Masaryk University
Faculty of Informatics

Transformation of Nondeterministic Büchi Automata to Slim Automata

Bachelor's Thesis

Pavel Šimovec

Brno, Spring 2021

MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Transformation of Nondeterministic Büchi Automata to Slim Automata

BACHELOR'S THESIS

Pavel Šimovec

Brno, Spring 2021

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

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: doc. RNDr. Jan Strejček, Ph.D.

Acknowledgements

I am extremely grateful to my supervisors František Blahoudek and Jan Strejček, for their consultations, invaluable advice, and support. Thank you for the opportunity to work on an exciting topic.

My gratitude extends to thank Ernst Moritz Hahn for his advice, providing us ePMC tool, and adding a requested feature that allowed me to compare the tools. I would also like to thank Mateo Perez for providing us Mungojerrie tool before its release and willingness to answer any questions about it.

I want to thank my aunt for her encouragement and support during my studies. Lastly, I would like to express my gratitude to the rest of my family and friends for their support.

Abstract

The thesis describes *slim Büchi good for Markov decision processes automata* construction and extends it for *generalized Büchi Automata*. The construction is implemented in a tool called *Seminator*. Slim automata constructed by Seminator are compared to other automata produced by different tools. The automata are compared by the number of states, construction time, and learning speed when used for reinforcement learning in the *Mungojerrie* tool.

Keywords

slim automata, Büchi Automata, GFM-automata, Seminator

Contents

1 Introduction

Büchi automaton is a finite machine over infinite words. It has been a topic of research for almost 60 years. There were discovered various kinds of similar machines with different properties and use cases. Non-deterministic Büchi automata, in general, are not well suitable for model checking or reinforcement learning. However, we can construct non-deterministic Büchi automata with a special property – good for Markov decision processes (GFM) [1], that makes the automata suitable. We will focus on slim automata [1]. Slim automata are specially constructed Büchi automata. This kind of automaton was defined by its construction and is good for MDP [1]. We implement the proposed algorithm and its second variant that we call weak [2]. We introduce the algorithm for generalized Büchi automata. Then we evaluate the resulting size of automata, and we compare it with a different tool that can create slim automata and other kinds of automata. In the end, we investigate the impact of slim automata from Seminator on reinforcement against GFM automata created by other tools.

The Preliminaries chapter defines Büchi automaton and generalized Büchi automaton. The next chapter defines breakpoint automaton and slim automaton as described in our primary source [1] and the weak variant. In the fourth chapter, we extend the algorithm to generalized Büchi automata. Fifth chapter describes implementation of mentioned slim automata inside Seminator [3][4][5] tool. In the sixth chapter, we compare the resulting automata size (number of states). First, we compare the size internally among implemented options. Then we compare it with ePMC [6] - another tool implementing slim automata. We finish the chapter by comparing different kinds of automata. In the last chapter, we use our automata in benchmarks of reinforcement learning tool Mungojerrie[7], and we compare learning speed on it with original automata in provided benchmarks with original benchmark automata and two types of automata internally supported by Mungojerrie (slim automata by ePMC and Limit-deterministic Büchi automata by ltl2ldba [8]).

2 Preliminaries

This chapter defines a Büchi automaton and a generalized Büchi automaton.

An *alphabet* Σ is a finite set of *letters*, an *ω-word* $w \in \Sigma^\omega$ is an infinite sequence of letters, and an *ω-language* $L \subseteq \Sigma^\omega$ is a set of ω-words.

2.1 Büchi Automaton

A Büchi automaton is a theoretical finite-state machine used to define ω-languages. It decides which infinite words (ω-words) belong to its language.

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

- Σ is a non-empty finite *alphabet*,
- Q is a non-empty finite set of *states*,
- $q_i \in Q$ is the initial state of $\mathcal{A}$.
- $\Delta \subseteq Q \times \Sigma \times Q$ is a set of *transitions*.
- $\Gamma \subseteq \Delta$ is a set of *accepting transitions*.

Intuitivelly, a transition (s, a, t) directionally connects the states s and t with the letter a.

By convention, we use capital greek letters to denote sets of transitions. For convenience, we also define for each set of transitions Δ a function δ (denoted by the corresponding small greek letter). We define the function $\delta\colon (q, a) \overset{\text{def}}{=} \{q' \in Q \mid (q, a, q') \in \Delta \wedge q \in Q\}$. Δ can be defined by δ when initial state q_i is defined.

A *run* r of $\mathcal{A}$ over an ω-word $w = w_0 w_1 w_2 \ldots$ is an infinite sequence of transitions $r \overset{\text{def}}{=} t_0 t_1 \ldots \in \Delta^\omega$, where $t_k = (q_k, w_k, q_{i+1})$, such that $q_0 = q_i$. A run of $\mathcal{A}$ is *accepting* if and only if it contains infinitely many accepting transitions from Γ.

Finally, we define the *language* $L(\mathcal{A}) \subseteq \Sigma^\omega$ recognized by the automaton $\mathcal{A}$. An ω-word $w \in \Sigma^\omega$ belongs to $L(\mathcal{A})$ if and only if 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 $G \subseteq 2^{\Delta}$ 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 $|G| = 1$.

3 Slim Automata Construction

This chapter defines *slim Büchi automaton* (slim automaton) in 2 variants - *strong* and *weak*. Slim automaton is defined through its construction, which is based on breakpoint construction.

3.1 Breakpoint Automaton

BP automata are constructed from BA, but their language is only a subset of the language from the original BA.

Construction Let us fix a Büchi Automaton $\mathcal{A} \stackrel{\text{def}}{=} (\Sigma, Q, q_i, \Delta, \Gamma)$.

We start with some notation. By 3^Q we denote the set $\{(S, S') \mid S' \subsetneq S \subseteq Q\}$ and by 3^Q_+ we denote $\{(S, S') \mid S' \subseteq S \subseteq Q\}$.

For convenience we introduce functions by sets of transitions, we define the function $\delta \colon 2^Q \times \Sigma \to 2^Q$ as $\delta \colon (S, a) \stackrel{\text{def}}{=} \{q' \in Q \mid (q, a, q') \in \Delta \wedge q \in S\}$. We define $\gamma : 2^Q \times \Sigma \to 2^Q$ analogously from Γ as $\gamma \colon (S, a) \stackrel{\text{def}}{=} \{q' \in Q \mid (q, a, q') \in \Gamma \wedge q \in S\}$.

With δ and γ, we define the raw breakpoint transition $\rho_\Gamma \colon 3^Q \times \Sigma \to 3^Q_+$ as

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

The first set follows the set of reachable states in the first set and the states that are reachable while passing at least one of the accepting transitions in the second set. The transitions of the breakpoint automaton $\mathcal{D}$ follow ρ with an exception: they reset the second set to the empty set when it equals the first; the resetting transitions are accepting. Formally, the breakpoint automaton $\mathcal{D}$ is $\stackrel{\text{def}}{=} (\Sigma, 3^Q, (q_i, \emptyset), \Delta_D, \Gamma_D)$ where Δ_D and Γ_D are defined as follows.

1. $((S, S'), a, (R, R')) \in \Delta_D$ if $\rho_\Gamma((S, S'), a) = (R, R')$ where $R' \subsetneq R$
2. $((S, S'), a, (R, \emptyset)) \in \Delta_D$ and $((S, S'), a, (R, \emptyset)) \in \Gamma_D$ if $\rho_\Gamma((S, S'), a) = (R, R)$
3. No other transitions are in Δ_D and Γ_D

Figure 3.1 shows application of this construction. The example demonstrates that $L(\mathcal{D}) \subseteq L(\mathcal{A})$ as the construction did not generate any accepting transition. Therefore original $L(\mathcal{A}) = \{a^\omega\}$, but $L(\mathcal{D})$ is empty.

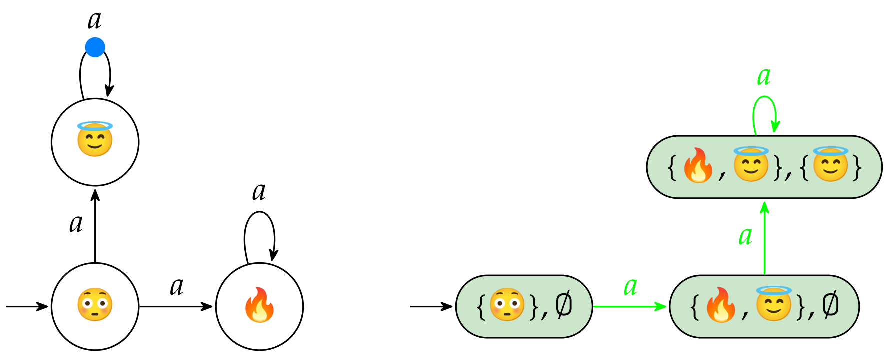

Figure 3.1: A Büchi Automaton $\mathcal{A}$ (left) and a breakpoint automaton $\mathcal{D}$ for $\mathcal{A}$ (right).

3.2 Slim automata

Slim automata are BP automata enriched with additional transitions. As a result, they are non-deterministic, Good for Markov decision processes [1] and equivalent to the input automata. In this section we define transition function for *strong slim* (γ_p)[1] and *weak slim* (γ_w) automata, $\gamma_w, \gamma_p : 3^Q \times \Sigma \to 3^Q$, that promote the second set of a breakpoint construction to the first set as follows.

1. if $\delta(S', a) = \gamma(S, a) = \emptyset$, 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 $\gamma_w : ((S, S'), a) = (\delta(S', a), \emptyset)$

$\mathcal{S} \stackrel{\text{def}}{=} (\Sigma, 3^Q, (q_i, \emptyset), \Delta_S, \Gamma_S)$ is (strong) slim, when Γ_p is generated by γ_p, then $\Delta_S = \Delta_D \cup \Gamma_p$ is set of transitions, and $\Gamma_S = \Gamma_D \cup \Gamma_p$ is set of accepting transitions. $L(\mathcal{S}) = L(\mathcal{A})$. The equivalence was proven in [1].

Alternatively, similarly defined using γ_w instead of γ_p, automaton $\mathcal{W} \overset{\text{def}}{=} (\Sigma, 3^Q, (q_i, \emptyset), \Delta_W, \Gamma_W)$ is (weak) slim when Γ_w is generated by γ_w, then $\Delta_W = \Delta_D \cup \Gamma_w$ is set of transitions, and $\Gamma_W = \Gamma_D \cup \Gamma_w$ is set of accepting transitions. $L(\mathcal{W}) = L(\mathcal{A})$ (proof would go similarly like the one for strong slim).

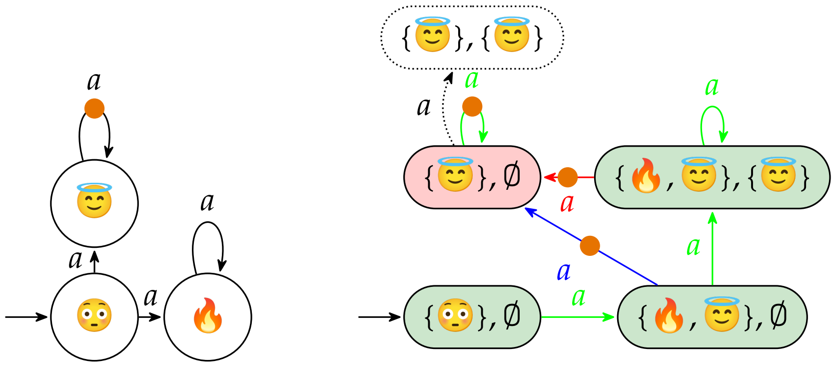

Figure 3.2: Slim automaton (right) and the original Büchi automaton from Figure 3.1(left). The image highlights underlying breakpoint construction by green color (as in Figure 3.1). A blue transition is created only by strong slim transitions. State and transition highlighted by red color would be created directly by both weak and strong slim transitions. Let us note that state {{😇},{😇}} is not part of produced automaton. It visualizes reset from breakpoint construction, step 2, which is applied even when the source state was created by slim transitions.

4 Slim Automaton Construction Generalized to TGBA

In this chapter, we discuss slim automata equivalent to a TGBA $\mathcal{T} \stackrel{\text{def}}{=} (\Sigma, Q, q_i, \Delta, G)$. One possibility is to *degeneralize* $\mathcal{T}$ and to use the previously mentioned algorithm in section 2.3. In the rest of this chapter, we introduce a direct construction of slim TGBA equivalent to $\mathcal{T}$.

Extended slim construction Intuitively, we will simulate the original automaton by checking its accepting conditions one by one. In the original automaton, have to go through an accepting transition of each accepting condition $g \in G = \{G_0, G_1, \dots, G_k\}$ infinitely many times. In the new automaton, we have just one accepting condition and a layer for each original accepting condition. Going through original accepting transitions of the layer that we are looking up promotes us to another layer. From the last layer, we get back to the first layer. Only the transitions that move us to layer up are accepting. As we check all accepting conditions of the original automaton, the new automaton will be equivalent to the original one. We will use the word "levels" instead of layers in the following definitions.

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 the original algorithm. We will keep track of *levels* $= \{0, 1, \dots, |G| - 1\}$ in the names of states. Let $|G|$ be number of *levels* and $i \in \mathbb{N}, i < |G|$ the current level. At each level i, we look at ith subset of G. We use the 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 if 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 $(S, S') \in 3^Q$ and let $i \in$ *levels*, by P we denote a state $P = (S, S', i)$.

We define γ_i from Γ_i for all $i \in levels$ in the same way we did for γ from Γ. We use raw breakpoint transitions ρ_Γ from tba breakpoint construction in Section 3.1.

Let $up(x) = (x + 1) \bmod |G|$.

The generalized breakpoint automaton $\mathcal{D} = (\Sigma, 3^Q \times \mathcal{N}, (q_i, \emptyset, 0), \Delta_B, \Gamma_B)$ is defined by breakpoint transitions (δ_B generates Δ_B and γ_B generates Γ_B) as follows.

When $\rho_\Gamma\colon ((S, S'), a) \to (R, R')$, then for all $i \in levels$:

For breakpoint transitions there are three cases:

1. if $R = \emptyset$, then $\delta_B(P, a)$ 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 $\gamma_B(P, a) = \delta_B((S, S', i), a) = (R, \emptyset, up(i))$.

For slim transitions there are two additional cases:

1. if $\delta(S', a) = \gamma_i(S, a) = \emptyset$, then $\gamma_p((S, S', i), a)$ is undefined, and
2. otherwise $\gamma_p((S, S', i), a) = \gamma_p((S, S', i), a) = (\delta(S', a) \cup \gamma_i(S, a), \emptyset, up(i))$. (Alternatively, for a weak slim automaton we do not include transitions $\gamma_i(S, a)$)

$\mathcal{S} \stackrel{\text{def}}{=} (\Sigma, (S, S', i), (q_i, \emptyset, 0), \Delta_S, \Gamma_S))$ is slim, when $\Delta_S = \Delta_B \cup \Delta_p$, generated by δ_B and γ_p, and $\Gamma_S = \Gamma_B \cup \Gamma_p$ is set of accepting transitions, that is generated by γ_B and γ_p.

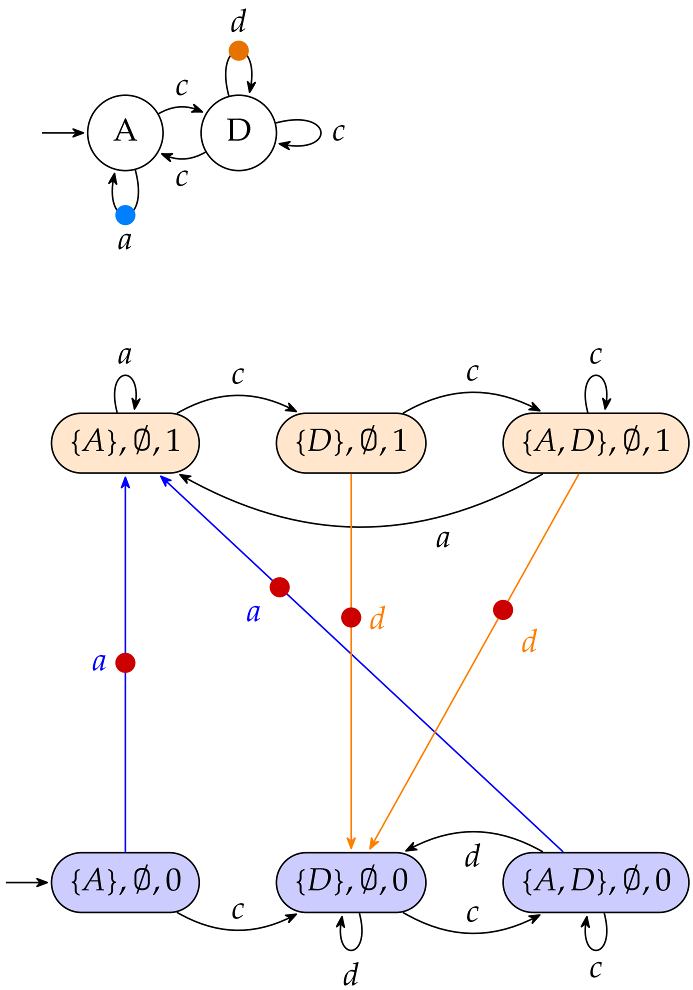

Figure 4.1: The original TGBA (top) and slim automaton with colored states emphasizing different levels (bottom).

5 Implementation

I have implemented the generalized construction of slim automata in both weak and strong version (3.2) (4). I have also added option to create breakpoint automata (3.1).

5.1 Technologies/Tools

The implementation is inside Seminator, which is implemented in C++17 builds on Spot library.

5.1.1 Seminator

Seminator is a Linux command-line tool which can be run with the `seminator` command. The tool transforms transition-based generalized Büchi automata (TGBAs) into equivalent semi-deterministic automata. [3][4][5]

The tool expects the input automaton in the Hanoi Omega-Automata (HOA) format [9] on the standard input stream, but it can also read the input automaton from a file.

5.1.2 Spot

”Spot is a C++ library with Python bindings and an assortment of command-line tools designed to manipulate LTL and ω-automata in batch.” [10, Abstract]

Relevant spot tools:

ltl2tgba ”translates LTL/PSL formulas into generalized Büchi automata,or deterministic parity automata” [10]

autfilt ”filters, converts, and transforms ω-automata” [10]

ltlcross ”cross-compares LTL/PSL-to-automata translators to find bugs” [10]

5.2 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.

Example 1 Transform automaton.hoa to a slim automaton.

```
$./seminator --slim -f automaton.hoa
```

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

For example `seminator --slim --strong --optimizations=0 --via-tba` generates output according to algorithm in 3.2. (Using `--via-tba` converts input to tba first).

Example 2 Transform automaton.hoa to unoptimized strong slim automaton

```
$./seminator --slim --strong --via-tgba --
    postprocess=0 -f automaton.hoa
```

`--slim` to generate slim automaton

`--weak` use only weak slim algorithm

`--strong` use only strong slim algorithm

Neither weak nor strong option specified - try both options and choose the one with a smaller automaton.

`--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 the smallest automaton.

Postprocess optimizations are enabled by default.

5.3 Implementation of Slim Automata inside Seminator

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

There already was a basis for breakpoint construction in Seminator, inside class `bp_twa`. As we can see in sections 3.1 and 3.2, slim automata construction builds on breakpoint automata construction.

That allows us to 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 γ_p, respectively γ_w according to section 4, whenever we receive `--slim` option.

Then the main function is extended to recognize our desired CLI options.

As Seminator did not offer a command-line option to create a breakpoint automaton, I have added the option `--bp` for comparison.

5.4 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. Ltl2tgba transforms the LTL formulas into automata in HOA format. Then the tool ltlcross cross-compares the automaton with *seminator –slim* with all supported additional parameters.

Only `seminator --slim --strong --via-tba --optimizations=0` is proved, as it follows construction from [1] which is proved.

5.5 Future of Implementation

Implementation: It would be good to optimize slim construction (especially from TGBA).

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

The subject of the following research, which is out of the scope of this thesis, could be to verify if Spot's optimizations do not break the simulation property.

6 Evaluation of automaton size

The evaluation part builds on seminator-evaluation. We compare the number of states of output automata on two datasets. The first dataset is 20 literature formulas; the second dataset is 500 LTLs that were randomly generated. We use a 120s timeout for each tool.

The first section starts with an internal comparison of slim automata created by Seminator. The second section compares these automata against ePMC[6], which is another tool producing slim automata. The third section compares slim automata against ldba and semi-deterministic automata. Ldba automata are produced by ltl2ldba [8], semi-deterministic automata that are produced by Seminator, and with ltl3tela[11]. Let us note that from mentioned types of automata, only semi-deterministic automata do not promise GFM property.

6.1 Slim automata produced by Seminator

In this section we compare automaton size generated by `seminator --slim`. We compare weak against slim and via-tba against via-tgba.

6.1.1 Comparisons among Unoptimized Configurations

In this subsection we compare base unoptimized seminator options.

In Table 6.1.1 strong slim automata have more states than weak ones, as expected because strong slim automata add more accepting transitions, which can create new states. Transformation of the automata to TBA first yields smaller automata. This might be caused by Spot having a well-optimized algorithm for degeneralization. The slim algorithm for TGBA proposed in this paper is naive, without any optimizations, and it degeneralizes the automaton during the process.

Table 6.1: Slim automata on both datasets without any post-processing (no timeouts).

seminator	weak		strong	
literature	size	time(s)	size	time(s)
via tba	1095	4	1112	5
via tgba	1122	4	1147	4

seminator	weak		strong	
random	size	time(s)	size	time(s)
via tba	17567	59	18764	57
via tgba	19320	58	20789	58

6.1.2 Post-Optimized

In this subsection we post-optimize results using `autfilt` tool.

GFM automata are closed under simulations [1, Section 3.1]. We are confident that spot's `autfilt` does not break the GFM property.

Table 6.2: In this table we compare all possible post-optimized combinations of parameters. By default `seminator --slim` tries all combination, runs optimizations, and then chooses the smallest automaton (best/best). For this experiment only 4 base options were computed directly through Seminator. Other ones were computed from obtained data.

seminator	weak		strong		best	
literature	size	time[s]	size	time[s]	size	time[s]
via tba	551	219	370	185	370	404
via tgba	588	210	408	174	402	384
best	551	429	370	359	365	788
seminator	weak		strong		best	
random	size	time[s]	size	time[s]	size	time[s]
via tba	8923	443	7404	476	7219	919
via tgba	10130	654	8500	591	8247	1245
best	8751	1097	7285	1067	7088	2164

As Table 6.2 shows, from 4 base options, after applying post-optimizations, strong slim algorithm surpasses a weak one by automaton size, even if it has worse results without the post-optimizations. Degeneralizing the automata as a first step still has smaller results. From 4 base options, the strong slim algorithm via-tba creates the smallest automata on average. Transforming input automata to tba first creates results that are close to the best ones. There were no timeouts in the literature dataset. In the random dateset, for the base options there were two timeouts for `--via-tba` and four timeouts for `--via-tgba`.

Strong against weak slim automata Let us focus on automaton size differences between weak and strong slim automata.

Scatter plot 6.1 reveals that a strong slim automaton is smaller in the majority of cases, but there are also cases where a weak slim automaton is smaller than the strong one.

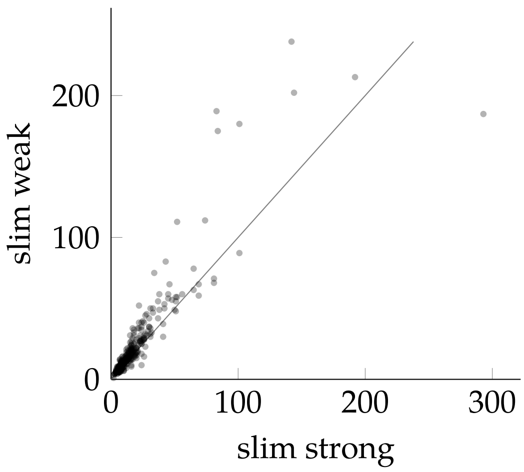

Figure 6.1: Scatter plot of strong against weak slim automata, equal values excluded.

Comparing minimal hits for weak and strong automata in Table 6.3 confirms that strong slim automata lead in unique minimal hits. However, weak slim automata have some unique minimal hits too, so it makes sense to try both options and choose the better one.

Table 6.3: Minimal hits of strong and weak slim automata.

literature	unique minimal hits	minimal hits
weak	4	9
strong	11	16
random	**unique minimal hits**	**minimal hits**
weak	68	202
strong	296	430

Via-tba against via-tgba Let us note that 13/20 formulas from literature and 391/500 formulas from the random dataset create automata that are already TBA.

Plot 6.2 density is low, as many of the results are the same (the majority of the automata were already tba). Via-tba has mostly better results, but there are examples where via-tgba outputs automaton twice smaller automaton. Table 6.4 confirms that via-tba has better results. Via-tgba has only 9 minimal hits compared to 91 from via-tba.

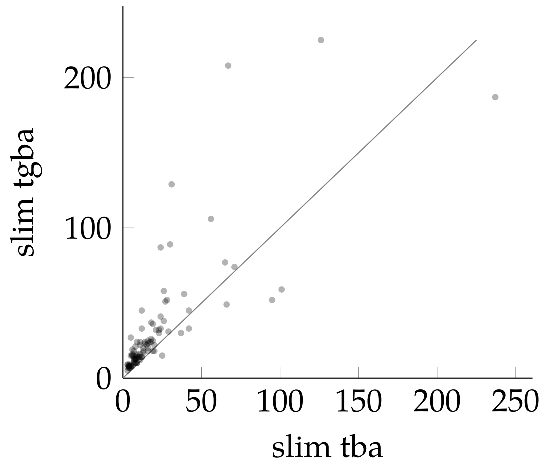

Figure 6.2: Scatter plot via-tba against via-tgba, equal values excluded.

Table 6.4: Minimal hits of via-tba against via-tgba.

literature	unique minimal hits	minimal hits
via-tba	7	20
via-tgba	0	13
random	unique minimal hits	minimal hits
via-tba	91	489
via-tgba	9	407

6.2 Slim Automata Produced Seminator versus ePMC

We can create slim automata using different tool called ePMC.

At first, we compare the best working basic parameters (parameters that try only one option) of each tool to create the smallest automata to obtain a fair time comparison. As Table 6.5 shows, Seminator creates smaller automata and creates them way faster. If we don't count timeouts and include Spot optimizations, Seminator runs approximately 25x faster than ePMC on the random dataset (without post-optimizations, Seminator is 90x faster on the dataset). In Figure 6.3 we can see, that ePMC has some better hits.

Table 6.5: EPMC acc stands for option, that uses accepting transitions whenever possible. It had slightly better results than other ePMC options. 120s timeouts are included in total time, however sum of automaton size is computed only from automata that were successfully created by both tools.

literature	size	time[s]	timeouts
ePMC acc	650	178	0
Seminator tba strong	436	151	0
random	size	time[s]	timeouts
ePMC acc	9643	5146	8
Seminator tba strong	7032	405	2

The section compares the smallest automata of each tool to see how smaller automata get by combining these tools.

As Figure 6.4 shows, Seminator slim has most of the best results, and the difference is even more significant than the single option comparison in Figure 6.3. Combining Seminator and ePMC for best automata did not bring much better results. On the random dataset total size of automata from Seminator slim is 7133. If we combine Seminator slim and ePMC, we get a total size of 7060, which is not a significant difference.

Then this section continues with comparison of minimal hits. Table 6.6 shows that by unique minimal hits of these tool ePMC has 35 unique minimal hits compared to 344 of Seminator. But as we can see from scatter plot 6.4, the size difference isn't that high.

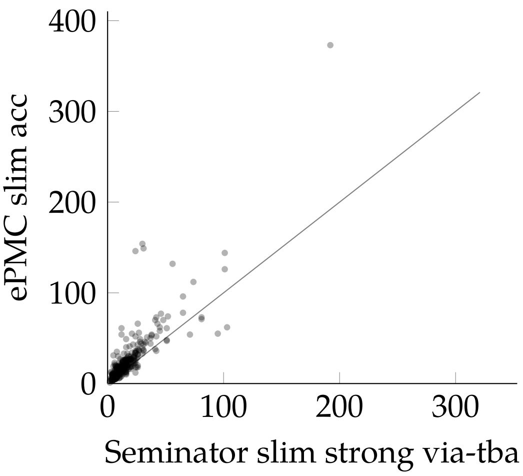

Figure 6.3: Scatter plot comparing sizes of ePMC acc and Seminator tba strong.

Table 6.6: Comparison showing how many times tool got smallest or uniquely smallest automata.

literature	unique minimal hits	minimal hits
ePMC	1	5
Seminator	15	19
random	unique minimal hits	minimal hits
ePMC	35	155
Seminator	344	464

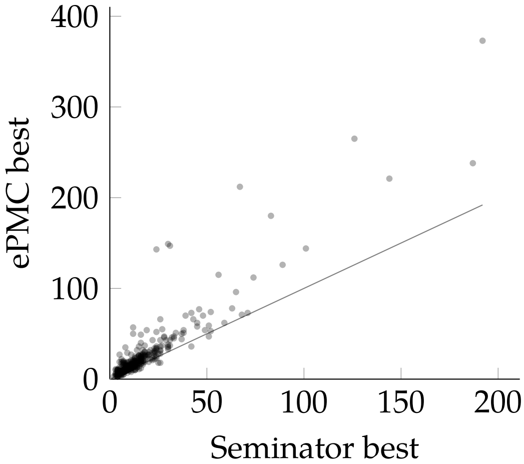

Figure 6.4: Scatter plot comparing smallest optimized automata generated by each tool.

6.3 Compare with Different Kinds of Automata

This section brings size comparison with diffent kinds of automata. It compares slim automata with ltl2ldba [8], default automata created by Seminator (semi-deterministic), and ePMC [6].

Results of table 6.7 show that semi-deterministic automata created by Seminator are the smallest on average among the compared tools. Ltl2ldba creates the smallest GFM automata, as semi-deterministic automata do not promise GFM property.

Table 6.7: Automata sizes after applying Spot's optimizations.

tool	literature	random
ePMC best	602	10570
ltl2ldba	331	4641
ltl3tela	161	3099
Seminator default	263	3896
Seminator slim best	431	7325

Now we compare `seminator --slim` with other tools using scatter plots.

Figure 6.5 shows that semi-deterministic automata are consistently smaller or just slightly bigger than slim automata. Figure 6.7 shows, that automata generated by ltl3tela are way smaller than slim automata. As Figure 6.6 shows, ltl2ldba creates consistently small automata (smaller than 40 states). However, its dominance over slim automata is not that consistent. The plot shows we can create reasonably smaller MDP automata on average using both tools and choosing the smallest automaton. Table 6.8 confirms that. For the literature dataset, the mixed tool even beats semi-deterministic automata from Seminator.

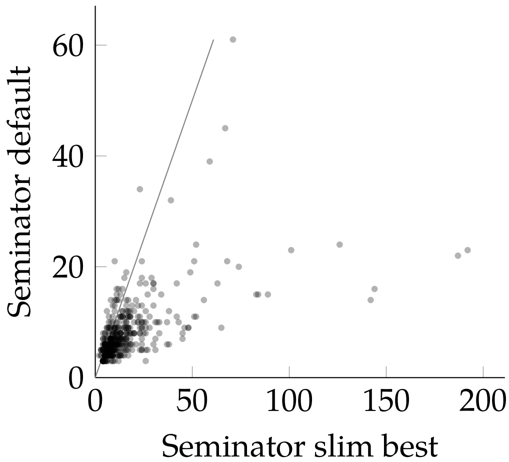

Figure 6.5: Comparison of semi-deterministic and slim automata created by Seminator.

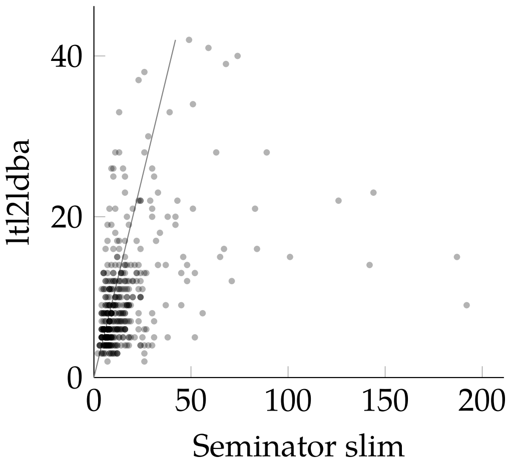

Figure 6.6: Slim automata from Seminator against automata from ltl2ldba.

Table 6.8: Automaton size comparison of Seminator slim + ltl2ldba combined tool against other tools.

tool	literature	random
ePMC best	602	10570
ltl2ldba	331	4641
ltl3tela	161	3099
Seminator default	263	3896
Seminator slim best	431	7325
Seminator slim + ltl2ldba best	262	4154

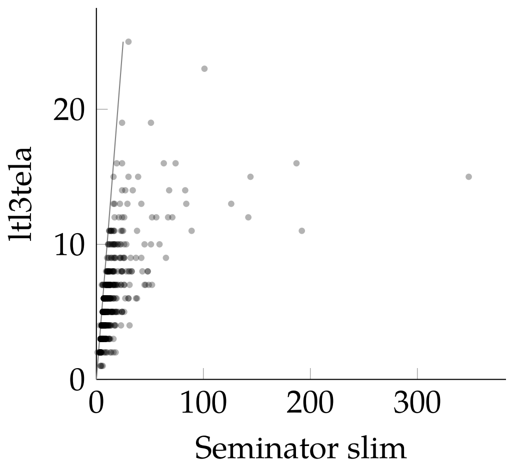

Figure 6.7: Slim automata from Seminator against automata from ltl3tela.

7 Mungojerrie Benchmarks

This chapter compares Seminator's slim automata on reinforcement learning tool Mungojerrie [7]. Reinforcement learning inside Mungojerrie has 2 phases. It has a learning phase with a given number of episodes and a model checking phase. The objective is defined by provided GFM automaton.[7]

The experiment uses benchmarks provided with the tool - *Examples*. Automata for the Examples are built in various ways. Some can even be handcrafted. Using LTL, which is provided as a name of automaton in each original benchmark, we create slim automata for comparison on benchmarks, as Mungojerrie can accept LTL and transform it using internally supported tools ltl2ldba and ePMC, or we can provide automaton that is GFM (the property is not checked by Mungojerrie).

Experiments are searching for the lowest necessary amount of episodes needed for reaching probability 1 to hit the objective in the model checking phase. Experiments run benchmarks ten times with pseudo-random seeds 0-9. If all runs end with success, the experiment computes the median and the average of results. A run can fail by timeout (600s) or by not reaching probability 1. We exclude uninteresting benchmarks, where all tools achieve the same result.

Table 7.1 shows that ltl2ldba has unique best results most often, but Seminator slim has some best hits too.

Examples have unclear origins, therefore Table 7.2 shows the results without the Examples. Ltl2ldba still has the best unique results most often, but Seminator gets closer to ltl2ldba. Compared to the previous table, Seminator now has more best averages and medians than ltl2ldba. The table also shows ePMC having a higher amount of second-best averages and medians than other tools.

Table 7.3 contains average episodes needed for a benchmark for each tool. It is visualized by the cactus plot in Figure 7.1. There we can see that ends of lines match to the number of failures in tables 7.1, 7.2.

Figure 7.2 shows the cactus plot of all benchmark runs. There we can observe that Seminator has the highest number of successful runs, therefore the lowest number of failures. It does not match the plot Figure 7.1. The reasoning follows: Looking at the number of failures on Table 7.2, both ePMC and ltl2ldba have one benchmark, which fails on all of its runs. Seminator has two benchmarks, which fail only on one run (by not achieving probability 1). Therefore Seminator succeeds at more runs than other tools in Figure 7.2.

Table 7.1: Results with Examples included

	Seminator	Examples	ePMC	Ltl2ldba
unique best average	5	7	1	12
unique best median	6	7	0	12
best average	11	12	7	13
second best average	9	5	9	6
best median	12	12	6	13
second best median	10	4	11	7
failures	4	7	3	4

Table 7.2: Results with Examples excluded

	Seminator	ePMC	Ltl2ldba
unique best average	9	1	13
unique best median	10	0	13
best average	14	6	13
second best average	5	14	4
best median	15	5	13
second best median	5	14	4
failures	2	1	2

Table 7.3: Table of average episodes needed to reach probability 1.

	Seminator	Examples	ePMC	Ltl2ldba
0	718.6	647.0	512.8	167.4
1	1.0	-	22.0	6.2
2	-	38.9	-	-
3	3620.4	3408.7	5763.8	1878.0
4	-	2171.2	-	-
5	5193.9	5193.9	-	4081.5
6	1.0	-	1.0	1.0
7	-	-	5906.1	1038.6
8	1310.4	1729.6	1156.8	983.8
9	718.6	-	512.8	167.4
10	3620.4	-	5763.8	1878.0
11	1.0	1.0	1.0	-
12	-	3236.3	5906.1	1212.8
13	1254.8	206.0	2788.7	5993.1
14	3095.3	4376.1	4376.1	-
15	1254.8	685.0	2788.7	6038.7
16	1.0	14.4	22.0	7.2
17	718.6	-	512.8	167.4
18	1187.2	320.5	562.4	324.5
19	124.5	124.5	124.5	595.0
20	1.0	12.2	1.0	8.4
21	1254.8	350.1	2788.7	5726.0
22	1250.7	1039.4	1250.7	1250.7
23	1.0	9.2	1.0	7.6
24	3620.4	6206.9	5763.8	1878.0
25	1254.8	299.2	2788.7	5842.1
26	1.0	9.0	22.0	6.4
27	718.6	-	512.8	167.4
28	683.8	638.8	618.6	11716.2
29	718.6	6448.8	512.8	167.4
30	1.0	14.4	22.0	6.4
31	19.3	13.0	84.1	19.5
32	1.0	1.0	1.0	8.8

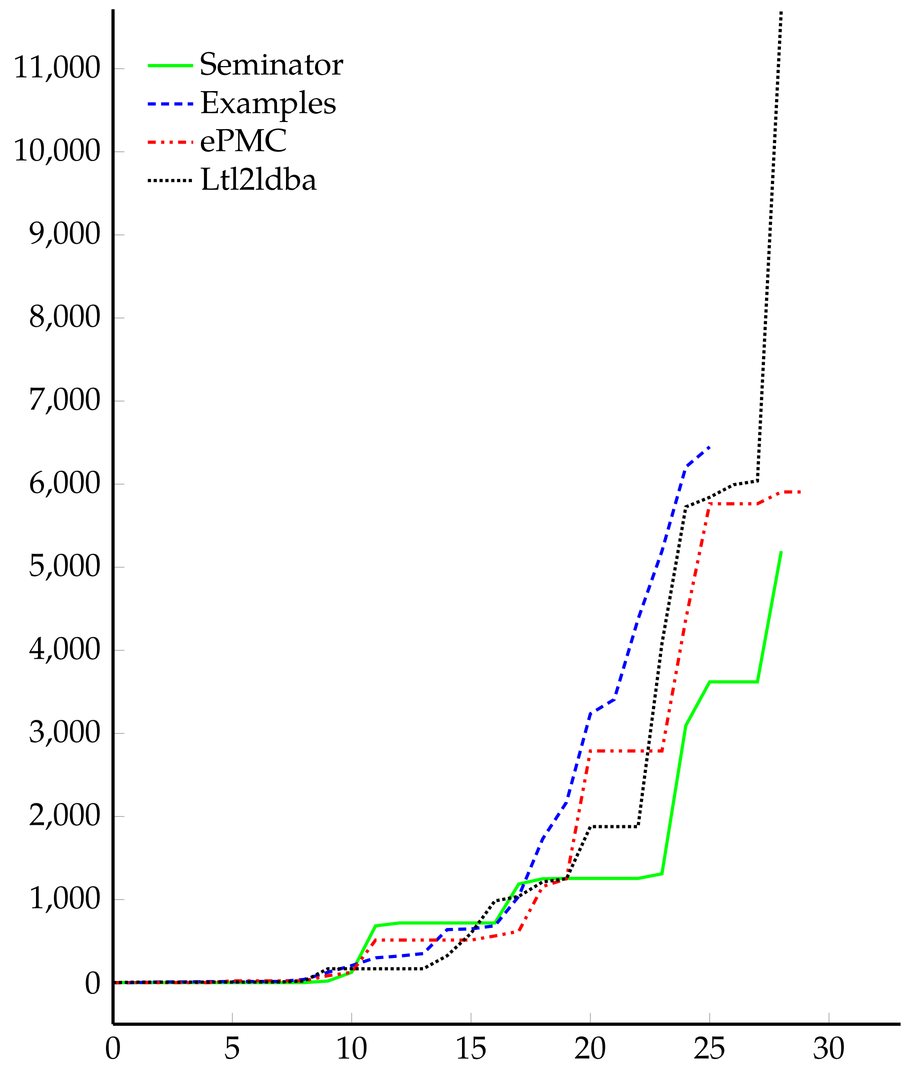

Figure 7.1: Sorted values for benchmark's average run for each tool by number of episodes (y-axis). Number of benchmarks is on x-axis.

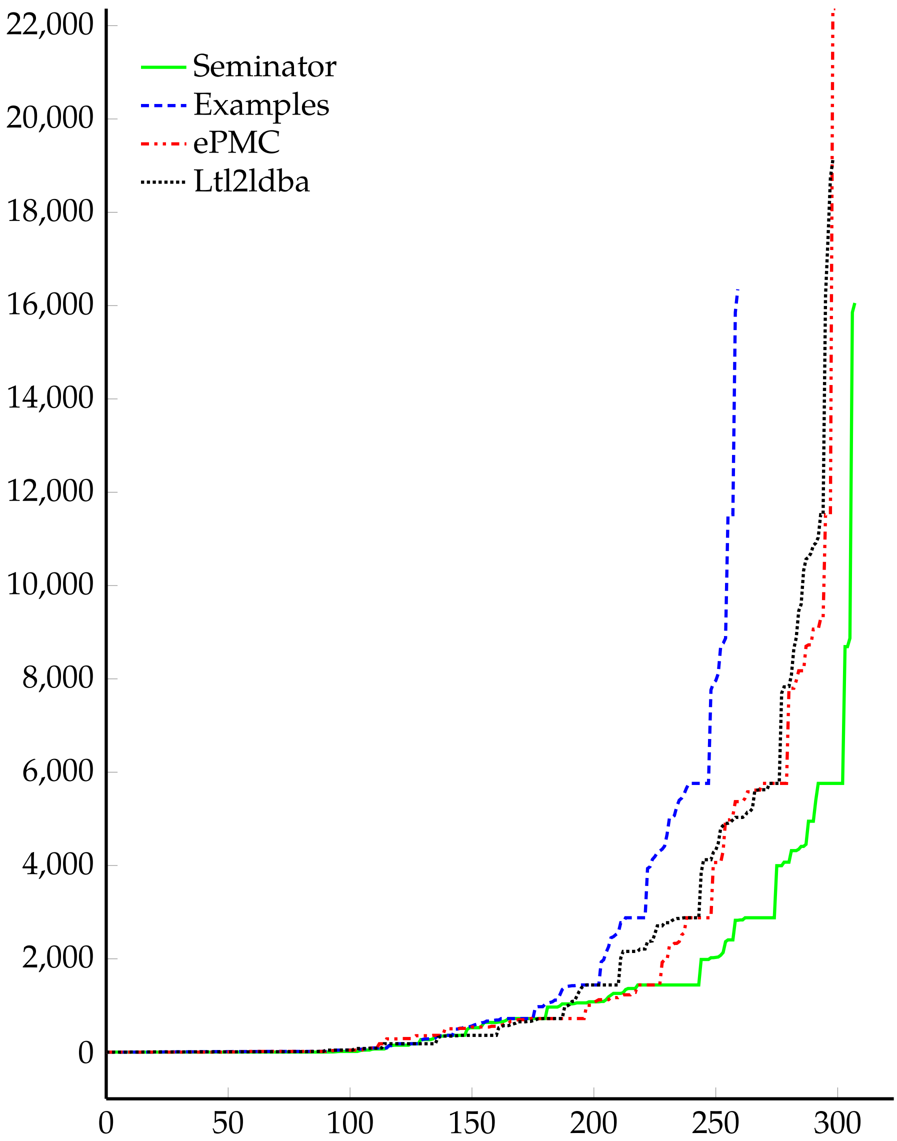

Figure 7.2: Sorted values for all of benchmark runs for each tool by number of episodes (y-axis). Number of benchmark runs is on x-axis.

8 Conclusions

I have described slim automata with their prerequisites (based on [1]) and their weak version. I have extended the construction to accept TGBA automata. I have also implemented the construction of slim automata inside Seminator, which was the primary goal. I have compared the automaton size of Seminator's slim automata internally among offered options and against other tools - ePMC, ltl2ldba, ltl3tela and Seminator's semi-deterministic automata. Seminator's slim automata are smaller than the slim automata from ePMC, and are created faster. I have ended up evaluating the performance of Seminator's slim automata on the recently released Mungojerrie tool and comparing it against its supported tools ltl2ldba and ePMC.

Future research could follow up by finding out which attributes of slim automata work the best for reinforcement learning and optimize the automata according to the results.

Appendices

A List of Electronic Attachments

As part of thesis, I have also submitted the following electronic attachment:

`seminator.zip` - Seminator which is released under the GNU GPL v3.0 license, extended by implementation of slim automata as described in Chapter 5.

Bibliography

1. HAHN, Ernst Moritz; PEREZ, Mateo; SCHEWE, Sven; SOMENZI, Fabio; TRIVEDI, Ashutosh; WOJTCZAK, Dominik. Good-for-MDPs Automata for Probabilistic Analysis and Reinforcement Learning. In: BIERE, Armin; PARKER, David (eds.). *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part I*. Springer, 2020, vol. 12078, pp. 306–323. Lecture Notes in Computer Science. Available from DOI: `10.1007/978-3-030-45190-5\_17`.
2. BLAHOUDEK, František [Private Communication]. 2021.
3. KLOKOČKA, Mikuláš. *Semi-Determinization of Omega-Automata* [*online*]. 2017 [cit. 2021-05-21]. Available also from: `https://is.muni.cz/th/c9v0s/`.
4. BLAHOUDEK, František; DURET-LUTZ, Alexandre; KLOKOČKA, Mikuláš; KŘETÍNSKÝ, Mojmír; STREJČEK, Jan. Seminator: A Tool for Semi-Determinization of Omega-Automata. In: EITER, Thomas; SANDS, David; SUTCLIFFE, Geoff (eds.). *Proceedings of the 21th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning* (*LPAR-21*). EasyChair Publications, 2017, vol. 46, pp. 356–367. EPiC Series in Computing. Available from DOI: `10.29007/k5nl`.
5. BLAHOUDEK, František; DURET-LUTZ, Alexandre; STREJČEK, Jan. Seminator 2 Can Complement Generalized Büchi Automata via Improved Semi-Determinization. In: *Proceedings of the 32nd International Conference on Computer-Aided Verification* (*CAV'20*). Springer, 2020, vol. 12225, pp. 15–27. Lecture Notes in Computer Science. Available from DOI: `10.1007/978-3-030-53291-8_2`.
6. HAHN, Ernst Moritz; LI, Yi; SCHEWE, Sven; TURRINI, Andrea; ZHANG, Lijun. IscasMC: A Web-Based Probabilistic Model Checker. In: *Nineteenth international symposium of the Formal Methods Europe association* (*FM*). Springer, 2014, vol. 8442, pp. 312–317. Lecture Notes in Computer Science.

7. HAHN, Ernst Moritz; PEREZ, Mateo; SCHEWE, Sven; SOMENZI, Fabio; TRIVEDI, Ashutosh; WOJTCZAK, Dominik. Omega-Regular Objectives in Model-Free Reinforcement Learning. In: VOJNAR, Tomáš; ZHANG, Lijun (eds.). *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2019, pp. 395–412. ISBN 978-3-030-17462-0.

8. SICKERT, Salomon; ESPARZA, Javier; JAAX, Stefan; KŘETÍNSKÝ, Jan. Limit-Deterministic Büchi Automata for Linear Temporal Logic. In: CHAUDHURI, Swarat; FARZAN, Azadeh (eds.). *Computer Aided Verification*. Cham: Springer International Publishing, 2016, pp. 312–332. ISBN 978-3-319-41540-6.

9. BABIAK, Tomáš; BLAHOUDEK, František; DURET-LUTZ, Alexandre; KLEIN, Joachim; KŘETÍNSKÝ, Jan; MÜLLER, David; PARKER, David; STREJČEK, Jan. The Hanoi Omega-Automata Format. In: KROENING, Daniel; PASAREANU, Corina S. (eds.). *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Springer, 2015, vol. 9206, pp. 479–486. Lecture Notes in Computer Science. Available from DOI: `10.1007/978-3-319-21690-4\_31`.

10. DURET-LUTZ, Alexandre; LEWKOWICZ, Alexandre; FAUCHILLE, Amaury; MICHAUD, Thibaud; RENAULT, Etienne; XU, Laurent. Spot 2.0 — a framework for LTL and ω-automata manipulation. In: *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*. Springer, 2016, vol. 9938, pp. 122–129. Lecture Notes in Computer Science. Available from DOI: `10.1007/978-3-319-46520-3_8`.

11. MAJOR, Juraj; BLAHOUDEK, František; STREJČEK, Jan; SASARÁKOVÁ, Miriama; ZBONČÁKOVÁ, Tatiana. ltl3tela: LTL to Small Deterministic or Nondeterministic Emerson-Lei Automata. In: CHEN, Yu-Fang; CHENG, Chih-Hong; ESPARZA, Javier (eds.). *Automated Technology for Verification and Analysis*. Cham: Springer International Publishing, 2019, pp. 357–365. ISBN 978-3-030-31784-3.