2.1) first

--via-tgba does not modify input automaton to tba.

neither --via-tba nor --via-tgba: try both options, choose smallest automaton

postprocess optimizations should be used be as a default option, use an option to disable

**Example** Transform automaton.hoa to a slim automaton.

```
$ cat automaton.hoa | ./seminator --slim
```

### 3.3 Implementation of Slim Automata inside Seminor

I have implemented the generalized slim construction and its options mentioned in previous section 3.2. Furthermore, I have added an option to create breakpoint automata.

There already was basis for breakpoint construction in seminor, inside class `bp_twa`. As we can see in sections 2.3 and 2.4, slim automata construction builds on breakpoint automata construction.

That allows us to simply extend the `bp_twa` class. We create class `slim` that inherits from `bp_twa`. In the `slim` class we build breakpoint automaton using `compute_successors` method. Then we extend the method by adding accepting transitions  $\gamma_p$ , respectively  $\gamma_w$  according to section 2.4, whenever we receive `--slim` option.

Then we extend main function to recognize our desired CLI options.

As `seminator` didn't offer a command line option to create a bp automaton, I have added one `--bp`, for comparison.

### 3.4 Testing and Verification

Implemented tests are basic, only language equivalence is checked. `ltlcross` and `ltl2tgba` tools are used. The tests use random LTL formulas that were already generated, the LTL formulas are transformed into automata in HOA format by `ltl2tgba`. Then the tool `ltlcross` cross-compares the automaton with `seminator --slim` with all supported [not yet] additional parameters.

Only `seminator --slim --strong --via-tba` (and with no optimizations) is proved, as it follows construction from [main source] which is proved.

### 3.5 How to Install Seminator

(jeste nevim kde tuto sekci dat, jestli ma mit tento nazev, co vsechno tady bude treba dat... no a jeste ro pak upravim podle toho jak to bude odevzdane v zipu)

To install the tool we need install `spot` and to run

```
autoreconf -i && ./configure && make.
```

### 3.6 Future of Implementation

Implementation: python bindings, optimizations of `slim` construction (especially from TGBA)

### 3. IMPLEMENTATION

---

tests/verification: There should be another kind of tests - to check if our slim automata simulate the input automata (so the GFM property is not broken)

Subject of following research, that is out of scope of this thesis, could be to verify if spot optimizations do not break the simulation property.



## 4 Evaluation

Evaluation part builds on seminotor-evaluation. We compare amount of states of output automata.

3x3 tabulka, 3 sloupec best of, pocety minimalnich automatu 2x2 tabulka.. celkovy cas v tabulce 3x3

### 4.1 Post-Optimized

	weak	strong	best
via tba	cell1	cell2	cell3
via tgba	cell4	cell5	cell6
best	cell7	cell8	cell9

### 4.2 weak/strong

#### 4.2.1 Comparisons among Unoptimized options

#### 4.2.2 Comparisons among Post-Optimized options

2.4

### 4.3 via tba/no change

#### 4.3.1 Comparisons among Unoptimized options

#### 4.3.2 Comparisons among Post-Optimized options

(2.5)

### 4.4 Slim automata produced seminotor ePMC

prvne tba zkusit pro epmc

### 4.5 Compare with semi-deterministic Automata

### 4.6 To Put Somewhere in This Chapter

If the automata optimizations by spot's `autfilt` tool break the simulation property, the results in following Evaluation chapter are pointless, as they are built on such assumption.

## 5 Conclusion

