AUTOMATA *for* FORMAL METHODS:

LITTLE STEPS TOWARDS PERFECTION

František Blahoudek

PHD THESIS
corrected version
(June 14, 2019)

Faculty of Informatics
Masaryk University
Brno

March 2018

Acknowledgements

I will always remember my postgraduate years as an intensive period of my life, full of both amazing and tough experiences. It was a period of joy and constant personal growth. I could never finish my thesis without inspiration and support of many people around me. In the following paragraphs, I would like to express my gratitude to at least some of them.

First of all, I would like to thank my supervisor Jan Strejček. He is the one who showed me the beauty of automata more than ten years ago and who gave me the opportunity to spread the beauty among my students as a teacher. I especially value his trust and patience, and I am more than grateful for his method of supervising through careful, inspiring, and close collaboration. I like the way he writes, and I hope I learnt at least some bits from him. I enjoyed sharing my passions for chocolate, good drinks, running, and colorful automata with this great teacher and mentor. And what I appreciate the most is that I can call Jan my friend with whom I can discuss science, life, love, and jokes. I will never forget that we were able to experience good laughs even at 4 in the morning before deadlines. And I also have to mention the unforgettable travel experiences from our business trips; it was a pure pleasure to watch Jan falling to the river Okawango after he attempted to drive a Mokoro boat.

I was also honoured to have Mojmír Křetínský as a supervisor for a year. I am thankful to him for his attitude to me, his willingness to help, and for his support and care in difficult days.

Alexandre Duret-Lutz is our exceptional collaborator. He disclosed me how much can scientists profit from mutual collaboration and he also encouraged me to learn and explore new technologies and to develop own useful tools. My research would be much harder and less enjoyable without him and his work on Spot. The integration of Spot with Jupyter had an enormous impact on my performance and saved me a lot of precious time.

As an attendee of the MOVEP summer school, I had the unique opportunity to share a good time with Sven Schewe. I enjoyed our talks about automata and life, and I appreciate his notorious good mood and the willingness to share ideas.

I have shared my office with three inspiring colleagues and friends. Petr Novotný, starting his PhD two years before me, was always a good source of inspiration and good advice; he taught me to appreciate good rum and to enjoy preparing my presentations properly. Luboš Korenčiak is an infinite source of jokes and good mood and a great companion to travels; he taught me to procrastinate and to be open to people. Martin Jonáš never hesitates to share his opinion and good taste; he taught me to drink up to four cups of coffee a day and to be concerned with typography.

Our office is a part of the Formela lab, a place where I have been meeting many wonderful colleagues and friends. I especially enjoyed meeting Tomáš Babiak, Tom Brázdil, Marek Chalupa, Jakub Gajarský, Mirek Klimoš, Jéňa Krčál, Honza Křetínský, Kája Malá, Mikuláš Klokočka, Honza Obdržálek, Vojta Rujbr, Vojta Řehák, Mima Sasaráková, Marek Trtík, Dominik Velan, Martina Vitovská, and Táňa Zbončáková. I wish to meet them on many occasions in the future. I also hope to meet the friends from the ParaDiSe lab, who contributed to the positive environment in the school.

Teaching has been an enjoyable part of my studies and it often served as a source of energy for me. I am thankful to my students that brought fun and good mood into my lessons. I am also grateful to all people that helped me to bring TeachingLab into life. I especially enjoyed the cooperation with Ondráš Přibyla, who brought many new insights and views into my life. A great deal of my gratitude goes to Martin Ukrop who continues in my effort to improve the quality of the student's teaching at our university.

It would be hardly possible to survive the PhD studies without the support of friends. I would especially like to thank Dědek, Vojta Duba, Roman Klein, Michal Klivický, Tom Kocmi, Honza Kudr, Saša Kuckir, Lukáš Straka, and Michal Zeman for their help and the wonderful time we spent together. I would also like to thank Věrka Slezáková, who supported me in my difficult times and who taught me much about life. I feel exceptional gratitude to my friends from Instruktoři Brno for many memorable experiences,[1] personal growth, and fun they have brought into my life. But most of all I value the close friendships I found there, notably with Ďáblice, Entiro, Finn, Glum, Jitka, Lenka, Mýca, and Rissie.

I feel the deepest gratitude to my parents, Marie and František. They have always offered me a warm place to return to, unconditional support, empathy, and love. They have encouraged me to pursue my goals and they have always been curious about my various adventures not only from business trips. Beyond all of this, I thank them for teaching me not to forget about fun in my life. I have the great luck to have a broad, supportive family whose members have kind words for me when needed and they also never miss an opportunity to make fun of me. I would like to especially mention my brother Petr and my aunts Anča and Petra and thank them for being close to me. I have always enjoyed the company and smiles of other members of my family, namely Anduj, Eva, Fanda, Honza, Marie, Miluška, Petra, Táňa, Vašek, and Zuzka.

Fanda Blahoudek
Brno
March 2018

[1] They made my group carry a boat for more than five km through a forest at night, for example.

Abstract

As ω-automata are a convenient representation of languages of infinite words, they are widespread in the area of formal methods; many algorithms that analyze systems with infinite behaviours rely on ω-automata. The efficient algorithms for the intersection, union, and emptiness checking for various classes of ω-automata made them appealing for model checking of properties expressed as ω-regular languages or as formulae in (not only) Linear Temporal Logic (LTL).

On the contrary, determinization and complementation of ω-automata are notoriously difficult problems. This fact complicates usage of the automata-based methods that need deterministic automata[2] or inherently employ language difference or complementation of ω-automata.[3]

This dissertation approaches ω-automata and formal methods from various directions and presents several contributions towards perfect automata for formal methods. The presentation of the contributions is divided into three parts.

- The first part is tightly connected to the model checker Spin and nondeterministic Büchi automata. We investigate how different automata for one language can influence the performance of Spin and we bring several interesting observations and recommendations for LTL translators. Moreover, we introduce a method that enables the creation of automata that are suited for a particular verification task. The automata convey knowledge about the system to be verified; this knowledge sometimes helps to make the automata significantly smaller and to speed up the model checking.
- The second part of the thesis is dedicated to the translation of LTL into deterministic automata. We present an efficient translation of a fragment of LTL into automata with generalized Rabin acceptance condition. We also discuss other approaches to the translation and offer an extensive experimental comparison of available tools.
- The last part discusses semi-deterministic automata, which are automata that are deterministic in the limit. We develop an algorithm (and a tool) for semi-determinization of Büchi automata, and an efficient algorithm for complementation of these automata.

[2] like *model checking of probabilistic systems* or *synthesis of reactive systems*

[3] like *termination analysis* in the tool Ultimate Automizer

Contents

List of Figures

List of Tables

1 Introduction

Automata play an essential role in the history of computer science. In the 1960s and 1970s automata over finite words were seen as abstract machines that process inputs and accept or reject them. This kind of view was mainly driven by their application at that time – automata were used to build lexicographic analysers, parsers and compilers. Their primary purpose was to check *syntax*. With the development in formal methods, automata became a popular formalism used to describe *behaviours* and *specification*[1] of software and hardware systems; they became a data structure for representing sets of behaviours. Their popularity stems from the fact that automata allow efficient implementation of operations like union, intersection, and complement. Another appealing aspect of automata over words is their intuitive graphical representation.

Automata over infinite words (ω-words), also known as ω-automata, were introduced by Büchi in 1962 as a tool to prove the decidability of the monadic second-order logic with Presburger arithmetic.[2] An infinite word cannot be read to its end by an automaton and thus Büchi had to innovate the acceptance mechanism of automata. His solution was the following: *an ω-automaton $\mathcal{A}$ accepts an ω-word w if $\mathcal{A}$ can visit some accepting state infinitely often while reading w.* Automata with this kind of acceptance condition are nowadays named after Büchi and they are the most widely used type of ω-automata to these days. However, as we will discuss later, their acceptance mechanism is not powerful enough for some applications, and thus more acceptance conditions like Muller, Rabin, Streett, parity, and others were introduced.

Vardi and Wolper started amazing scientific progress in the area of ω-automata in 1986[3] when they realized that ω-automata are a natural choice as a data structure for methods that analyze systems with infinite behaviour.[4] ω-automata lie at the heart of many solutions of interesting problems from the area of formal methods ranging from *system monitoring* through *system analysis* and *verification* to *system synthesis*. Solutions to these problems are typically computationally hard and the computation time and memory consumption often hugely depend on automata used on the way. While ω-automata inherit the decidability properties of automata over finite words, some operations like determinization, complementation, etc. are substantially harder for ω-automata. The needs of efficient construction of practical ω-automata and efficient manipulation of ω-automata has driven the scientific progress to these days. This thesis confirms the previous statement and presents part of my contribution to the fascinating world of automata-theory, mostly motivated by practical needs of verification methods. In the next few paragraphs, we will discuss areas of automata theory touched by this thesis.

[1] specification in the form of a set of intended or erroneous behaviours

[2] Büchi (1962), "On a Decision Method in Restricted Second Order Arithmetic", [1].

[3] Vardi and Wolper (1986), "An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report)", [2].

[4] A print server or a controller of a power plant, for example.

A notable example of an ω-automata-based verification method is the automata-theoretic approach to model checking discussed in Chapter 3.

LTL translations. The inputs of a verification task are typically a *system* to be verified and its formal *specification*. The specification is often given as a formula of some modal logic. *Linear Temporal Logic (LTL)* is often the logic of choice as it allows to reason about the evolution of the system in time and thus can express many useful properties. For example, the natural expectation from a print server that *every print request is eventually processed* can be written as an LTL formula $\varphi = \mathsf{G}(\texttt{request} \implies \mathsf{F}\,\texttt{print})$. A standard step in verification is a *translation* of this formula into an ω-automaton that represents all behaviours that satisfy φ; see Figure 1.1 for a Büchi automaton for φ. As many chapters of the thesis are somehow related to the construction of ω-automata for LTL formulae, we will discuss LTL translations in more detail.

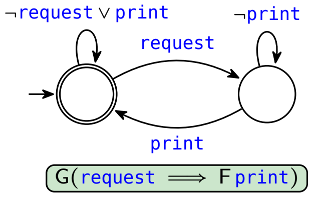


Figure 1.1: Büchi automaton $\mathcal{A}_\varphi$ for φ.

Every LTL formula φ can be translated to a *nondeterministic Büchi automaton (NBA)* $\mathcal{A}_\varphi$ with the number of states exponentially dependent on the size of φ. The translation of LTL into NBA is a well-studied problem. Scientists have already suggested many approaches to the translation. Evaluations show that no approach is superior to the others on its own, without further optimizations. Therefore, rewriting of the input formulae and reductions of the automata at different stages of the translation became the most powerful weapons in the battle for the best LTL-to-BA translator. The rapid development brought to the community translators like Spot and LTL3BA that are very efficient in practice, and they often avoid the exponential blow-up. Many experts, including authors of the mentioned tools, believe that there is not much hope for smaller NBA here. However, this is not the end of the story of LTL translations as we show in the next three paragraphs.

Some applications cannot be solved using NBA directly. For example, *controller synthesis for reactive systems*[5] is addressed by reduction to the problem of finding a winning strategy in a two-player game. The game is usually constructed from an ω-automaton for the specification, and we need a deterministic ω-automaton for this task.[6] Further, problems from the family of *model checking of probabilistic systems* are typically solved using deterministic ω-automata. How can we efficiently construct them? A natural choice is to take efficient translators of LTL to NBA and determinize the NBA we get for our formula. Let us discuss this option in more detail.

Determinization of ω-automata is substantially harder than the one of automata over finite words. For finite words, we have an efficient procedure known as the *powerset construction* that takes a nondeterministic automaton with n states and constructs an equivalent deterministic automaton with at most 2^n states.[7] This method is known to be tight and is well understood. In the world of Büchi automata, the powerset construction is not correct anymore; see Figure 1.2. The increase in complexity of a correct determinization is two-fold. First, deterministic Büchi automata are less expressive than their nondeterministic counterparts and thus we have to use some more complex acceptance condition. Second, for a Büchi automaton with n states we can build, using the tight upper bound on determinization,[8] a Rabin automaton

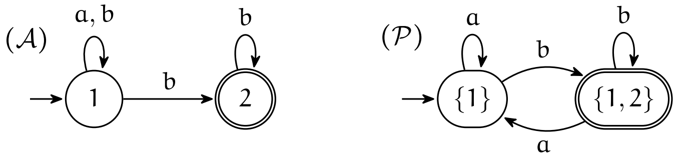


Figure 1.2: The automata $\mathcal{A}$ and $\mathcal{P}$ demonstrate that the powerset construction is not correct for ω-automata. The automaton $\mathcal{P}$ is the result of the powerset construction applied on $\mathcal{A}$. While $\mathcal{A}$ accepts all ω-words with only a finite number of as, $\mathcal{P}$ accepts all ω-words that have infinitely many bs (and possibly also infinitely many as).

[5] The problem of *controller synthesis for reactive systems* takes as input a specification φ, set of available actions of an *environment*, and set of available actions of a controller. While the actions of the environment are out of our control, we can control the actions of the controller. A solution to this problem is to automatically generate a controller that will react to the actions of the environment in a way that guarantees satisfaction of φ no matter what actions the environment performs.

[6] Alternatively, so-called *good-for-games* Rabin or parity automata do not need to be fully deterministic and still can be reduced effectively to a two-player game.

[7] Rabin and Scott (1959), “Finite Automata and Their Decision Problems”, [3].

[8] Schewe (2009), “Tighter Bounds for the Determinisation of Büchi Automata”, [4]; Colcombet and Zdanowski (2009), “A Tight Lower Bound for Determinization of Transition Labeled Büchi Automata”, [5].

with at most $(1.65n)^n$ states and 2^{n+1} accepting sets. If we aim for parity acceptance which is more suitable for solving games (and thus controller synthesis), we can have automata with at most $\mathcal{O}(n!^2)$ states and $2n$ priorities.

I would like to mention two approaches that researchers pursue to overcome the high complexity of ω-automata determinization. The first approach is a direct translation of LTL into various deterministic ω-automata. The second approach investigates new methods of solving model checking of probabilistic systems using ω-automata that are not fully deterministic, for example unambiguous or semi-deterministic[9] ω-automata. These methods brought us a new challenge of efficient translation of LTL into semi-deterministic automata, either directly or via nondeterministic automata with subsequent efficient semi-determinization.

Complementation. Complementation is another operation that is substantially harder for ω-automata than for automata over finite words. It took over half a century of research to find matching upper[10] and lower[11] bounds $\Theta((0.76n)^n)$ for complementing Büchi automata. Despite the high complexity, complementation of Büchi automata is a valuable tool for verification, language inclusion, or language subtraction. With the growing understanding of the worst-case complexity, the practical cost of complementing Büchi automata has become a second line of research as the worst case can often be avoided. Our motivation to tackle complementation of Büchi automata comes from the program termination analysis of ULTIMATE BÜCHI AUTOMIZER.[12] The aim of a program termination analysis is to decide whether a given program terminates on all inputs. In other words, it tries to establish or disprove that all infinite execution paths in the program flowgraph are infeasible. The ULTIMATE BÜCHI AUTOMIZER uses Büchi automata to represent infinite paths that are already known to be infeasible and it subtracts these paths (using complement and product) from the program flowgraph to identify the set of infinite execution paths whose infeasibility still needs to be proven.

Suitability of automata for model checking. The set of languages that can be recognized by automata over finite words are exactly the *regular languages* and the *ω-regular languages* for (most types of) ω-automata. While there is a unique minimal deterministic automaton for each regular language, the situation is more complicated for ω-automata – there is no equivalent to the minimization algorithm that we know for automata over finite words. Moreover, size is not the only relevant property of ω-automata that influences the process of model checking. Small size, the degree of determinism, and the simplicity of the acceptance condition can positively influence the performance of verification tools but they are often contradictory requirements from the perspective of LTL translators at the same time.[13] Furthermore, other aspects of particular ω-automata may influence model checking even more dramatically, for example, the location of accepting or initial states. With the variety of available tools for LTL to ω-automata translation, we have many ω-automata to consider to use for verification. Figure 1.3 shows six automata for the formula $\mathsf{GF}a \wedge \mathsf{GF}b$. *Which one is the most suitable for a given verification task?* We cannot answer this question entirely, but we offer at least some deeper insight for tasks solved by the model checker Spin.

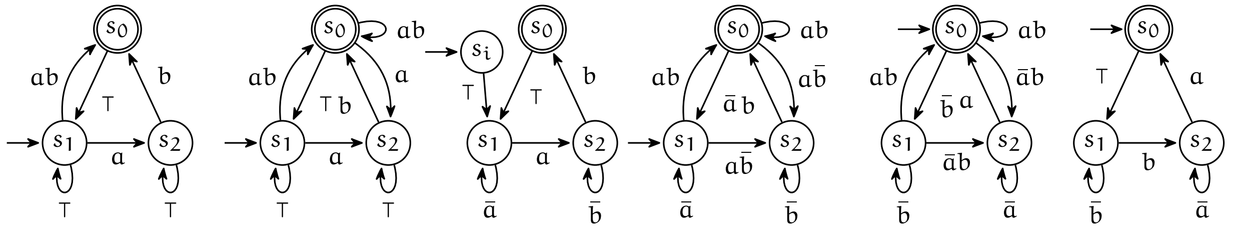
[9] An unambiguous automaton has at most one accepting run for each word. In a semi-deterministic automaton, each accepting run avoids nondeterministic states from some point on. Semi-deterministic automata are also known as *limit-deterministic* or *deterministic-in-the-limit*.

[10] Schewe (2009), "Büchi Complementation Made Tight", [6].

[11] Yan (2008), "Lower Bounds for Complementation of Omega-Automata Via the Full Automata Technique", [7].

[12] Heizmann, Hoenicke, and Podelski (2014), "Termination Analysis by Learning Terminating Programs", [8].

[13] For example, we can have a one-state deterministic Rabin automaton for the formula $\varphi = \mathsf{FG}a$ while no deterministic Büchi can express φ. Moreover, no Büchi automaton with less then two states exists for φ.



($\mathcal{C}_1$) Spin

($\mathcal{C}_2$) LTL2BA & LTL3BA

($\mathcal{C}_3$) MoDeLLa

($\mathcal{C}_4$) LTL3BA (det)

($\mathcal{C}_5$) Spot & Spot (det)

($\mathcal{C}_6$) Spot (no jump)

Figure 1.3: Automata for $\mathsf{GF}a \wedge \mathsf{GF}b$ generated by different tools and options.

1.1 OUTLINE AND CONTRIBUTION OF THE THESIS

Chapter 2 provides preliminaries and most definitions used throughout the thesis. In particular it introduces ω-automata and LTL. The rest of the thesis is divided into three parts; each part is devoted to ω-automata with varying degrees of determinism. The first part focuses on nondeterministic automata. It is followed by a part that deals with deterministic automata. Finally, the last part of the thesis discusses algorithms for semi-deterministic automata. The thesis contributes to the automata theory in the following areas.

Nondeterministic Büchi automata for explicit model checking. We study the connection of Büchi automata and concrete verification tasks performed by a successful explicit model checker called *Spin*. In particular we focus on two aspects.

- In Chapter 3 we search for properties of Büchi automata that really influence the performance of the central algorithm of Spin – *Nested Depth First Search*. We do so by manual analysis of several automata and by experiments with common LTL-to-BA translators and realistic verification tasks. As a result of these experiences, we gain a better insight into the characteristics of automata that work well with Spin.
- In Chapter 4 we provide methods that take a particular system to be verified, analyze the meaning of atomic propositions that are present in the system, and use this analysis to improve Büchi automata built from LTL specifications. As a result, we get smaller automata with shorter edge labels that are easier to understand. Thanks to these ω-automata we can improve the run time of Spin.

Translation of LTL into deterministic ω-automata.

- In Chapter 5 we define *May/Must alternating automata (MMAA)*, show (constructively) their expressive equivalence to LTL($\mathsf{F_s}, \mathsf{G_s}$), and provide a procedure that converts MMAA into deterministic transition-based generalized Rabin automata. These steps connect into an efficient translation of LTL($\mathsf{F_s}, \mathsf{G_s}$) into deterministic ω-automata. We have implemented this method in the tool LTL3DRA that is publicly available.

LTL($\mathsf{F_s}, \mathsf{G_s}$) is a fragment of LTL which uses the temporal operators *strict eventually* and *strict always* only.

- Chapter 6 offers an exhaustive experimental evaluation and comparison of various methods that transform formulae of LTL (and its fragments) into deterministic ω-automata.

Semi-deterministic Büchi automata construction and complementation.

- In Chpater 7 we first describe a transition-based adoption of the standard semi-determinization procedure for Büchi automata by Courcoubetis and Yannakakis[14] and we extend the algorithm with an SCC-aware[15] optimization. We also show how to tweak the construction to produce cut-deterministic automata (a stronger form of semi-determinism). We further present an algorithm for semi-determinization of generalized Büchi automata that is similar to the one presented by Hahn et al. in 2015.[16] All procedures were implemented in an open source tool called Seminator. We also evaluate and compare Seminator to other relevant tools.
- In Chapter 8 we present a specialized algorithm for complementation of semi-deterministic Büchi automata. For a semi-deterministic Büchi automaton with n states our algorithm creates an unambiguous Büchi automaton with at most 4^n states that recognizes complement of the language of the input automaton. Besides the theoretical result, this algorithm was successfully used to speed-up termination analysis in the ULTIMATE BÜCHI AUTOMIZER.

1.2 AUTHOR'S PUBLICATIONS AND HIS CONTRIBUTION

1.2.1 *Core of the Thesis*

Each of Chapters 3–8 is based on a conference publication co-authored by me. I list the publications and discuss my contribution, respecting the order of the chapters.

SPIN 2014 František Blahoudek, Alexandre Duret-Lutz, Mojmír Křetínský, and Jan Strejček.
"Is there a Best Büchi Automaton for Explicit Model Checking?" [11].
My contribution: *Participated in discussions, performed all experiments, participated in writing of the main body.* 30%

SPIN 2015 František Blahoudek, Alexandre Duret-Lutz, Vojtěch Rujbr, and Jan Strejček.
"On Refinement of Büchi Automata for Explicit Model Checking" [12].
My contribution: *Participated in discussions, on experiments, and on writing of the main body.* 25%

ATVA 2013 Tomáš Babiak, František Blahoudek, Mojmír Křetínský, and Jan Strejček.
"Effective Translation of LTL to Deterministic Rabin Automata: Beyond the (F, G)-Fragment" [13].
My contribution: *Participated in discussions, formulated the main algorithms and devised and written most of the proofs. Marginally collaborated on implementation and performed all experiments. Participated in writing of the main body.* 50%

LPAR 2013 František Blahoudek, Mojmír Křetínský, and Jan Strejček.
"Comparison of LTL to Deterministic Rabin Automata Translators" [14].
My contribution: *Participated in discussions, performed all experiments, participated in writing of the main body.* 55%

[14] Courcoubetis and Yannakakis (1988), "Verifying Temporal Properties of Finite-State Probabilistic Programs", [9].

[15] based on knowledge about *strongly connected components*

[16] Hahn et al. (2015), "Lazy Probabilistic Model Checking without Determinisation", [10].

LPAR 2017 František Blahoudek, Alexandre Duret-Lutz, Mikuláš Klokočka, Mojmír Křetínský, and Jan Strejček.
"Seminator: A Tool for Semi-Determinization of Omega-Automata" [15].
My contribution: *Participated in discussions and in formulation of algorithms, participated in writing the paper. Marginally participated in implementation and performed all experiments.* 30%

TACAS 2016 František Blahoudek, Matthias Heizmann, Sven Schewe, Jan Strejček, and Ming-Hsien Tsai.
"Complementing Semi-deterministic Büchi Automata" [16].
My contribution: *Participated in discussions and together with Sven Schewe formulated the algorithm. Substantially participated in writing the paper, performed the data analysis and prepare the final version of the experimental evaluation.* 25%

The thesis is based on these conference papers. However, some of the material was completely rewritten and some parts were substantially extended. In particular,

- the thesis uses a definition of ω-automata that rely on acceptance marks and Emerson-Lei acceptance condition in formal constructions,
- in comparison to **ATVA 2013** [13], the proofs in Chapter 5 have been reformulated using new terminology and concept of escaping multitransitions. The degeneralization of Rabin automata was completely rewritten.
- The comparison of tools from **LPAR 2013** [14] has been fully rewritten and revised. New tools have been included (determinization methods of Spot, Rabinizer 3, Rabinizer 4, LTL3TELA) and those that did not well in **LPAR 2013** [14] have been omitted.
- The presentation of material from **LPAR 2017** [15] has been completely rewritten, enhanced with formal descriptions of more algorithms, with illustrations and with proofs. Moreover, SCC-aware optimization has been described and implemented. New versions of Seminator and of other tools have been used in experimental evaluation.

Tools. The research done for this thesis has impact on several tools from the community. LTL3DRA[17] is an implementation of the translation of LTL to deterministic ω-automata presented in **ATVA 2014** [13]. Seminator[18] implements all algorithms described in Chapter 7 and it was presented in **LPAR 2017** [15]. The methods developed for **SPIN 2015** [12] were added to Spot.[19] The complementation algorithm described in **TACAS 2016** [16] is implemented in GOAL[20] and ULTIMATE BÜCHI AUTOMIZER.[21]

1.2.2 Other Publications and Projects

Hanoi Omega-Automata (HOA) Format. HOA format[22] is a flexible textual exchange format for ω-automata. It enables one to express deterministic, nondeterministic, or alternating automata in a uniform, human-readable, and succinct way. HOA format supports various structural variants such as labels

[17] https://github.com/xblahoud/ltl3dra

[18] https://github.com/mklokocka/seminator/

[19] https://spot.lrde.epita.fr/

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

[21] http://ultimate.informatik.uni-freiburg.de/

[22] Full specification of the format including some examples can be found at https://adl.github.io/hoaf/

on states or transitions, state-based or transition-based acceptance. Every ω-automaton is equipped with an Emerson-Lei acceptance condition (a Boolean formula over the acceptance primitives *infinitely often* and *finitely often*) which can express all acceptance conditions mentioned so far and more. The format was presented at the conference CAV 2015:

CAV 2015 Tomáš Babiak, František Blahoudek, Alexandre Duret-Lutz, Joachim Klein, Jan Křetínský, David Müller, David Parker, and Jan Strejček. "The Hanoi Omega-Automata Format" [17].

Translation of LTL into Transition-based Emerson-Lei Automata (TELA). We have created *LTL3TELA*,[23] which is a translator of LTL to (possibly nondeterministic) TELA. Similarly to LTL3BA and LTL3DRA, the translation uses alternating automata as an intermediate step. This experimental approach to LTL translation addresses the trade-off between complexity of acceptance condition and size of ω-automata – in comparison to Spot or LTL3BA it can produce smaller ω-automata with acceptance conditions that are usually harder to check.

[23] https://github.com/jurajmajor/ltl3tela

2 Preliminaries

This chapter introduces *ω-automata* and *Linear Temporal Logic (LTL)*.

Alphabets. An *alphabet* is a finite set of *letters*. We use two types of alphabets. In *classical* alphabets, letters are *symbols*, like in $\Sigma = \{a, b, c\}$. Letters in *propositional* alphabets are subsets of a finite set of atomic propositions; if $AP = \{a, b\}$ is a set of atomic propositions, $\Sigma = 2^{AP} = \big\{\emptyset, \{a\}, \{b\}, \{a, b\}\big\}$ is a propositional alphabet over AP. We usually use the symbol α to reference the letters of an alphabet.

Infinite words. An *infinite word* (or simply a *word*) over Σ is an infinite sequence of letters $u = u_0 u_1 u_2 \ldots \in \Sigma^\omega$. By $u_{i..}$ we denote the ith suffix $u_{i..} = u_i u_{i+1} \ldots$ of u.

2.1 ω-automata

ω-automata are finite automata over infinite words. The thesis does not cover automata over finite words and thus we also use the term automata to reference ω-automata. An ω-automaton is always equipped with some acceptance condition, typically Büchi, Rabin, Streett, or parity. Even though acceptance conditions of all automata used through the thesis could be classified as more or less standard, for clarity reasons, our definition follows the approach of the *Hanoi Omega-Automata (HOA)* format[1] and uses acceptance marks and acceptance formulae to describe the acceptance mechanism of automata. To clearly distinguish between the automata structure and its acceptance mechanism, we start with definition of a semiautomaton.

Semiautomata. A *semiautomaton* is a tuple $\mathcal{T} = (S, \Sigma, \delta, s_I)$, where S is a finite set of *states*, Σ is an alphabet, $\delta \subseteq S \times \Sigma \times S$ is a *transition relation*, and $s_I \in S$ is the *initial state*. A triple $t = (s, \alpha, s')\in \delta$ is a *transition* of s leading to s' under α and we also say that α is the label of t. A state s' is *reachable* from s in $\mathcal{T}$, denoted by $s \rightsquigarrow_{\mathcal{T}} s'$, iff there exists a sequence of transitions $(s_0, \alpha_0, s_1) \ldots (s_{k-1}, \alpha_{k-1}, s_k)$ such that $s_0 = s$ and $s_k = s'$. We use $s \leftrightsquigarrow_{\mathcal{T}} s'$ to denote the fact that s and s' are mutually reachable.

We write $s \rightsquigarrow s'$ and $s \leftrightsquigarrow s'$ instead of $s \rightsquigarrow_{\mathcal{T}} s'$ and $s \leftrightsquigarrow_{\mathcal{T}} s'$ when $\mathcal{T}$ is clear from context.

SCC. A *strongly connected component (SCC)* $C \subseteq S$ is a set of states that are all mutually reachable. An SCC C is *maximal* if no state outside C is mutually reachable with states from C. For each automaton there is a unique decomposition of the states into maximal strongly connected components.

Determinism. A state $s \in S$ is *deterministic* in δ if it has at most one transition under α in δ for each $\alpha \in \Sigma$. An SCC is deterministic if it consists of

[1] Babiak et al. (2015), "The Hanoi Omega-Automata Format", [17], see also https://adl.github.io/hoaf/.

deterministic states only and finally, a semiautomaton $\mathcal{T}$ and the transition relation δ are *deterministic* if all states from S are deterministic in δ.

Runs. A *run* of a semiautomaton $\mathcal{T}$ over a word $u = u_0u_1\ldots \in \Sigma^\omega$ is an infinite sequence $\sigma = (s_0, u_0, s_1)(s_1, u_1, s_2)\ldots \in \delta^\omega$ of transitions such that $s_0 = s_I$. A deterministic semiautomaton has at most one run for each word $u \in \Sigma^\omega$.

ω-automata. An *ω-automaton* is a tuple $\mathcal{A} = (S, \Sigma, \delta, s_I, M, \mu, \Phi)$ where (S, Σ, δ, s_I) is a semiautomaton, M is a finite set of *marks*, $\mu\colon M \to 2^{S\cup\delta}$ is a function that places marks on states and transitions, and finally Φ is an *acceptance formula*. We say that a transition or a state has a mark ● $\in M$ if it is a member of $\mu($●$)$. The acceptance formula is a positive Boolean combination of terms Inf● and Fin● where ● ranges over the set of marks M.

An ω-automaton is a semiautomaton with marks on states or transitions and with an acceptance formula. The marks with the acceptance formula say which runs of the semiautomaton are accepting.

The intuitive meaning of Inf is *to visit infinitely often* and the one of Fin is *to visit only finitely often*. For example, a generalized Büchi condition with two marks is expressed as Inf❶ ∧ Inf❷.

Semantics. The semiautomaton defines the runs of $\mathcal{A}$ and the acceptance marks and formula give semantics to these runs. Let σ be a run of $\mathcal{A}$. $\mathsf{Rec}(\sigma)$ is the set of states and transitions that appear infinitely often (recurrently) in the run. The *marks* of σ is the set of marks that are placed on states and transitions from $\mathsf{Rec}(\sigma)$, more precisely marks$(\sigma) = \{$● $\mid \mu($●$) \cap \mathsf{Rec}(\sigma) \neq \emptyset\}$. The run σ satisfies Inf● if ● $\in$ marks(σ) and it satisfies Fin■ if ■ $\notin$ marks(σ).[2] The run is *accepting* if it satisfies Φ. The language of $\mathcal{A}$ is the set $L(\mathcal{A})$ of all words $u \in \Sigma^\omega$ such that $\mathcal{A}$ has an accepting run over u.

Visualisation. We draw automata as in Figure 2.1. States are represented by nodes; the initial state has an incoming edge from an empty space, the acceptance formula is in the yellow box below the automaton itself, transitions are depicted as edges. If the automaton has a propositional alphabet, transitions between two states that have identical marks but different labels are merged into one *edge*. The edge is labelled by a boolean formula over atomic propositions in a condensed notation; the label is satisfied by exactly all labels of the merged transitions. For example, the label $\bar{a}b$ in the right automaton with $\Sigma = 2^{\{a,b\}}$ stands for $\neg a \wedge b$ and represents the unique transition under $\{b\}$, and any edge of the left automaton with label $\bar{b}$ represents transitions under $\{a\}$ and $\emptyset$. Sometimes a green box provides a corresponding LTL formula as in the case of the right automaton. Names of automata are typeset using a calligraphic alphabet and are enclosed in parenthesis in figures.

The condensed notation omits conjunctions and uses $\bar{a}$ for $\neg a$.

Tools that manipulate or generate automata usually also merge transitions into edges (both internally and for input/output). An edge is then a triple (s, l, s') where l is the edge-label.

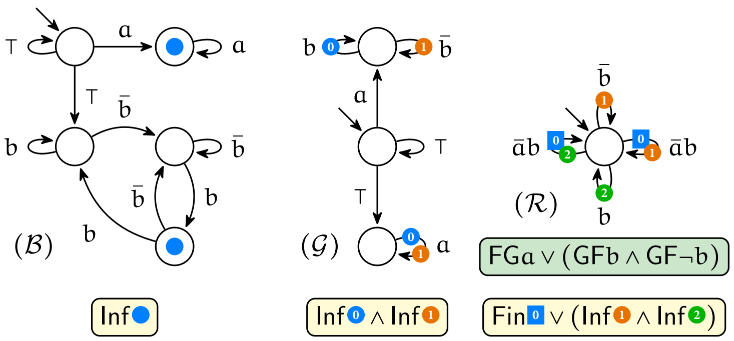


Figure 2.1: Three automata for the LTL formula FGa ∨ (GFb ∧ GF¬b). From left to right: Büchi with marks on states, generalized Büchi with marks on transitions, and deterministic generalized Rabin with marks on transitions.

[2] In this thesis we use a unique mark for each term of Φ and by convention we use circles for marks that appear in Inf-terms and squares for those in Fin-terms.

Standard acceptance conditions. We can express all standard acceptance conditions in our setting, you can see some examples above in Figure 2.1. We do not distinguish explicitly between state-based and transition-based acceptance[3] (we even allow to mix them). For Büchi and co-Büchi automata we need only one mark and the corresponding acceptance formulae are Inf● and Fin■, respectively, for generalized Büchi with k acceptance sets we need k marks and the formula is $\bigwedge_{i=0}^{k-1} \mathsf{Inf}$ⓘ. For a Rabin automaton with h Rabin pairs we need $2h$ marks and the formula is $\bigvee_{k=0}^{h-1} (\mathsf{Fin}\,\boxed{k} \wedge \mathsf{Inf}$ⓚ$)$. A Rabin pair is a conjunction of a co-Büchi and a Büchi condition, in a generalized Rabin pair the Büchi part is replaced by generalized Büchi and thus the acceptance formula for generalized Rabin automata is $\bigvee_{k\in K} \big(\mathsf{Fin}\,\boxed{k} \wedge \bigwedge_{j\in J_k} \mathsf{Inf}$ⓙ$\big)$.

Abbreviations. We often need to refer to automata that have certain properties. As their description can be rather long, we use abbreviations for automata types. A type of an automaton is influenced by the following three properties.

determinism: Deterministic [D], Nondeterministic [N], semi-deterministic [sD], cut-deterministic [cD]

the placement of marks: transitions [T], states [S]

acceptance condition: Büchi [B], generalized Büchi [GB], Rabin [R], generalized Rabin [GR]

In abbreviations, we use the same order as in the list and add an A which stands for automaton (or automata, regarding the context). We leave out these properties that are not of our interest. For example, the abbreviation *BA* denotes *Büchi automata* and *DTGRA* denotes *deterministic generalized Rabin automata with marks on transitions.*

Expressibility remark. The definition of an automaton used in this thesis allows for each label $\alpha \in \Sigma$ at most one transition between two states. In the HOA format you can also describe automata that have more such transitions that differ in the marks they carry. Such automata are not expressible by our definition. That is on purpose as it simplifies the presentation of most of the material and we also do not lose anything. Indeed, more transitions between two states are only useful for automata with some Fin-terms in the acceptance formula and marks on transitions. We use such automata only in Part II where all these automata are deterministic. Finally, no choice between transitions is permitted anyway in deterministic automata.

[3] State-based automata have marks only on states while transition-based automata have marks on transitions.

2.2 LINEAR TEMPORAL LOGIC (LTL)

The syntax of LTL is defined by

$$\varphi ::= \top \mid a \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mathsf{X}\varphi \mid \varphi \,\mathsf{U}\, \varphi,$$

We also use standard Boolean connectives (like $\Longrightarrow$ and $\Longleftrightarrow$) in their usual meaning as shorthands.

where $\top$ stands for *true*, a ranges over a countable set AP of *atomic propositions*, X and U are temporal operators called *next* and *until*, respectively. LTL formulae are interpreted over infinite words over the propositional alphabet $\Sigma = 2^{AP'}$, where AP' is a finite subset of AP.

We inductively define when a word u *satisfies* a formula φ, written $u \models \varphi$, as follows.

$$\begin{array}{ll}
u \models \top & \\
u \models a & \text{iff } a \in u_0 \\
u \models \neg\varphi & \text{iff } u \not\models \varphi \\
u \models \varphi_1 \vee \varphi_2 & \text{iff } u \models \varphi_1 \text{ or } u \models \varphi_2 \\
u \models \varphi_1 \wedge \varphi_2 & \text{iff } u \models \varphi_1 \text{ and } u \models \varphi_2 \\
u \models \mathsf{X}\varphi & \text{iff } u_{1..} \models \varphi \\
u \models \varphi_1 \,\mathsf{U}\, \varphi_2 & \text{iff } \exists i \geq 0 \,.\, (\, u_{i..} \models \varphi_2 \text{ and } \forall\, 0 \leq j < i \,.\, u_{j..} \models \varphi_1 \,)
\end{array}$$

Given an alphabet Σ, a formula φ defines the language $L^{\Sigma}(\varphi) = \{u \in \Sigma^{\omega} \mid u \models \varphi\}$. We write $L(\varphi)$ instead of $L^{2^{AP(\varphi)}}(\varphi)$, where $AP(\varphi)$ denotes the set of atomic propositions occurring in the formula φ.

We define the derived unary temporal operators *eventually* (F), *always* (G), *strict eventually* ($\mathsf{F_s}$), *strict always* ($\mathsf{G_s}$), and *releases* (R) by the following equivalences:

$$\begin{array}{ll}
\mathsf{F}\varphi \equiv \top \,\mathsf{U}\, \varphi & \mathsf{G}\varphi \equiv \neg\mathsf{F}\neg\varphi \\
\mathsf{F_s}\varphi \equiv \mathsf{XF}\varphi & \mathsf{G_s}\varphi \equiv \mathsf{XG}\varphi
\end{array}$$

$$\varphi_1 \,\mathsf{R}\, \varphi_2 \equiv \neg(\neg\varphi_1 \,\mathsf{U}\, \neg\varphi_2)$$

An LTL formula is in *positive normal form* if no operator occurs in the scope of any negation. Each LTL formula can be transformed to this form using De Morgan's laws for $\wedge$ and $\vee$ and the following equivalences:

$$\begin{array}{ll}
\neg\mathsf{F}\psi \equiv \mathsf{G}\neg\psi & \neg\mathsf{G}\psi \equiv \mathsf{F}\neg\psi \\
\neg\mathsf{F_s}\psi \equiv \mathsf{G_s}\neg\psi & \neg\mathsf{G_s}\psi \equiv \mathsf{F_s}\neg\psi \\
\neg(\varphi_1 \,\mathsf{R}\, \varphi_2) \equiv \neg\varphi_1 \,\mathsf{U}\, \neg\varphi_2 & \neg(\varphi_1 \,\mathsf{U}\, \varphi_2) \equiv \neg\varphi_1 \,\mathsf{R}\, \neg\varphi_2
\end{array}$$

$$\neg\mathsf{X}\varphi \equiv \mathsf{X}\neg\varphi$$

We say that a formula is *temporal* if its topmost operator is neither conjunction nor disjunction; note that a and $\neg a$ are also temporal formulae.

Part I

HOW BÜCHI AUTOMATA INFLUENCE EXPLICIT MODEL CHECKING

3 *Is There a Best Büchi Automaton for Spin?*

Model Checking

In the traditional view, the *model checking*[1] problem decides whether a given system is a model of a given formula, that is whether all behaviours of the system satisfy the formula. We see the model checking as a tool that decides whether or not the system has an erroneous behaviour; we start with a formula φ that describes the erroneous behaviour[2]and we consider the system correct if no behaviour of the system satisfies φ. Model checking of LTL expects that φ is an LTL formula.

The automata-theoretic approach[3] to model checking relies on automata to internally represent both the specification and the system; it usually proceeds in the following four steps as illustrated by Figure 3.1.

1. Build the state space $\mathcal{S}$; the state space represents all possible executions of the system to be verified,

2. translate the LTL formula φ into a Büchi automaton[4] $\mathcal{A}_\varphi$ that accepts all faulty behaviours,

3. build the synchronous product $\mathcal{S} \otimes \mathcal{A}_\varphi$ of the system and the automaton; the product represents all behaviours of $\mathcal{S}$ that conform to $\mathcal{A}$ and φ and thus are erroneous, and finally

4. check this product for emptiness.

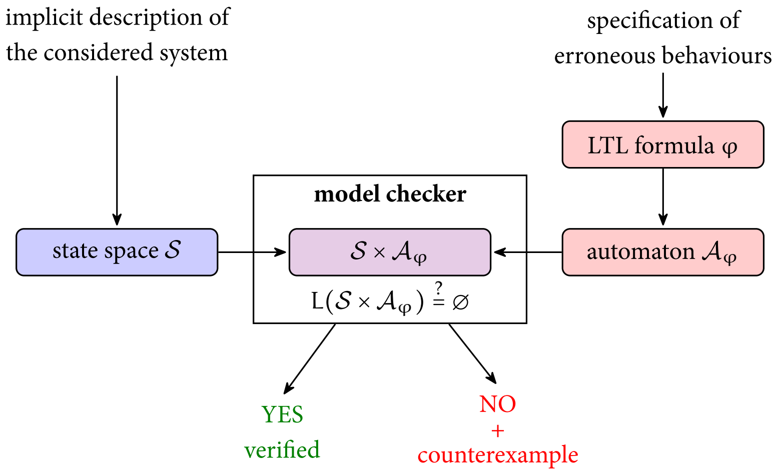

implicit description of
the considered system
specification of
erroneous behaviours
LTL formula φ
model checker
state space $\mathcal{S}$
$\mathcal{S} \times \mathcal{A}_\varphi$
automaton $\mathcal{A}_\varphi$
$L(\mathcal{S} \times \mathcal{A}_\varphi) \overset{?}{=} \varnothing$
YES
verified
NO
+
counterexample

Figure 3.1: Automata-theoretic approach to model checking.

Although we anticipate here a specification as an LTL formula, we may generalize many results of this part to applications where the erroneous behaviours are given directly as Büchi automata or in another formalism that can be converted into automata.

[1] Baier and Katoen (2008), “Principles of Model Checking”, [18].

[2] We can simply negate the input formula to switch between the two views.

[3] Vardi (1995), “An Automata-Theoretic Approach to Linear Temporal Logic”, [19].

[4] also called *property automaton*

The automata approach effectively reduces the problem of model checking to the problem of language emptiness for Büchi automata. If $\mathsf{L}(\mathcal{S} \otimes \mathcal{A}_\varphi)$ is empty then we can consider $\mathcal{S}$ to be safe with respect to φ. On the other hand, if the product $\mathcal{S} \otimes \mathcal{A}_\varphi$ accepts a word w then we have a concrete example of the erroneous behaviour of $\mathcal{S}$.

Indeed, the main work of a model checker consists of building the product and check its language for emptiness.

In the traditional view of model checking, w is known as a *counterexample*.

Spin[5] is a successful *explicit* model checker that relies on the automata approach. The word explicit emphasises the fact that it explicitly enumeratesall the states of $\mathcal{S}$ and of the product $\mathcal{S} \otimes \mathcal{A}_\varphi$ and stores them in the memory. The explicit approach often suffers from the so-called *state space explosion* problem — the product is simply too large to be stored in memory or takes too long to analyze. Many model checkers (including Spin) perform the steps 3 and 4 simultaneously — they build the product *on-the-fly* according to the needs of the emptiness check. In this way, the model checkers build and store only the relevant part of the product. To fight the state space explosion problem, developers of model checkers implemented many other methods how to handle the given product more effectively.[6]

When you want to make the product smaller, you have to focus on the property automaton $\mathcal{A}_\varphi$; the system is given. This is where the LTL-to-BA translators came into the play. There are many algorithms and tools for translating LTL formulae into Büchi automata and they produce various language equivalent automata. For instance, Figure 3.5 on the page 36 shows several Büchi automata for the LTL formula $\mathsf{GF}a \land \mathsf{GF}b$.[7] This chapter address the following question. *Should one be preferred over the others?*

To pick the best automaton for a given formula is more than difficult — it is even impossible if we do know how $\mathcal{S}$ looks like. The intuition that a smaller $\mathcal{A}_\varphi$ produces a smaller synchronous product $\mathcal{S} \otimes \mathcal{A}_\varphi$ is not always correct.[8] We discuss various approaches to product reductions considered previously by authors of LTL-to-BA translators or of automata reductions in Section 3.2.

The property automaton influences not only the number of states or transitions in the product. The automaton can heavily influence also the emptiness check (step 4). Before we discuss how the emptiness check depends on the property automaton, we have to understand how the emptiness check of Spin works. From the variety of possible emptiness check algorithms, Spin chooses *Nested Depth-First Search (NDFS)*.[9]

[5] Holzmann (1997), "The Model Checker SPIN", [20]; Holzmann (2003), "The SPIN Model Checker: Primer and Reference Manual", [21].

[6] See Pelánek (2008), "Fighting State Space Explosion: Review and Evaluation", [22], for a nice review.

[7] This and the following chapter deal mainly with Büchi automata with marks on states. Therefore, we use the classic convention for their visualization: the accepting states are marked with a double circle and we omit the acceptance formula.

[8] See Figure 3.3 on page 34 for an example.

[9] Holzmann, Peled, and Yannakakis (1996), "On Nested Depth First Search", [23].

Nested Depth-First Search (NDFS)

To check the language emptiness of the product $\mathcal{S} \otimes \mathcal{A}_\varphi$, Spin has to search for a cycle that is reachable from the initial state and that contains at least one accepting state. By default, Spin uses an algorithm that is based on two nested depth-first searches: *blue* and *red*. The blue DFS plays the leading role. It explores the product and every time it would backtrack from an accepting state $\mathtt{s}$[10] it starts a red DFS from $\mathtt{s}$. If the red DFS reaches any state on the blue DFS search stack then a reachable and accepting cycle is found[11] and the algorithm reports it as a counterexample. Otherwise, the red DFS terminates and the blue DFS can continue. The two DFS always ignore states that have been completely explored by an instance of the red DFS, so a state is never visited more than twice.

Spin utilizes an extra optimization, if the blue DFS hits its own search stack by following a transition that is either going to or coming from an accepting state, Spin reports an accepting cycle without even starting any red DFS.[12]

Now we are ready to see that the number of states or transitions in not always relevant: ultimately, only the part of the product that is explored by the emptiness check does matter. Some authors of automata optimizations or LTL-to-BA translation improvements provide also run times of a selected emptiness check executed on the product of obtained automata and either random state spaces or few realistic systems.[13] Etessami and Holzmann even complained that the relation between the size of $\mathcal{A}_\varphi$ and the run time of the model checking procedure was difficult to predict, especially in the presence of a counterexample.

When a counterexample exists in the product, the emptiness check may report it more or less rapidly depending on the order in which the NDFS explores the transitions of the product. With any luck, the first transition selected at each step of the DFS will lead to an accepting cycle. Conversely, the first transitions followed might lead to a huge component of the product that just turns out to be a dead-end, and from which the emptiness check has to backtrack before finding the counterexample.

The selected transition order in $\mathcal{S} \otimes \mathcal{A}_\varphi$ depends on the order of the transitions in the property automaton $\mathcal{A}_\varphi$. Previous attempts to explore reordering of the transitions of $\mathcal{A}$ to help the emptiness check have been inconclusive.[14] Furthermore, the swarming techniques[15] used nowadays makes this topic even less attractive: in these approaches, several threads compete to find a counterexample in $\mathcal{S} \otimes \mathcal{A}_\varphi$ using a different, random transition order for $\mathcal{A}_\varphi$. Therefore, we do not address the question of the transition order.

Like the previous two paragraphs and Figure 3.3 document, methods that aim mainly to decrease the size and determinism of the automata cannot be universal and we cannot hope for the best automaton for all verification tasks with the same specification. Therefore we focus on other aspects that are helpful for *Nested Depth First Search (NDFS)* – the emptiness check of Spin. To gain a better insight into the characteristics of automata that work well with Spin, we look at concrete examples of how formulae are translated into automata differently by existing tools and how these automata influence NDFS.

[10] We backtrack from $\mathtt{s}$ after all successors of $\mathtt{s}$ have been explored by the blue DFS.

[11] Since $\mathtt{s}$ is reachable from all states on the blue DFS search stack.

[12] Gastin, Moro, and Zeitoun (2004), "Minimization of Counterexamples in SPIN", [24]; Schwoon and Esparza (2005), "A Note on On-the-Fly Verification Algorithms", [25].

[13] Etessami and Holzmann (2000), "Optimizing Büchi Automata", [26]; Dax, Eisinger, and Klaedtke (2007), "Mechanizing the Powerset Construction for Restricted Classes of ω-Automata", [27], for example.

[14] Geldenhuys and Valmari (2005), "More Efficient On-the-Fly LTL Verification with Tarjan's Algorithm", [28].

[15] Holzmann, Joshi, and Groce (2011), "Swarm Verification Techniques", [29].

3.1 MOTIVATION BY EMPIRICAL DATA: HOW MUCH CAN AUTOMATA INFLUENCE SPIN

First of all, we present experimental results showing how important the impact of Büchi automata on Spin's performance can be. We use the following benchmark, software, and hardware.

Benchmark. We base our benchmark on the set of 769 realistic model checking tasks BEEM.[16] A verification task consists of a system in Promela[17] and an LTL formula that describes a desired property of the system.[18] We have enriched the benchmark set by a few tasks. To each system describing some mutual exclusion algorithm,[19] we added three specification formulae:

1. $\mathsf{GF}(\mathsf{P}_0@\mathrm{CS}) \implies \mathsf{GF}(\mathsf{P}_0@\mathrm{NCS})$ meaning that if a process P_0 spends infinitely many steps in a critical section, then it also spends infinitely many steps in a non-critical section,

2. $\mathsf{GF}(\mathsf{P}_0@\mathrm{NCS}) \implies \mathsf{GF}(\mathsf{P}_0@\mathrm{CS})$ meaning that if a process P_0 spends infinitely many steps in a non-critical section, then it also spends infinitely many steps in a critical section,

3. $\mathsf{FG}\neg\big((\mathsf{P}_0@\mathrm{CS} \wedge \mathsf{P}_1@\mathrm{CS}) \vee (\mathsf{P}_0@\mathrm{CS} \wedge \mathsf{P}_2@\mathrm{CS}) \vee (\mathsf{P}_1@\mathrm{CS} \wedge \mathsf{P}_2@\mathrm{CS})\big)$ meaning that after finitely many steps, it never happens that two of the processes P_0, P_1, and P_2 are in a critical section at the same time.

To sum up, we consider $769 + 3 \cdot 23 = 838$ verification tasks. All the benchmarks and measurements presented in this section are available at http://fi.muni.cz/~xstrejc/publications/spin2014.tar.gz.

Software. We use the five LTL-to-BA translators presented in Table 3.1: Spin and LTL2BA are well established and popular translators, MoDeLLa was the first translator focusing on determinism of the produced automata, and LTL3BA and Spot represent contemporary translators. The last two translators are used in several settings: the settings denoted by *LTL3BA (det)* and *Spot (det)* aim to produce more deterministic automata, while the setting called *Spot (no jump)* is explained in Section 3.3.

tool	version	command
Spin [21]	6.2.5	`spin -f`
LTL2BA [31]	1.1	`ltl2ba -f`
MoDeLLa [32]	1.5.9	`mod2spin -f`
LTL3BA [33]	1.0.2	`ltl3ba -S -f`
LTL3BA (det)		`ltl3ba -S -M -f`
Spot [34]	1.2.4	`ltl2tgba -s`
Spot (det)		`ltl2tgba -s -D`
Spot (no jump)		`ltl2tgba -s -x degen-lskip=0`

Table 3.1: Considered LTL-to-BA translators, for reference.

Spin version 6.2.5 is also used as the model checker in our evaluation. We limited its maximal search depth to 100 000 000 and we kept the default settings otherwise. In particular, the partial-order reduction, which severely limits the exploration of the state-space, is enabled.[20] To obtain some of the statistics, we used the *ltlcross* tool from the Spot library.

[16] Pelánek (2007), "BEEM: Benchmarks for Explicit Model Checkers", [30].

[17] **PROcess MEta LAnguage** is a modelling language used by SPIN for both systems and property automata.

[18] We negate the formula so that it describes erroneous behaviours.

[19] altogether 23 instances of parametric models called `anderson`, `peterson`, and `bakery`

[20] See the script `stat.pl` in the archive for the exact parameters we used with Spin.

Hardware and settings. All computations were performed on a machine with eight physical processors and 448 GiB RAM.[21] Each execution of Spin has been restricted by 30 minutes timeout and a memory limit of 20GiB. The memory limit was always reached first.

Workflow. For each of the 838 considered verification tasks, we negate the specification formula,[22] we translate the negated formula by all the mentioned translators and we run Spin on the system with each of the obtained automata. Translation of the negated formula to an automaton is instantaneous[23] in nearly all cases: there is only one formula for which the translator built in Spin needs a couple of seconds to finish.

Originally, we have measured the impact of Büchi automata on Spin by its run time. Unfortunately, our computation server is shared with other users and its variable workload has led to enormous dispersion of measured run times. We have observed a run time difference of over 300% on the same input. Hence, instead of on run times, we focus on the count of *visited transitions*, which is a stable statistic produced directly by Spin. The number of visited transitions accumulates the numbers of product transitions explored in depth-first searches executed during a run of the NDFS algorithm. Hence, the number of visited transitions should be proportional to the run time on a dedicated machine. Spin also provides statistics for *stored states*, which is the total count of constructed and stored product states and should be proportional to the memory consumed by Spin.

Evaluation. Spin successfully finishes the computation within the given limits for at least two automata obtained by different translation tools for exactly 823 tasks. For each such verification task, we find the maximal and the minimal numbers of visited transitions and we compute their ratio. Intuitively, the ratio represents how many times slower Spin can be if we choose the worst of the produced automata compared to the best of those. Out of the 823 tasks, the ratio is exactly 1 in only 35 cases. In other words, in more than 95% of the considered verification tasks, the choice of an LTL-to-BA translator has an influence on the run time of Spin.

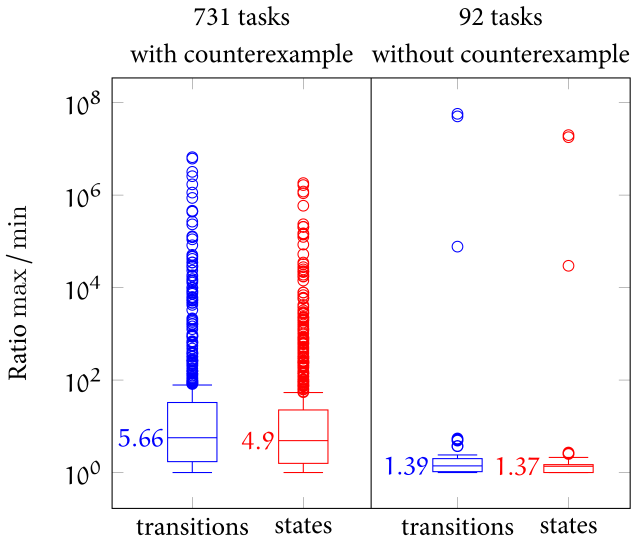


Figure 3.2: Impact of the Büchi automata on model checking. For each verification task, we compute ratios between the maximum and minimum number of transitions (or unique states) visited by Spin using all available Büchi automata. In each column, a box spans between the first and third quartiles of the ratio, and is split by the median (whose value is given). The whiskers show the range of ratios below the first and above the third quartile that are not further away from the quartiles than 1.5 times the interquartile range. Other values are shown as outliers using circles.

[21] In more detail: the machine is an HP DL980 G7 server with 8 eight-core 64-bit 2.26GHz processors Intel Xeon X7560 and with 448 GiB DDR3 RAM. We ran at most 8 instances of Spin in parallel.

[22] to have a formula for erroneous behaviour

[23] It takes less than 0.1 s.

As expected, the ratios significantly differ for verification tasks where the model satisfies a given formula and for those with a counterexample. Out of the 823 tasks, 731 tasks contain counterexamples while 92 tasks do not. The ratios for these two sets are presented by box-plots in Figure 3.2. One can clearly see that the selection of a Büchi automaton has a bigger impact on the verification tasks with counterexamples (median ratio is over 5.6) than on the tasks without counterexamples (median ratio is 1.4). Both sets contain extreme cases where the ratios exceed 10^6.

If we compute the ratios of maximal and minimal numbers of stored states, we get the ratio 1 in only 68 out of the 823 tasks. The situation is analogous to the ratios of visited transitions, but the ratios of stored states are slightly lower.

To sum up, the choice of the Büchi automaton can have a dramatic impact on speed and memory consumption of Spin.

3.2 STANDARD APPROACH TO OPTIMIZATION: HELPING THE PRODUCT

Most of the work on optimizing the translation of LTL formulae to Büchi automata has focused on building Büchi automata with the smallest possible number of states.[24] This is motivated by the observation that the synchronous product of a Büchi automaton $\mathcal{A}$ with a state space $\mathcal{S}$ can have the same number of states as their Cartesian product in the worst case: $|\mathcal{S} \otimes \mathcal{A}| \le |\mathcal{S}| \times |\mathcal{A}|$. Therefore, decreasing $|\mathcal{A}|$ lowers the upper bound on $|\mathcal{S} \otimes \mathcal{A}|$.

However, it is possible to build contrived examples where a smaller $|\mathcal{A}|$ yields a larger product. For instance, removing one state in the automaton $\mathcal{A}_1$ of Figure 3.3 doubles the size of its product with the state space $\mathcal{S}$ of the same figure from 3 to 6 states. Of course, if $\mathcal{S}$ was a similar cycle of 2 states, the smaller automaton $\mathcal{A}_2$ would give a smaller product. Hence, one cannot hope to build an optimal property automaton $\mathcal{A}$ without a priori knowledge of the system $\mathcal{S}$.

With the introduction of LBTT,[25] a tool that checks the output of different LTL-to-BA translators by doing many cross-comparisons, including some products with random state spaces, tool designers started to evaluate not only the size of the produced automata but also the size of their products with random state spaces.[26] A recent clone of LBTT called `ltlcross`[27] computes multiple products with random state spaces to lessen the *luck factor*. Sebastiani and Tonetta used this "product with a random state space" measurement to benchmark their translator MoDeLLa against other available translators to support the claim that producing "more deterministic" Büchi automata might be more important than producing small Büchi automata.[28]

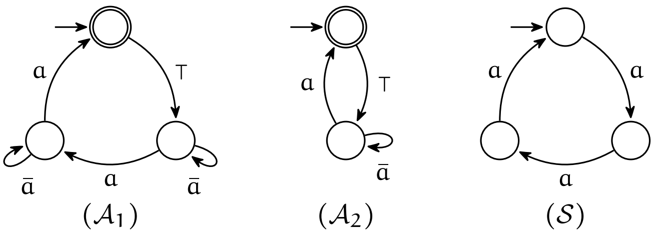


Figure 3.3: Two BA for $\mathsf{GF}a$ and a state space. $\mathcal{S} \otimes \mathcal{A}_1$ has 3 states whereas $\mathcal{S} \otimes \mathcal{A}_2$ has 6.

[24] e.g. Gastin and Oddoux (2001), [31]; Couvreur (1999), [35]; Somenzi and Bloem (2000), [36]; Giannakopoulou and Lerda (2002), [37]; Thirioux (2002), [38].

[25] Tauriainen and Heljanko (2002), "Testing LTL Formula Translation into Büchi Automata", [39].

[26] e.g. Sebastiani and Tonetta (2003), [32]; Duret-Lutz and Poitrenaud (2004), [40].

[27] Duret-Lutz (2013), "Manipulating LTL Formulas Using Spot 1.0", [41].

[28] Sebastiani and Tonetta (2003), "*More Deterministic* vs. *Smaller* Büchi Automata for Efficient LTL Model Checking", [32].

		automata				products		cases with *product trans* bigger than...							
	n	states	ndst	edges	trans	states	trans	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
(1) Spin	161	1739	1474	9318	46252	260934	8892105	0	102	143	107	150	150	150	146
(2) LTL2BA	178	1003	802	3360	30159	191668	5556159	5	0	137	49	161	157	156	142
(3) MoDeLLa	178	1297	647	4311	23874	216938	4193567	15	33	0	41	110	116	114	91
(4) LTL3BA	178	795	595	2209	21240	151373	4273646	0	23	126	0	149	153	152	140
(5) LTL3BA (det)	178	830	326	2405	14414	155716	2901474	0	0	10	5	0	76	75	63
(6) Spot	178	657	94	1615	10304	127792	2326271	1	6	15	5	30	0	1	1
(7) Spot (det)	178	662	88	1639	10414	128178	2328422	1	7	17	6	33	4	0	0
(8) Spot (no jump)	178	785	104	1874	12273	152592	2719360	12	28	40	27	70	61	57	0

Table 3.2: Benchmark based on automata and product sizes. Column *n* indicates how many translations are successful within the allocated time. The *automata* columns show accumulated values of standard automata characteristics for all successful translations. Column *ndst* gives the number of non-deterministic states in the automata. All produced automata are synchronized with the same 100 random systems, and the median number of states and transitions of these products is kept. The *products* columns represent the medians accumulated over all successful translations. The right-most part of the table counts the number of formulae for which the translator on the row produces an automaton with higher median number of transitions in the products that the translator of the column.

You can find a typical example of a benchmark based on product sizes in Table 3.2. The table shows numbers for 178 formulae from the literature[29] translated by 8 different LTL-to-BA translators. The timeout for one translation was set to 60 seconds.

The table shows that MoDeLLa generates automata that are slightly bigger than LTL2BA (its competitor in 2003), but when looking at the product, MoDeLLa causes fewer transitions to be built. If the number of transitions is proportional to the run time of a model checker and the number of states is proportional to its memory consumption, MoDeLLa has effectively traded memory for speed.

MoDeLLa's results do not appear to hold today: more recent translators such as LTL3BA or the translator of Spot can produce automata that are significantly smaller and yield smaller products with random state spaces. These translators also have options to produce more deterministic automata, but the resulting products are not always better.

The right part of Table 3.2 compares the translators by the sizes of products of produced automata with a fixed set of random systems. For instance, one can observe that even though Spot (6) produces the lowest accumulated number of product transitions in this benchmark, there are 30 formulae where the generated products have more transitions than those obtained by LTL3BA (det) (5). Conversely, automata from LTL3BA (det) produce products with more transitions than those of Spot for 76 formulae.

As Spin constructs the product on-the-fly, if we optimize $\mathcal{A}$ to minimize $|\mathcal{S} \otimes \mathcal{A}|$, we may not necessarily optimize $\mathcal{A}$ for the model checking procedure. The emptiness check may explore only a part of the product, or, conversely, it may explore the whole product twice. Ultimately, any change to $\mathcal{A}$ should be measured particularly by its effect on the model checker. For instance, Dax et al. performed such an evaluation.[30] In addition to explaining how to build minimal weak deterministic Büchi automata (WDBA) for a subclass of LTL, they showed that their minimal WDBA are smaller than the non-deterministic

[29] Etessami and Holzmann (2000), [26]; Somenzi and Bloem (2000), [36]; Dwyer, Avrunin, and Corbett (1998), [42].

[30] Dax, Eisinger, and Klaedtke (2007), "Mechanizing the Powerset Construction for Restricted Classes of ω-Automata", [27].

BA produced by other translators and they also show that they improved the run times of Spin on a few verification tasks.[31]

3.3 ANOTHER VIEW TO OPTIMIZATION: HELPING THE EMPTINESS CHECK

3.3.1 *Weak Automata*

Remember that the blue DFS can detect an accepting cycle without running a red DFS? It happens when the blue DFS hits its own stack on (or from) an accepting state. With this optimization in mind, we suggest that of the two automata of Figure 3.4, $\mathcal{B}_2$ should be preferred. Indeed, when the blue DFS reaches a state of its search stack in the product $\mathcal{S} \otimes \mathcal{B}_2$, it is guaranteed to come from (and go to) an accepting state, detecting the accepting cycle without starting any red DFS. In the product $\mathcal{S} \otimes \mathcal{B}_1$ we might be less lucky if we close the cycle with the transition at the bottom of $\mathcal{B}_1$: in that case the product has to be explored a second time by the red DFS.

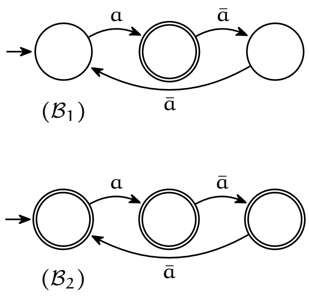


Figure 3.4: Two automata for the LTL formula $a \land \mathsf{G}(a \implies \mathsf{X}(\bar{a} \land \mathsf{X}(\bar{a} \land \mathsf{X}a)))$. $\mathcal{B}_1$ is inherently weak and $\mathcal{B}_2$ is weak.

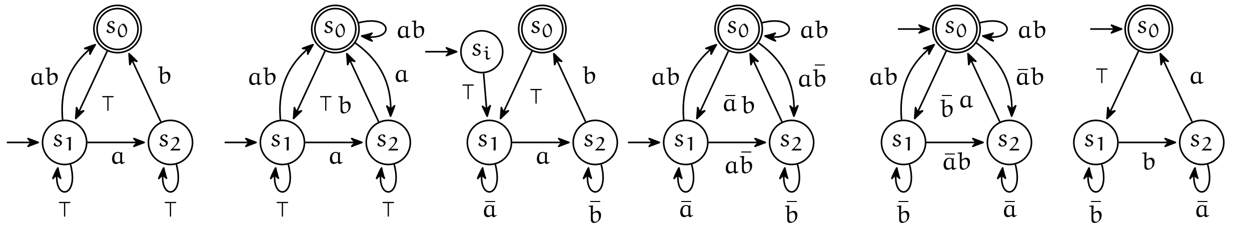
We actually illustrate the distinction between weak automata and inherently weak automata by this example. An *inherently weak automaton* is an automaton in which strongly connected components (SCCs) cannot mix accepting cycles with non-accepting cycles. A *weak automaton* is an inherently weak automaton in which the states of each SCC are either all accepting or all non-accepting. Any inherently weak automaton can be easily transformed into an equivalent weak automaton.[32]

Having more accepting states is not necessarily good from the point of view of the NDFS since a red DFS is started every time the blue DFS backtracks from an accepting state. However, if an entire SCC is non-accepting, the first red DFS will cover it fully, and each successive red DFS will immediately return because it attempts to process a state that has already been seen by a previous red DFS.

3.3.2 *Automata for* $\mathsf{GF}a \land \mathsf{GF}b$

Figure 3.5 shows six different Büchi automata for the formula $\mathsf{GF}a \land \mathsf{GF}b$ produced by the considered tools. Note that if you ignore the exchange of a and b,[33] automata $\mathcal{C}_4$ and $\mathcal{C}_5$ differ only in the initial state and thus cannot be distinguished by any determinism-based or size-based metrics.

Table 3.3 captures data about Spin's runs on the bakery mutual exclusion protocol taken from BEEM and the property automata of Figure 3.5. The propositions a and b describe situations that (different) pairs of processes are



($\mathcal{C}_1$) Spin

($\mathcal{C}_2$) LTL2BA & LTL3BA

($\mathcal{C}_3$) MoDeLLa

($\mathcal{C}_4$) LTL3BA (det)

($\mathcal{C}_5$) Spot & Spot (det)

($\mathcal{C}_6$) Spot (no jump)

Figure 3.5: Automata for $\mathsf{GF}a \land \mathsf{GF}b$ generated by different tools and options.

[31] We had omitted their tool from our benchmark because (1) it only supports a subset of LTL, and (2) their optimization is implemented in Spot and both tools would, therefore, return the same automata. Besides, the subset of LTL does not include the formulae studied in Sections 3.3.2 and 3.3.4.

[32] We can safely mark all states in accepting strongly connected components as accepting, see: Boigelot, Jodogne, and Wolper (2001), "On the Use of Weak Automata for Deciding Linear Arithmetic with Integer and Real Variables", [43].

[33] The atomic propositions a and b have a symmetric purpose in the original formula.

	automaton size				statistics from Spin's execution		
	states	ndst	edges	trans	stored states	visited trans	time
$\mathcal{C}_1$ Spin	3	2	6	17	27531713	95071k	88s
$\mathcal{C}_2$ LTL2BA & LTL3BA	3	3	8	20	27531713	95071k	99s
$\mathcal{C}_3$ MoDeLLa	4	0	6	16	27531714	95071k	109s
$\mathcal{C}_4$ LTL3BA (det)	3	0	8	12	27531713	95071k	101s
$\mathcal{C}_5$ Spot & Spot (det)	3	0	8	12	27531714	190143k	211s
$\mathcal{C}_6$ Spot (no jump)	3	0	5	12	27531714	190143k	191s

Table 3.3: Statistics about generated automata and Spin's run on system `bakery.7.pm` and formula $\mathsf{GF}a \wedge \mathsf{GF}b$ where neither a nor b ever occurs in the system. The corresponding automata are shown in Fig. 3.5.

in the critical section at the same time. The protocol prevents such situation, so neither a nor b is ever true in the model. We observe that in case of products with automata $\mathcal{C}_5$ and $\mathcal{C}_6$ (both produced by Spot), Spin explores the products twice because Spin triggers the red DFS from the initial state of the product. This is not the case for the other automata. This yields the following hypothesis: *When we suppose that there is no accepting cycle in the product, the automaton should keep its accepting states as hard to reach from the initial state as possible.* The further the accepting states are from the initial state, the more chance we have that the blue DFS will never reach any accepting state and therefore no red DFS will be triggered.

For instance, if we ignore the renaming of atomic propositions, the automaton $\mathcal{C}_3$ could be obtained from $\mathcal{C}_6$ by *unrolling* the accepting cycle by one step, so that the cycle is entered on a non-accepting state, and the accepting state is actually the last one visited on the cycle.[34] This superfluous initial state only makes a negligible difference on the product, and does not incur any noticeable difference for Spin compared to $\mathcal{C}_1$, $\mathcal{C}_2$, or $\mathcal{C}_4$.

Similarly, if we do not expect an accepting cycle in the product, the inherently weak automaton $\mathcal{B}_1$ of Figure 3.4 could be changed by letting the right-most state be accepting instead of the middle one.

3.3.3 *Translation Differences*

Most LTL-to-BA translators follow a multi-step procedure where they first translate a given LTL formula into a generalized Büchi automaton, often with marks on transitions, such as those of Figure 3.6. Translators then *degeneralize* these automata to obtain a BA. Other simplification procedures may be applied to these automata, but it turns out that the last three automata of Figure 3.5 were all obtained by degeneralizing $\mathcal{G}_1$ in Figure 3.6, and their differences are due to choices made in the degeneralization procedure.

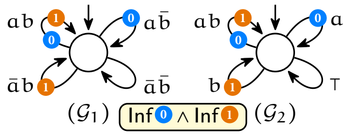


Figure 3.6: Two TGBA for $\mathsf{GF}a \wedge \mathsf{GF}b$.

When degeneralizing a TGBA $\mathcal{G}$ with acceptance marks $●^0, \ldots, ●^h$ (the ⓪ and ① on the Figure 3.6), the structure of $\mathcal{G}$ is cloned $h+2$ times. Let us call each of these clones a level. For each state of level $l \leq h$, all transitions originally marked with $●^l$ have their destination redirected to the next level, the destination of all transitions in level $h+1$ are redirected to level 0. Finally, all the states of the level $h+1$ are made accepting. The initial state can be put on any level.

This procedure ensures that words accepted by the degeneralized automaton correspond to words recognized by runs of $\mathcal{G}$ that visit all acceptance marks

[34] This is not actually the reason why MoDeLLa produces $\mathcal{C}_3$. Internally, MoDeLLa translates the formula into a Büchi automaton with labels on states and has to deal with possibly multiple initial states. When it outputs an automaton, it *always* adds an extra initial state with copies of the outgoing transitions of all the original initial states, even if the original automaton had only one initial state. See also $\mathcal{D}_3$ of Figure 3.7 where s_0 and s_2 were the original initial states.

infinitely often. Accepting cycles in products involving these degeneralized automata will always involve at least $h + 2$ states.

The degeneralization applied to $\mathcal{G}_1$ with the initial state on the last level and the acceptance marks ordered as ❶, then ⓿, produces the automaton $\mathcal{C}_6$ of Figure 3.5. Changing the degeneralization order to ⓿, then ❶, and putting the initial states on the first level would give automaton $\mathcal{C}_4$.

An optimization introduced with LTL2BA[35] consists in jumping levels. If a transition of a level $l \leq h$ is marked by all marks ●$^{l} \ldots$●^{j}, its destination can be redirected directly to the level $j + 1$. Similarly, if a transition from the level $h + 1$ is marked by ●$^{0} \ldots$●^{j}, it can be redirected to the level $j + 1$. Implementing this optimization gives automaton $\mathcal{C}_5$.

Often (but not in this example), jumping levels is a way to avoid creating useless copies of some states. Another side effect of this optimization is that some accepting cycles may be shorter than $h + 2$: the change effectively keeps the automaton as close to the accepting level as possible. If we are looking for counterexamples, $\mathcal{C}_5$ appear better than $\mathcal{C}_6$ because its accepting cycles are shorter on average.

We recall that the initial state of a degeneralized automaton can be put on any level. For example, Giannakopoulou and Lerda noticed that by changing the initial level, they could sometimes save some states, so they try to use both the first and the last level and keep the smallest automaton.[36] In our example, $\mathcal{C}_4$ and $\mathcal{C}_5$ differ only by the choice of the initial level,[37] there is no size difference, and yet it makes a huge difference in the run time of Spin, as discussed in the previous section.

Another translation difference evidently comes from the difference between the generalized automata obtained from the LTL formula. In our case $\mathcal{C}_4, \mathcal{C}_5$, and $\mathcal{C}_6$ were obtained from $\mathcal{G}_1$ while $\mathcal{C}_1$ and $\mathcal{C}_2$ were obtained from $\mathcal{G}_2$.[38] The difference between $\mathcal{G}_1$ and $\mathcal{G}_2$ is caused by choices made during the translation to favour deterministic states in the case of $\mathcal{G}_1$. In our example of Table 3.3, this improved determinism makes no difference since a and b are never true in the system.

3.3.4 *Automata for* $\neg(\mathsf{GF}a \implies \mathsf{GF}b)$

We now focus on another concrete case: $\neg(\mathsf{GF}a \implies \mathsf{GF}b)$ on mutex protocols. The formula without negation describes that if some process visits infinitely often the critical section, it infinitely often leaves it. This property holds in model `peterson.4.pm` and therefore Spin has to build the whole product to find out that it contains no accepting cycle. Table 3.4 presents results of Spin runs on the model `peterson.4.pm` and different Büchi automata for this formula.

In this case, each tool produces a different automaton, as shown in the first part of Figure 3.7. Note again that automata $\mathcal{D}_2$ and $\mathcal{D}_4$ cannot be distinguished only by determinism and size metrics (see Table 3.4). They differ only in the target of the outgoing edge of s_0, yet we observe a significant difference in Spin's behaviours.

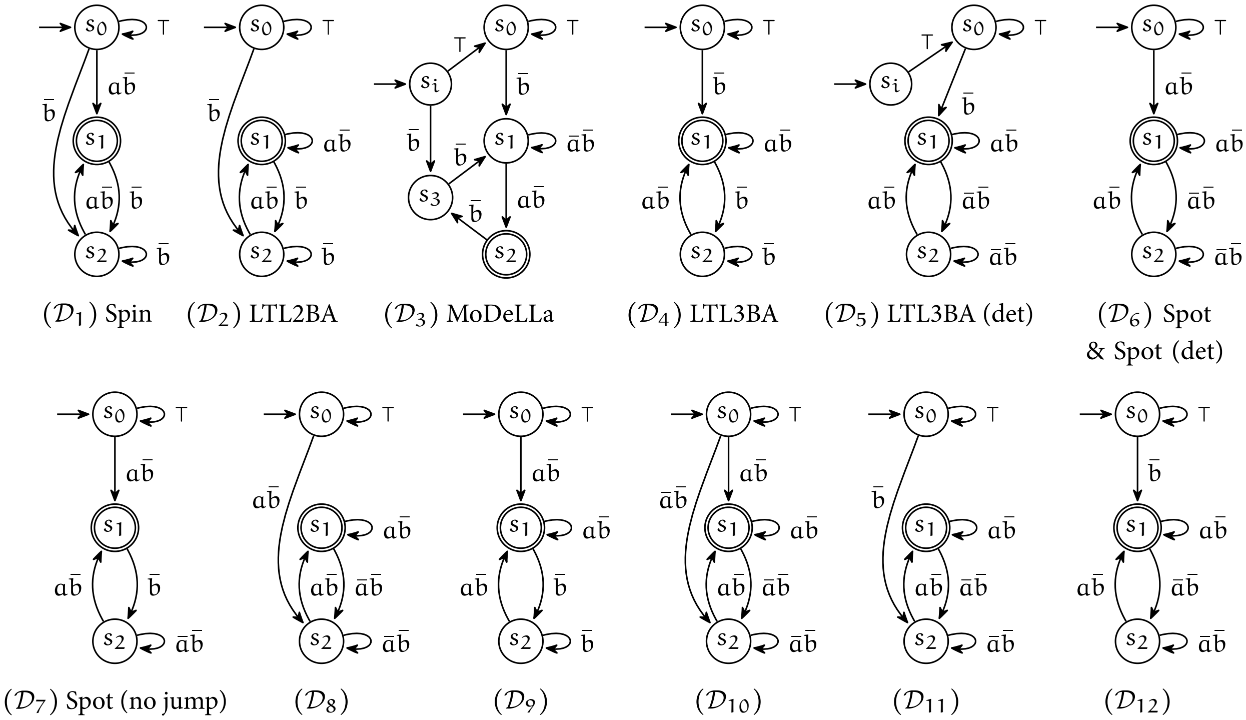
We actually use 12 different automata for this formula. The first seven of the table are generated by the considered tools. The others are handwritten by

[35] Gastin and Oddoux (2001), "Fast LTL to Büchi Automata Translation", [31].

[36] Giannakopoulou and Lerda (2002), "From States to Transitions: Improving Translation of LTL Formulae to Büchi Automata", [37].

[37] In fact, $\mathcal{C}_4$ and $\mathcal{C}_5$ differ also in degeneralization order but this is negligible as a and b are symmetric in our problem.

[38] The difference between $\mathcal{C}_1$ and $\mathcal{C}_2$ is that Spin ($\mathcal{C}_1$) performs no level jumping from the accepting state.



($\mathcal{D}_1$) Spin ($\mathcal{D}_2$) LTL2BA ($\mathcal{D}_3$) MoDeLLa ($\mathcal{D}_4$) LTL3BA ($\mathcal{D}_5$) LTL3BA (det) ($\mathcal{D}_6$) Spot & Spot (det)

($\mathcal{D}_7$) Spot (no jump) ($\mathcal{D}_8$) ($\mathcal{D}_9$) ($\mathcal{D}_{10}$) ($\mathcal{D}_{11}$) ($\mathcal{D}_{12}$)

Figure 3.7: Automata for the LTL formula $\neg(\mathsf{GF}a \implies \mathsf{GF}b)$.

	automaton size				statistics from Spin's execution		
	states	ndst	edges	trans	stored states	visited trans	time
$\mathcal{D}_1$ Spin	3	2	6	12	1577846	7680k	6.04s
$\mathcal{D}_2$ LTL2BA	3	3	6	12	1577440	7684k	5.95s
$\mathcal{D}_3$ MoDeLLa	5	2	8	18	1580893	7670k	6.13s
$\mathcal{D}_4$ LTL3BA	3	3	6	12	2299250	15583k	12.10s
$\mathcal{D}_5$ LTL3BA (det)	4	1	7	14	2297625	15561k	12.00s
$\mathcal{D}_6$ Spot	3	1	6	9	848641	2853k	2.26s
$\mathcal{D}_7$ Spot (no jump)	3	1	5	9	852094	2863k	2.34s
$\mathcal{D}_8$	3	1	6	9	848641	2853k	2.43s
$\mathcal{D}_9$	3	3	6	11	852094	2878k	2.43s
$\mathcal{D}_{10}$	3	1	7	10	1575844	7658k	7.38s
$\mathcal{D}_{11}$	3	1	6	10	1577440	7657k	7.07s
$\mathcal{D}_{12}$	3	1	6	10	2297625	15561k	12.30s

Table 3.4: Statistics about generated automata and Spin's run on the empty product model `peterson.4.pm` and automata for $\neg(\mathsf{GF}a \implies \mathsf{GF}b)$. The automata are shown in Fig. 3.7.

modifying the previous automata to explore which aspects of the automata make a significant difference in Spin's behaviour as described further.

$\mathcal{D}_8$ is adapted from $\mathcal{D}_6$ by changing the degeneralization level on which we enter the SCC. $\mathcal{D}_9$ keeps the strong initial guard of $\mathcal{D}_6$ but then uses the accepting SCC of $\mathcal{D}_2$. $\mathcal{D}_{10}$ is a mix of $\mathcal{D}_6$ and $\mathcal{D}_2$ to observe the influence of the guards $\bar{a}\bar{b}$ compared to $\bar{b}$. $\mathcal{D}_{11}$ is a version of $\mathcal{D}_2$ in which the SCC is made deterministic as in $\mathcal{D}_6$. Finally, $\mathcal{D}_{12}$ fixes $\mathcal{D}_5$ by removing the spurious initial state s_i.

Based on Table 3.4 we can group these automata into three categories, listed from the best to the worst with respect to Spin's performance. Before we discuss these categories, it is important to notice that in a model where a means *the process is in the critical section* and b means *the process leaves the critical section*, we can expect most of the state space to be labelled by $\bar{a}\bar{b}$.

$\mathcal{D}_6, \mathcal{D}_7, \mathcal{D}_8, \mathcal{D}_9$ Automata with the smallest number of transitions. Note that the *no jump* version ($\mathcal{D}_7$) and the one with a non-deterministic SCC ($\mathcal{D}_9$) both yields a few more states and transitions in the product, but the difference is not significant. The key property of these automata is that they can leave the state s_0 only by reading $a\bar{b}$, whereas other automata are more permissive.

$\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3, \mathcal{D}_{10}, \mathcal{D}_{11}$ All these automata exhibit more non-determinism on state s_0 and will enter the accepting SCC even after reading $\bar{a}\bar{b}$. However, when this happens, they do not reach the accepting state before $a\bar{b}$ is read, so this limits the number of red DFS.

$\mathcal{D}_4, \mathcal{D}_5, \mathcal{D}_{12}$ These automata go from s_0 to the accepting state s_1 each time they read $\bar{a}\bar{b}$. This both makes the product unnecessarily large and forces many calls to the red DFS.[39] The non-determinism in accepting SCC of $\mathcal{D}_4$ causes it to visits only slightly more states than the other two automata.

A comparison of automata $\mathcal{D}_6$ and $\mathcal{D}_{11}$ and their impact on Spin's performance show that the hypothesis of Section 3.3.2 cannot be used alone to select the best automaton. Indeed, $\mathcal{D}_6$ outperforms $\mathcal{D}_{11}$ even if the distance from the initial to the accepting state is shorter in $\mathcal{D}_6$. Here the more restrictive label of transition from s_0 to s_1 in $\mathcal{D}_6$ plays an important role as well. These automata demonstrate that we should both try to "improve the product" (Section 3.2) by using more restrictive labels for A, and keep accepting states as hard to reach as possible (compare $\mathcal{D}_{11}$ to $\mathcal{D}_{12}$).

To sum up, if we suppose that there is no accepting cycle in the product, the automaton should

1. keep accepting states as far as possible from the initial state (compare $\mathcal{D}_{11}$ to $\mathcal{D}_{12}$) and

2. use more restrictive labels (compare $\mathcal{D}_6$ to $\mathcal{D}_{12}$)

in order to make the accepting states as hard to reach as possible. Moreover, making use of more restrictive labels can also help to reduce the product.

[39] A state of the product has two components: a system's state and an automaton's state. Every time the blue DFS backtracks from a product's state with s_1 in the component for the property automaton.

3.4 SUMMARY OF THE CHAPTER

There is no such thing as a best Büchi automaton for explicit model checking. Although building a small product generally helps the emptiness check we have provided evidence that the size of $\mathcal{A}_\varphi$ and even the size of $\mathcal{S} \otimes \mathcal{A}_\varphi$ does not always correlate to the performance of the NDFS on the product. For instance, the locations of accepting states of $\mathcal{A}_\varphi$ can have a dramatic impact on the run time of Spin. Unfortunately, there is no single general rule we could give here. The right choice vastly depends on the particular verification task we aim to solve.

We show how can we tailor automata for particular system of the given verification task in Chapter 4 where we exploit some knowledge about the system. Without any such knowledge, we may at least predict the expected result of the model checking and based on this prediction we can at least place accepting states accordingly.

If $\mathcal{S} \otimes \mathcal{A}_\varphi$ contains no accepting cycle, the best automaton for Spin to verify it should have accepting states that are hard to reach from the initial state, as it will lessen the chance that a red DFS is started. We observed that such a choice can be made during the degeneralization procedure, or by unrolling some accepting cycles.

If, on the contrary, $\mathcal{S} \otimes \mathcal{A}_\varphi$ contains an accepting cycle, Spin can find it faster if the accepting states of $\mathcal{A}_\varphi$ are easy to reach from the initial state and the accepting cycles are short. Moreover, NDFS can benefit greatly from weak automata.

4 Specifications meet systems

In the previous chapter, we learnt that we can place accepting states of automata in a way that is helpful for Spin — under the assumption that we guess the expected result of the emptiness check correctly. If we were able to guess the result, we would not need to run the model checker, thus the assumption is unrealistic in practice.

In this chapter, we continue further with our campaign for ideal automata that are tailored for a particular verification task.[1] Our approach differs from the one of the previous chapter in three directions: we aim to build a smaller product rather than to make Nested DFS more effective, we build upon information about the system itself rather than on knowledge about the product, and we rely on information that we can acquire with only little effort for each system.

Spin verifies systems given in a modelling language called Promela. The Promela code is an implicit and compact representation of the system. A system in Promela consists of several interacting processes. The Figure 4.1 shows a skeleton of a process `P0` in the Promela language. The labels `NCS`, `waiting`, and `CS` are labels of process's *locations*, the process can move between locations by **`goto`** commands. For every process `P` and each location `loc`, Spin recognizes atomic propositions of the form `P@loc` which holds if the last location reached by `P` is `loc`.

```
active proctype P0() {
NCS: if
:: ...; goto waiting;
fi;
waiting: if
:: ...; goto CS;
fi;
...
CS: if
:: ...; goto NCS;
fi;
}
```

Figure 4.1: Skeleton of a code for a process `P0` that is used in the bakery mutual exclusion protocol description in the Promela language. Locations and the process name are in blue. The actual function code is left out for brevity.

A process cannot be in two different locations at the same time. Thus we say that the atomic propositions `P0@NCS`, `P0@waiting`, and `P0@CS` are mutually *incompatible* – no two of them can hold at the same time. Why do we care about incompatible propositions? Consider the LTL formula $\varphi = \mathsf{G}$ `P0@waiting` $\vee$ $\mathsf{FG}\neg$`P0@CS` and the two automata of Figure 4.2.[2] The two automata differ in the languages they accept. The left one accepts $L(\varphi)$ while the language of the right automaton accepts a smaller language: it accepts a subset of $L(\varphi)$. For example, the left automaton accepts the word $\{a, b\}^{\omega} = \{$`P0@waiting`, `P0@CS`$\}^{\omega}$ while the right one does not.[3] However, we can use them interchangeably for model checking of a system that contains `P0` without changing the result. Indeed, the languages of the two automata differ only in words that make no sense for systems that contain `P0` — the words are certainly not behaviours of the systems. Therefore such words are never present in the language of the product, and thus the language does not change.

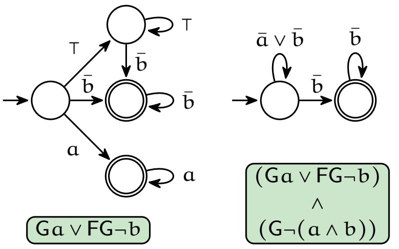

⊤
⊤
$\bar{b}$
$\bar{b}$
$\bar{b}$
a
a
$\bar{a} \vee \bar{b}$
$\bar{b}$
$\bar{b}$
$\mathsf{G}a \vee \mathsf{FG}\neg b$
$(\mathsf{G}a \vee \mathsf{FG}\neg b)$
$\wedge$
$(\mathsf{G}\neg(a \wedge b))$

Figure 4.2: Büchi Automata for $\mathsf{G}a \vee \mathsf{FG}\neg b$ produced by Spot without (left) and with (right) information about the incompatibility of propositions a and b.

Moreover, the automaton from the right would apparently lead to a smaller product. When we know that a and b are never valid at the same time, then $\mathsf{G}a$ implies $\mathsf{G}\neg b$ and thus also $\mathsf{FG}\neg b$. The right-hand side automaton makes use of this fact and checks only for $\mathsf{FG}\neg b$. Not only will the product be smaller, also the number of the red DFS runs will be lower with this automaton.

In the rest of this chapter we shall discuss formally how to use the information about incompatible propositions to *refine* the specification when it is given either by an LTL formula or by a Büchi automaton. We talk about

In fact, we can apply the results also to specifications given as PSL formulae or as other types of ω-automata.

[1] A verification task is a pair of a system and an LTL formula.

[2] Where a stands for `P0@waiting` and b stands for `P0@CS`.

[3] To be more precise, runs of the right automaton even blocks when reading $\{a, b\}$ for the first time.

formula refinement or *automaton refinement*, respectively. Both these operations were implemented by Alexandre Duret-Lutz in Spot 1.99.1, available at https://spot.lrde.epita.fr/.

Using refinement, we get a property automaton that may have fewer edges or even fewer states than the initial property automaton. All these changes often have a positive effect on the rest of the model checking process, as documented by experimental evaluation.

As a side effect of the specification refinement, we typically obtain automata with long edge labels. Besides the fact that such automata are harder to read by humans, Spin needs more time when building the product to evaluate these long edge labels. However, the labels explicitly contain the information about the incompatible propositions. As the information is already implicitly in the verified systems, we can employ the incompatibility of propositions to make these labels short again (and even shorter than they were originally).

The chapter is concluded by interesting cases discovered during our intensive experiments.

4.1 SPECIFICATION REFINEMENT AND CONSTRAINTS

The Promela code of the system implicitly describes an underlying automaton[4] for the system and the code already provides us with some relevant information about the automaton. In particular, we can detect that some combinations of propositions in $AP(\varphi)$ and their negations are never valid at the same time. We can express this information by a *constraint* κ, which is a Boolean formula over $AP(\varphi)$ satisfied by all combinations of atomic propositions except the invalid combinations.

For instance, `x>10`, `y<5`, and `x<y` cannot hold together. This information follows directly from the meaning of the atomic propositions and the related constraint is $\neg((\texttt{x>10}) \land (\texttt{y<5}) \land (\texttt{x<y}))$. As already discussed, atomic propositions saying that a process `P` is in various locations[5] are always incompatible. Moreover, they are even mutually exclusive. If E is a set of mutually exclusive atomic propositions, the corresponding constraint is:

$$\bigwedge_{\substack{a,b\in E\\ a\neq b}} \neg(a \land b)$$

While such constraints may seem obvious to the reader, tools that translate LTL formulae into Büchi automata do not analyze the semantics of atomic propositions, and thus they do not know that `x>10` and `x<4` are incompatible. It is the job of the refinement algorithms for formulae and for automata to make the constraint κ explicit for the tools and thus gain smaller automata.

The aforementioned examples of incompatible propositions can be easily detected: by an SMT solver or even better by a regular expression. A more complicated static analysis of the system can identify more impossible combinations. For instance, the analysis can find out that if a process `P` is in a location `loc`, then local variable `P:x` has value 0, and thus atomic propositions `P@loc` and `P:x>0` never hold together, expressed as $\neg((\texttt{P@loc}) \land (\texttt{P:x>0}))$. We do not focus on finding incompatible propositions; we show how this information can be used to improve model checking.

[4] It is, in fact, a Kripke structure. A Kripke structure can be seen as an automaton with labels on states instead of edges and with all states accepting.

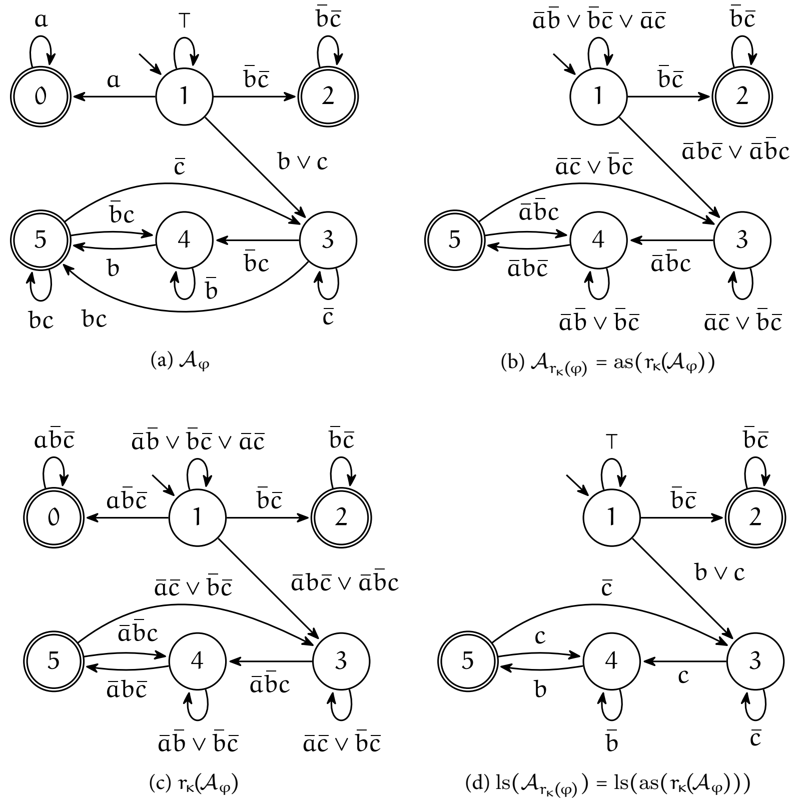
[5] For example `P@loc1`, `P@loc2`, and `P@loc3`

4.2 FORMULA REFINEMENT

The refinement of an LTL formula φ with respect to a constraint κ is a formula $r_\kappa(\varphi)$; it explicitly encodes κ into the formula and is defined by

$$r_\kappa(\varphi) = \varphi \wedge \mathsf{G}\kappa.$$

This extra information allows tools that translate LTL formulae into automata to produce smaller automata. For instance the Büchi automaton $\mathcal{A}_\varphi$ in Figure 4.3(a) was generated by Spot from the formula $\varphi = \mathsf{F}(\mathsf{G}a \vee (\mathsf{GF}b \iff \mathsf{GF}c))$. For the refined formula $r_\kappa(\varphi)$ using the constraint for the mutual exclusivity of $\{a, b, c\}$, Spot produced the automaton in Figure 4.3(b). This automaton is smaller: the edge between states 3 and 5 labelled by bc is known to be never satisfiable, and the state 0 is found to be superfluous.[6]


a
⊤
$\bar{b}\bar{c}$
0
a
1
$\bar{b}\bar{c}$
2
$\bar{c}$
$b \vee c$
$\bar{b}c$
5
4
3
b
$\bar{b}c$
$\bar{b}$
bc
bc
$\bar{c}$
$\bar{a}\bar{b} \vee \bar{b}\bar{c} \vee \bar{a}\bar{c}$
$\bar{b}\bar{c}$
1
$\bar{b}\bar{c}$
2
$\bar{a}\bar{c} \vee \bar{b}\bar{c}$
$\bar{a}b\bar{c} \vee \bar{a}\bar{b}c$
$\bar{a}\bar{b}c$
5
4
3
$\bar{a}b\bar{c}$
$\bar{a}\bar{b}c$
$\bar{a}\bar{b} \vee \bar{b}\bar{c}$
$\bar{a}\bar{c} \vee \bar{b}\bar{c}$
$a\bar{b}\bar{c}$
$\bar{a}\bar{b} \vee \bar{b}\bar{c} \vee \bar{a}\bar{c}$
$\bar{b}\bar{c}$
0
$a\bar{b}\bar{c}$
1
$\bar{b}\bar{c}$
2
$\bar{a}\bar{c} \vee \bar{b}\bar{c}$
$\bar{a}b\bar{c} \vee \bar{a}\bar{b}c$
$\bar{a}\bar{b}c$
5
4
3
$\bar{a}b\bar{c}$
$\bar{a}\bar{b}c$
$\bar{a}\bar{b} \vee \bar{b}\bar{c}$
$\bar{a}\bar{c} \vee \bar{b}\bar{c}$
⊤
$\bar{b}\bar{c}$
1
$\bar{b}\bar{c}$
2
$b \vee c$
$\bar{c}$
c
5
4
3
b
c
$\bar{b}$
$\bar{c}$

(a) $\mathcal{A}_\varphi$

(b) $\mathcal{A}_{r_\kappa(\varphi)} = \mathrm{as}(r_\kappa(\mathcal{A}_\varphi))$

(c) $r_\kappa(\mathcal{A}_\varphi)$

(d) $\mathrm{ls}(\mathcal{A}_{r_\kappa(\varphi)}) = \mathrm{ls}(\mathrm{as}(r_\kappa(\mathcal{A}_\varphi)))$

Figure 4.3: Automata without and with specification refinement for the LTL formula $\varphi = \mathsf{F}(\mathsf{G}a \vee (\mathsf{GF}b \iff \mathsf{GF}c))$ and the constraint $\kappa = \neg(a \wedge b) \wedge \neg(a \wedge c) \wedge \neg(b \wedge c)$.

[6] Indeed, the incoming edges of state 0 would be labelled by $a\bar{b}\bar{c}$, so that part of the automaton is covered by the state 2 already.

4.3 AUTOMATON REFINEMENT

Alternatively, the refinement can be performed directly on the property automaton which allows us to benefit from some known constraints even if we want to specify erroneous behaviours directly as an automaton. In order to refine a given automaton $\mathcal{A}$ by a constraint κ, we add κ in conjunction to all edge labels of $\mathcal{A}$ and remove the edge whenever the new label reduces to false. We denote the *refined automaton* by $r_\kappa(\mathcal{A})$.

It is equivalent to replacing every edge of $\mathcal{A}$ in the form (r_1, ℓ, r_2) by $(r_1, \ell \wedge \kappa, r_2)$.

Figure 4.3(c) shows a refined automaton for the automaton of Figure 4.3(a). In this case, state 0 is not removed. However, we can get rid of this state if we run some simplification algorithms, such as simulation-based reductions,[7] which are often employed in LTL to automata translators. The result of this simplification pass is then again in Figure 4.3(b). If $\mathrm{as}(\mathcal{A}_\varphi)$ is the result of the same simplifications which are used by the translator that translated φ to $\mathcal{A}_\varphi$, one would expect that the $\mathcal{A}_{r_\kappa(\varphi)} = \mathrm{as}(r_\kappa(\mathcal{A}_\varphi))$ always holds as in the example of Figure 4.3(b). This is not true in practice for two reasons:

- Some translators have LTL rewriting rules that may react strangely to the refined formula, sometimes to the point of producing larger automata.
- Some translators include automata simplification algorithms that can only be applied when the formula is known, so they cannot be run on arbitrary automata. For instance, Spot employs WDBA-minimization.[8]

Nonetheless, both formula refinement and automaton refinement have three noticeable effects on the model checking process:

- First, the automaton constructed with formula or automaton refinement is often smaller than the original automaton (for example, removing some transitions can make two states equivalent and such states can be merged). This can have a very positive effect on the model checking process.
- Second, if the unsatisfiable transitions are removed, Spin does not need to repeatedly evaluate the labels of these transitions during the product construction, only to finally ignore them.
- Last, the longer labels produced by this refinement may take longer to evaluate depending on how the model checker is implemented. This is the only negative effect, and we fix it in Section 4.5.

[7] Babiak et al. (2013), "Compositional Approach to Suspension and Other Improvements to LTL Translation", [44].

[8] Dax, Eisinger, and Klaedtke (2007), "Mechanizing the Powerset Construction for Restricted Classes of ω-Automata", [27]; Duret-Lutz (2014), "LTL Translation Improvements in Spot 1.0", [34].

4.4 EXPERIMENTAL EVALUATION

Tools. In our experiments, we use four LTL-to-BA translators presented in Table 4.1. Two of the translators, namely LTL3BA and Spot, are used with two settings: the default ones and the settings with the suffix "-det" that aim to produce more deterministic automata. All translators are restricted by 20-minute timeout. For formula refinement, automaton refinement, and automaton simplifications we use the tools `ltlfilt` and `autfilt` from Spot 1.99.1; see examples of the corresponding commands below where $\varphi = \mathsf{F}(\mathsf{G}a \lor (\mathsf{GF}b \iff \mathsf{GF}c))$, $\mathcal{A}$ is always stored in `input.hoa`, and κ is the constraint for the mutually exclusive set $\{a, b, c\}$. If there are several mutually exclusive sets, one can use `--exclusive-ap` multiple times.

```
% ltlfilt -f 'F(Ga | (GFb <-> GFc))' --exclusive-ap='a,b,c'
F(Ga | (GFb <-> GFc)) & G(!(a & b) & !(a & c) & !(b & c))
% autfilt --exclusive-ap='a,b,c' input.aut
% autfilt --high --small input.hoa
```

Command to build $r_\kappa(\varphi)$ from φ and its output in grey.

Command to build $r_\kappa(\mathcal{A})$ from $\mathcal{A}$.

Command to simplify $\mathcal{A}$.

The emptiness checks of Spin was run with the maximum search depth of 100 000 000, memory limit 20 GiB, the option `-DNOSTUTTER`,[9] and partial-order reduction enabled for tasks with *next*-free formulae. The emptiness check is always restricted by 30-minute timeout.

You can find the exact commands, the measured data and detailed information about this benchmark at http://fi.muni.cz/~xstrejc/publications/spin2015/

Benchmark. Our benchmark is made of 3316 verification tasks where some propositions are referring to distinct locations of a single process. We started with 789 verification tasks[10] from Beem[11] and we removed 8 duplicate tasks. Unfortunately, Beem contains only about 25 different types of specification formulae[12] and most of them have a very simple structure. To get more varied formulae, we added verification tasks using the same Beem systems and randomly generated formulae.

We generated these additional tasks as follows. For each instance of a Beem system,[13] we generated 10 000 random formulae using the tool `randltl` from Spot. More precisely, we ran

```
% randltl -n10000 -tree-size=30..50 <list of propositions>
```

where the atomic propositions were gathered from all original Beem formulae for the corresponding instance. For each such verification task, we ran Spot 1.2.5 to translate the formula into a Büchi automaton and then we ran Spin with the settings as described above. We selected verification tasks where

translator	version	command
Spin [21; 26]	6.3.2	`spin`
LTL2BA [31]	1.1	`ltl2ba`
LTL3BA [33]	1.1.2	`ltl3ba`
LTL3BA-det		`ltl3ba -M0`
Spot [34]	1.99b	`ltl2tgba -s`
Spot-det		`ltl2tgba -s --deterministic`

Table 4.1: Considered LTL-to-BA translators, for reference. The reference of Spin is valid also for the model checker.

[9] See Section 4.6.3 for the explanation.

[10] A verification task is a pair of a Promela code of a system and an LTL formula describing erroneous behaviours.

[11] Pelánek (2007), "BEEM: Benchmarks for Explicit Model Checkers", [30].

[12] the others differ only in atomic propositions or their combinations

[13] 23 parametric systems, altogether 133 instances

- Spot translates the formula within 20 minutes,
- Spin's verifier finished in more than 5 seconds and less than 30 minutes, and
- Spin neither reached maximum search depth nor ran out of memory.

The two timeouts of Spot in Table 4.2 can be explained either by the fact that we used an older version (1.2.5 vs 1.99b) to generate the formulae from the tasks or by a time that is very close to the 20 minutes threshold.

We got 6069 generated verification tasks with random formulae. For each verification task (original or generated), we constructed exclusive sets based on atomic propositions referring to process locations. The constraints we used for specification refinement are therefore based only on the fact that one process cannot be in two locations at the same time. We removed all verification tasks for which we did not detect such constraints.

In the end, we have 3316 verification tasks of reasonable complexity and with constraints. These tasks employ 101 instances of 16 parametrized systems from Beem. Of all the tasks, 50 are from Beem, the rest use generated formulae.

Hardware. All computations were performed on the same machine as the experiments from the previous chapter. The machine was shared with other users and its variable workload has again led to high dispersion of measured run times. Hence, instead of run times, we use the number of transitions visited by Spin, which is stable across multiple executions and should be proportional to the run time.

The CSV file with the measured data from the URL also contains the measured time. It could be used to draw the same conclusions as we did using the visited transitions.

4.4.1 Impact of Formula Refinement

For each verification task $(\mathcal{S}, \varphi)$ and each translator of Table 4.1, we translate φ to an automaton $\mathcal{A}_\varphi$ and run Spin on $\mathcal{S}$ and $\mathcal{A}_\varphi$ (original task). Then we refine the formula to $r_\kappa(\varphi)$ and repeat the process (refined task). Table 4.2 shows the numbers of translation timeouts, Spin fails,[14] and successfully solved verification problems. The data indicate that formula refinement has a mostly positive effect on the model checking process: for all but one translator,[15] the refinement increases the number of successfully solved tasks. Nevertheless, the number of tasks solved both with and without formula refinement is always smaller than the number of original tasks successfully solved. This means that the effect of the formula refinement is negative in some cases.

	original tasks $(\mathcal{S}, \varphi)$			refined tasks $(\mathcal{S}, r_\kappa(\varphi))$			
translator	translation timeouts	Spin fails	tasks solved	translation timeouts	Spin fails	tasks solved	both tasks solved
Spin	801	232	2283	926	201	2189	2183
LTL2BA	5	341	2970	2	302	3012	2929
LTL3BA	0	80	3236	0	55	3261	3227
LTL3BA-det	0	34	3282	0	27	3289	3279
Spot	2	27	3287	0	19	3297	3286
Spot-det	2	26	3288	0	19	3297	3287
All	810	740	18346	928	623	18345	18191

Table 4.2: Statistics of fails and successfully solved verification tasks with and without formula refinement.

[14] This number covers the cases when Spin timeouts, runs out of memory, or reaches the maximum search depth

[15] We discuss the case of the translator Spin in more details in Section 4.6.2.

Table 4.3 shows that the property automaton for a refined formula frequently has fewer states than the automaton for the original formula. However, we cannot easily tell whether states are removed simply because they are inaccessible after refinement (i.e., the constraint κ removed all the transitions leading to a state) or if the refinement enabled additional simplifications as in Figure 4.3. In the former case, the refinement would have a little impact on the size of the product: it is only saving useless attempts to synchronize transitions that can never be synchronized while building this product.

effect	Spin	LTL2BA	LTL3BA	LTL3BA-det	Spot	Spot-det
+states	514	41	15	148	13	17
−states	168	1482	1679	1723	1722	1720
=states,+edges	37	17	0	0	9	10
=states,−edges	43	337	293	326	345	344
=states,=edges,+trans.	153	211	283	173	280	280
=states,=edges,−trans.	1226	785	899	848	849	848
no size change	42	56	58	61	68	68
All	2183	2929	3227	3279	3286	3287

Table 4.3: Effect of formula refinement on property automata. For each translator and each verification task, we compare the size of $\mathcal{A}_\varphi$ with the size of $\mathcal{A}_{r_\kappa(\varphi)}$ and report on the number of cases where the refinement resulted in additional states (+states) or fewer states (−states). In case of equality, we look at the number of edges or transitions. For each translator we consider only the tasks from the last column of Table 4.2, which are tasks solved both with and without formula refinement.

Finally, we turn our attention to the actual effect of formula refinement on the performance of the emptiness check implemented in Spin. For each translator and each verification task, let t_1 be the number of transitions visited by Spin for the original task and t_2 be the same number for the refined task. Scatter plots in Figure 4.4 on the page 50 show each pair (t_1, t_2) as a dot at this coordinate. The color of each dot says whether the property automaton for the refined formula has more or fewer states than the automaton for the original formula. The data is shown separately for each translator. We also distinguish the tasks with some erroneous behaviour from those without error. As many dots in the scatter plots are overlapping, we present the data also via *improvement ratios* t_2/t_1. Values of t_2/t_1 smaller than 1 correspond to cases where formula refinement actually helped Spin, while values larger than 1 correspond to cases where the refinement caused Spin to work more.

Figure 4.5 gives an idea of the distribution of these improvement ratios in our benchmark. In this figure, all improvement ratios for a given tool are sorted from lowest to highest, and then they are plotted using their rank as x coordinate and using a logarithmic scale for the ratio. One can immediately see on these curves that there is a large plateau around $y = 1$ corresponding to the cases where there is no substantial change. Among the tasks without error, there are usually many cases with the ratio below 0.95 (a definite improvement), and very few cases above 1.05 (cases where refinement hurts more than it helps). A special class of cases that are improved are those that are found equivalent to false after refinement: those usually have a very high improvement ratio, as the exploration of the product is now limited to a single transition.[16] The refined formula cannot be equivalent to false in tasks with an error. Relatively high numbers of these “false” cases imply that the formula refinement technique is an effective sanity check detecting specifications unsatisfiable under given constraints. Table 4.4 gives counts of improvement ratios in these classes.

The high number of “false” cases is due to the use of random formulae. In real tasks, such a *false* case would likely indicate a bug in the specification.

[16] Spin immediately realizes that the empty automaton cannot be satisfied

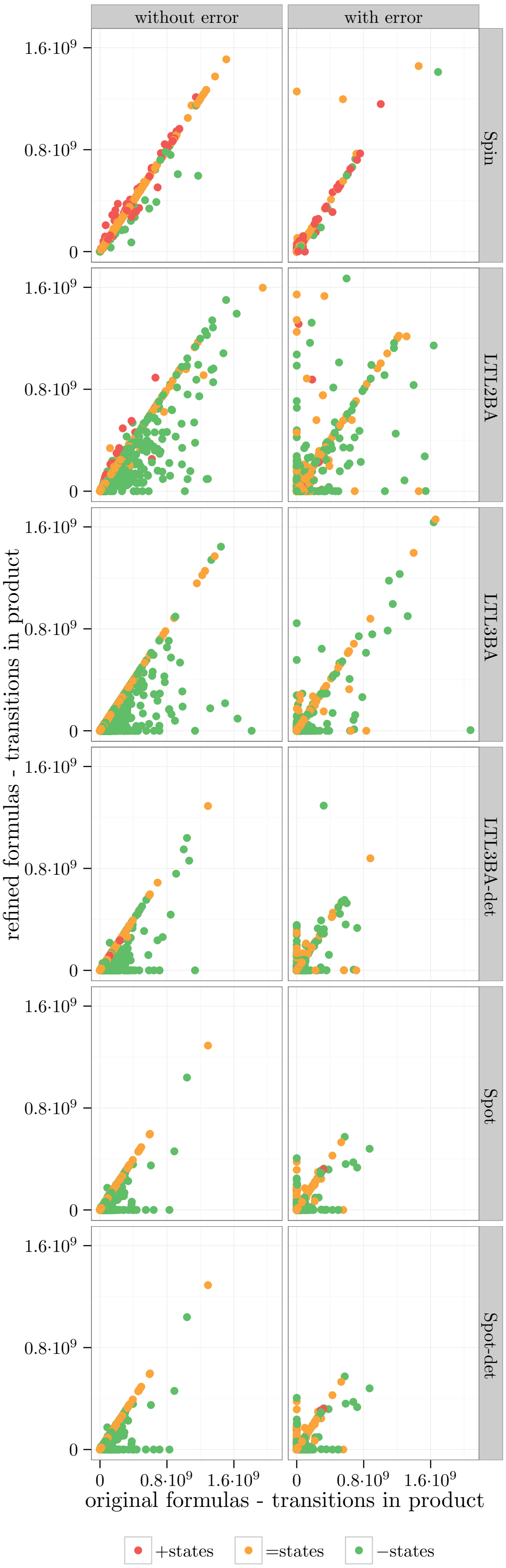


Figure 4.4: Comparison of the numbers of product transitions visited by Spin on the original tasks (t_1) and their *formula*-refined versions (t_2).

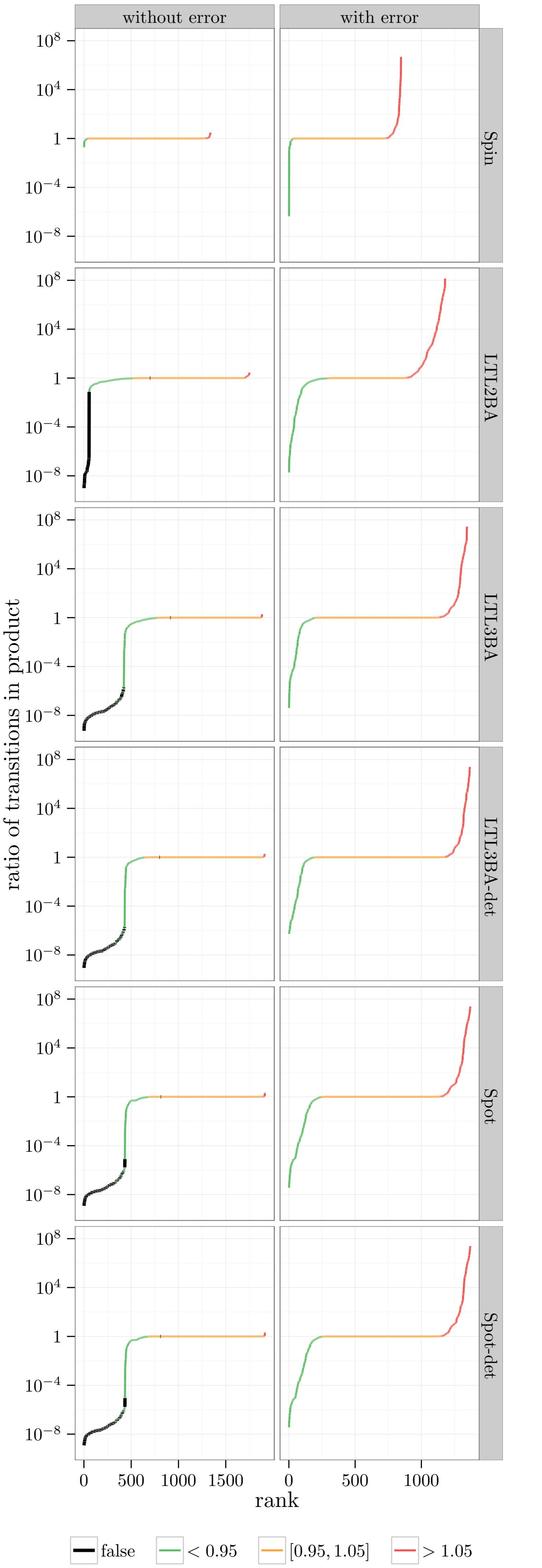


Figure 4.5: Distribution of the improvement ratios (t_2/t_1). Cases that have been reduced to false are highlighted in bold. **Note log scale.**

Figures 4.4 and 4.5 and Table 4.4 show that for tasks without error, formula refinement has a negative effect[17] only very rarely and such effect is relatively small. The positive effect is more frequent and substantial in many cases. The table implies that LTL3BA and Spot can profit more from the refinement as they identify radically more false cases and they have significantly less cases with negative effect than the other translators. This observation can be explained by advanced simplification techniques implemented in LTL3BA and Spot.

You can find more detailed data that relate the effect on automata and on model checking in Tables 4.12 and 4.13 on the pages 58 and 59.

	without error					with error			
	false	<0.95	[0.95,1.05]	>1.05	All	<0.95	[0.95,1.05]	>1.05	All
Spin	0	30	1257	50	1337	27	708	111	846
LTL2BA	61	462	1179	48	1750	288	602	289	1179
LTL3BA	374	401	1101	7	1883	194	942	208	1344
detLTL3BA	382	264	1255	12	1913	186	993	187	1366
Spot	384	300	1213	20	1917	244	902	223	1369
detSpot	385	297	1218	18	1918	248	903	218	1369
All	1586	1754	7223	155	10718	1187	5050	1236	7473

Table 4.4: Distribution of the improvement ratios for formula refinement. The counts of false cases are not included in the <0.95 classes.

In the tasks with erroneous behaviours, we observe that the number of improved cases is almost balanced by the number of degraded cases (except for Spin). This can be explained by the fact that refining an LTL formula may alter the shape of the output automaton, and thus change its transition order. Therefore the model checker may have more or less luck in finding an erroneous run. When such a run is found, Spin ends the computation without exploring the rest of the product.

Table 4.5 shows that measuring the number of transitions explored by Spin instead of time has no effect on conclusions. There are only 93 (out of 18 191) tasks where refinement improved the number of explored transitions but Spin needed more time. However, this is caused mainly by the unreliable measurements of the run times, which is obvious in the cases where the formula was reduced to false and Spin still needed more time to evaluate the task after refinement.

	time ratio					
	without error			with error		
trans ratio	<0.95	[0.95,1.05]	>1.05	<0.95	[0.95,1.05]	>1.05
false	1552	14	20	0	0	0
<0.95	1552	159	43	1006	131	50
[0.95, 1.05]	492	2358	4373	617	2263	2170
>1.05	1	4	150	79	221	936

Table 4.5: Relation of change in the number of transitions to the change in the measured run time (unreliable) of Spin.

[17] Some of the negative effects are discussed in Section 4.6.

4.4.2 *Impact of Automaton Refinement*

As mentioned before, automaton refinement itself only cuts off some parts of the automaton that are not used in the product. It has a bigger effect only when additional simplification algorithms are executed after the refinement. In our experiments, we combined the automaton refinement with automaton simplifications implemented in Spot.

To measure the effect of automaton refinement, we prepared the benchmark as follows. We took the 3316 verification tasks used before. For every task, we translated the formula with all considered translators and simplified the produced automata using Spot – that is the automaton $\mathcal{A}$. The simplification is here applied to make the comparison of model checking with and without automaton refinement fair: without this step, we could not really distinguish the effect of automaton refinement (followed by simplifications) from the effect of simplifications themselves. If the automaton translation and simplification successfully finishes, we get a pair of a system and a simplified automaton (original task). After removing duplicates, we have 9352 original tasks.

For each task, we run Spin with the original automaton. Then we refine and simplify the automaton and run Spin again. While the automaton refinement is very cheap, the successive simplification can be quite expensive. So we apply a 20-minute timeout to simplifications. Table 4.6 provides numbers of Spin fails on original tasks, timeouts of refined automata simplifications, and Spin failures on refined tasks. In the following, we work only with tasks solved both with and without automaton refinement.

original tasks $(\mathcal{S}, \mathcal{A})$		refined tasks $(\mathcal{S}, \mathrm{as}(r_\kappa(\mathcal{A})))$			
Spin fails	tasks solved	simplification of $r_\kappa(\mathcal{A})$ timeouts	Spin fails	tasks solved	both tasks solved
291	9061	12	99	9241	9038

Table 4.6: Statistics of fails and successfully solved verification tasks with and without automata refinement.

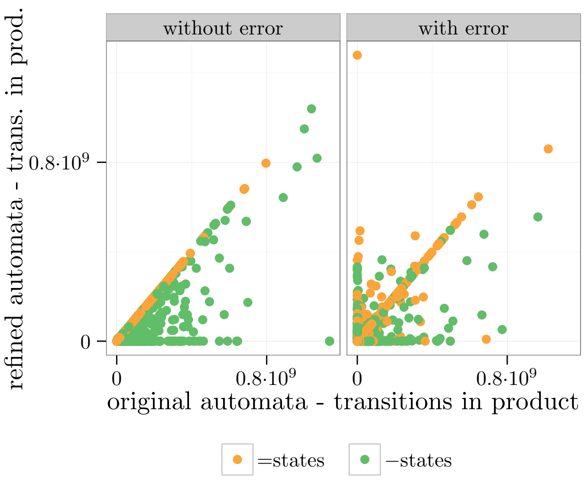


Figure 4.6: Comparison of the numbers of product transitions visited by Spin on the original tasks (t_1) and their *automata*-refined versions (t_2).

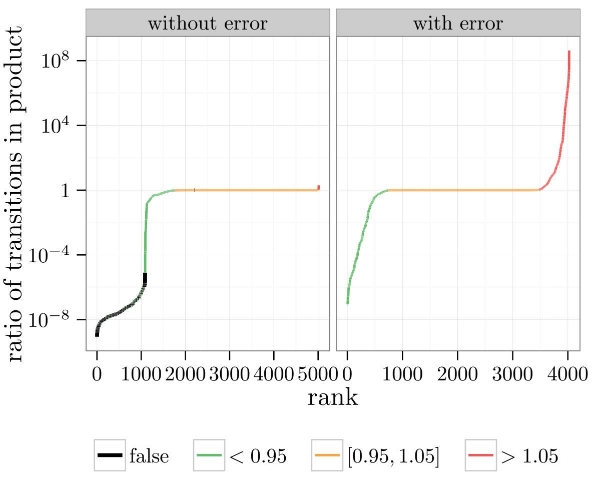


Figure 4.7: Distribution of the improvement ratios (t_2/t_1). Cases that have been reduced to false are highlighted in bold. **Note log scale.**

As in the previous section, Table 4.7 presents the effect of automaton refinement and simplification on the sizes of property automata. The refined and simplified automata are smaller in the vast majority of cases and never bigger.

The effect of automaton refinement and simplification on the performance of the emptiness check in Spin is presented in Figures 4.6 and 4.7, and Table 4.8 in the same way as previously. On tasks without error, the effect is similar to formula refinement: it is often positive and almost never negative. On tasks with error, the positive effect is more frequent than the negative one.

effect	
+states	0
−states	4955
=states,+edges	0
=states,−edges	1013
=states,=edges,+trans.	0
=states,=edges,−trans.	2400
no size change	670

Table 4.7: Effect of automaton refinement on property automata.

	without error	with error
false	906	0
< 0.95	853	735
$[0.95, 1.05]$	3251	2743
> 1.05	5	545
All	5015	4023

Table 4.8: Distribution of the improvement ratios for automaton refinement.

4.4.3 *Comparison of Formula and Automaton Refinement*

Here we compare the formula refinement and automaton refinement using Spot for the formula translation. For each of the 3316 considered tasks, we refine the formula, translate it by Spot, and run Spin. Then we take the task again, translate the original formula by Spot, refine and simplify the automaton, and run Spin. Table 4.9 provides statistics about automata construction timeouts,[18] Spin timeouts, and solved tasks. Both approaches detected 380 identical cases where the refined specification reduces to false. In the following, we present the data from the 3256 – 380 = 2876 tasks solved by both approaches and not trivially equivalent to false.

tasks with formula refinement			tasks with automaton refinement			
automaton construction timeouts	Spin fails	tasks solved	automaton construction timeouts	Spin fails	tasks solved	both tasks solved
0	19	3297	35	25	3256	3256

Table 4.9: Statistics of fails and successfully solved verification tasks with formula refinement and automaton refinement.

Tables 4.10 and 4.11 and Figures 4.8 and 4.9 are analogous to the tables and figures in the previous sections (the position of original tasks in the previous sections is taken by tasks with formula refinement). Table 4.10 says that automaton refinement often produces property automata with more states than formula refinement. However, Figure 4.8 and Table 4.11 show that the overall effect of automata and formula refinement on the performance of Spin is fully comparable, slightly in favour of formula refinement.

effect	
+states	315
−states	82
=states,+edges	52
=states,−edges	51
=states,=edges,+trans.	26
=states,=edges,−trans.	428
no size change	1922

Table 4.10: Comparison of automata produced by formula refinement and automaton refinement (+states counts tasks where $as(r_\kappa(\mathcal{A}_\varphi))$ has more states than $\mathcal{A}_{r_\kappa(\varphi)}$ and so on).

4.5 LABEL SIMPLIFICATION

As mentioned in Section 4.1, a side-effect of specification refinement is that edges get more complex labels. This is visible when comparing the automaton of Figure 4.3(b) to the one of Figure 4.3(a). For example, the self-loop on state 3 is labelled by $\bar{a}\bar{c} \vee \bar{b}\bar{c}$ instead of the original $\bar{c}$. In our experiment, the overall average length of an edge label (counted as the number of occurrences of atomic propositions in the label) in the automata $\mathcal{A}_{r_\kappa(\varphi)}$ for refined formulae is 6.58, while the average label length in the corresponding automata $\mathcal{A}_\varphi$ for unrefined formulae is only 4.20. Spin compiles the labels during the construction of the product into C code that matches the system transitions. For example, Figure 4.10 depicts the C code corresponding to the labels $\bar{a}\bar{c} \vee \bar{b}\bar{c}$ and $\bar{c}$. Clearly, longer labels can slow down the verification process without in-

	without error	with error
< 0.95	44	133
$[0.95, 1.05]$	1399	970
> 1.05	71	259
All	1514	1362

Table 4.11: Distribution of the improvement ratios for automaton refinement over formula refinement.

[18] This number comprises Spot timeouts and also simplification of refined automata timeouts in the case of automaton refinement

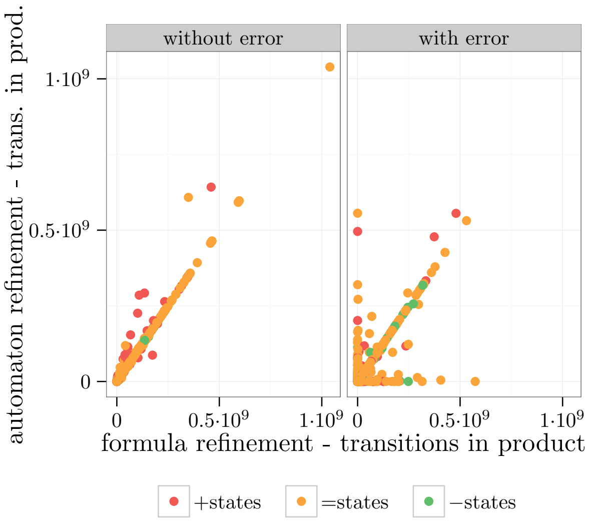


Figure 4.8: Comparison of the numbers of product transitions visited by Spin in *formula*-refined tasks (t_1) and their *automata*-refined versions (t_2).

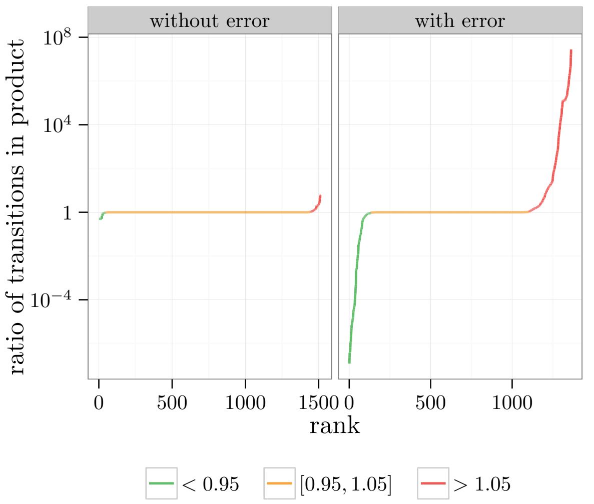


Figure 4.9: Distribution of the improvement ratios (t_2/t_1). Cases that have been reduced to false are highlighted in bold. **Note the log scale.**

fluencing any Spin statistics like visited transitions and stored states. However, the expected slowdown should be only small as checking the labels is much cheaper than computing the successors of states of the system or storing the states.

```
if (!(((!((((int)((P1 *)Pptr(f_pid(1)))->_p) == 27))&&
        !((((int)((P1 *)Pptr(f_pid(1)))->_p) == 5)))||
        (!((((int)((P1 *)Pptr(f_pid(1)))->_p) == 27))&&
        !((((int)((P1 *)Pptr(f_pid(1)))->_p) == 9)))))) ...

if (!( !((((int)((P1 *)Pptr(f_pid(1)))->_p) == 27)))) ...
```

Figure 4.10: Parts of two `pan.m` files that Spin generates when it checks a system against two automata of Figure 4.3. The upper part encodes an edge of $\mathcal{A}_{r_\kappa(\varphi)}$ labelled by $\bar{a}\bar{c} \vee \bar{b}\bar{c}$ and the last line represents an analogous edge of $\mathcal{A}$ with label $\bar{c}$.

We eliminate this slowdown by a step that resembles a converse of refinement. Refinement uses the given constraint κ to make edge labels more precise (restrictive). Label simplification uses κ to make the edge labels *less* precise and shorter, but equivalent to the original labels under the constraint κ. For instance, $b\bar{c}$ can be shortened to b if we know that b and c never hold together in the system. The edge label is in fact a Boolean function and we can simplify these based on so-called *don't care*[19] information. Concretely, we have implemented the simplification in Spot using the Minato-Morreale algorithm.[20] The algorithm takes two Boolean functions $\lfloor f \rfloor$ and $\lceil f \rceil$ and produces a minimal label that covers at least all the assignments satisfying $\lfloor f \rfloor$, and that is not satisfiable by at least all the assignments not satisfying $\lceil f \rceil$. To simplify a label ℓ using a constraint κ, we call this algorithm with $\lfloor f \rfloor = \ell \wedge \kappa$ and $\lceil f \rceil = \ell \vee \neg\kappa$.

```
% autfilt --exclusive-ap='a,b,c' \
          --simplify-exclusive-ap input.hoa
```

Command that simplifies labels of $\mathcal{A}$.

Figure 4.3(d) shows the result of label simplification (denoted as function ls) applied to Figure 4.3(b).

We applied the label simplification to all automata obtained by formula refinement and the average label length dropped to 3.19, which is even lower than 4.20 which is the value for automata without refinement. We selected

[19] We do not care if the simplified label additionally covers some variable assignments that can never happen in the system.

[20] Minato (1993), "Fast Generation of Prime-Irredundant Covers from Binary Decision Diagrams", [45].

several cases with high reduction of label length and run Spin several times with automata before and after label simplification on a weaker, but isolated machine to get reliable run times. In these tests, Spin runs up to 3.5% faster after label simplification.

4.6 WHEN REFINEMENT HARMS AND FOUND BUGS

In few cases, specification refinement decreased the performance of Spin. We have identified three origins of these situations.

4.6.1 The Case of Strongly Connected Components

Figure 4.11 shows one of the few tasks without error where the refined formula translated by Spot degrades the performance of Spin. Spin performs better with the automaton $\mathcal{A}_\varphi$ (Figure 4.11(a)) than with the smaller automaton $\mathcal{A}_{r_\kappa(\varphi)}$ (Figure 4.11(b)).

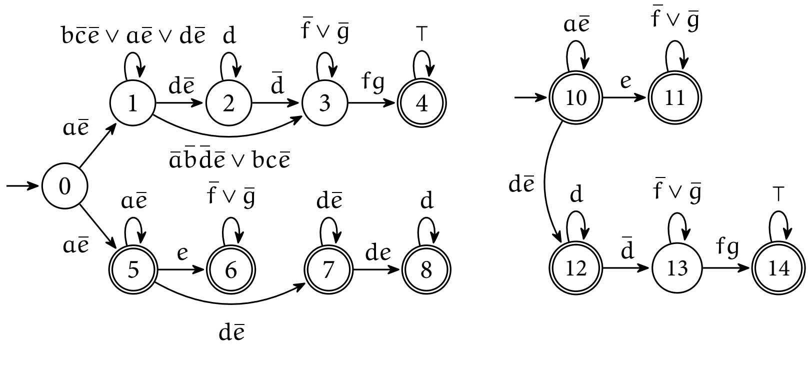
The reason why Spin works better with the larger of these two automata was already discussed in the previous chapter. It is related to the sensitivity of Nested DFS algorithm to the location of accepting states. In the automaton of Figure 4.11(b) the state 12 is accepting. Whenever the blue DFS backtracks a state of the product that is synchronized with state 12, it has to start a red DFS that will explore again the states synchronized with 12 and 13 previously explored by the blue DFS.

The re-exploration of states synchronized with 13 is something that

1. did not happen in the original automaton because there is no accepting state preceding the corresponding state 3, and

2. is useless because there is no way to get back to state 12 after moving to state 13.

The NDFS algorithm could be patched to avoid this problem by simply constraining the red DFS to explore only the states of the product whose projection on the property automaton belongs to the same strongly connected component as its starting accepting state. This optimization was already suggested by Edelkamp et al. with one additional trick:if we know that the current SCC is weak,[21] then running a red DFS is not needed at all as the blue DFS is guaranteed to find any accepting cycle by itself.[22] In the scenarios described by Figures 4.11(a) and 4.11(b), all the SCCs have a single state, so the product



(a) useful part of $\mathcal{A}_\varphi$

(b) $\mathcal{A}_{r_\kappa(\varphi)}$

Figure 4.11: An uncommon case where $\mathcal{A}_{r_\kappa(\varphi)}$ is much smaller than $\mathcal{A}_\varphi$, and yet Spin performs better with $\mathcal{A}_\varphi$.

The automaton presented in Figure 4.11(a) is a pruned version of the real automaton. We have removed all transitions that do not appear in the product with the system. For instance, in this pruned automaton it is obvious that the state 7 can be merged with the state 8, but the presence of other edges in the original automaton prevented this simplification.

States synchronized with 14 are ignored as they have been already seen by a previous red DFS.

[21] All states of a weak SCC are accepting or all are non-accepting.

[22] Edelkamp, Lluch-Lafuente, and Leue (2001), "Directed Explicit Model Checking with HSF-SPIN", [46]; Edelkamp, Leue, and Lluch-Lafuente (2004), "Directed Explicit-State Model Checking in the Validation of Communication Protocols", [47].

automaton will be weak and the red DFS should not be needed. Computing the strongly connected components of the property automaton can be done in time that is linear to the size of that automaton (typically a small value) before the actual emptiness check starts, so this is a cheap way to improve the model checking time.

4.6.2 *Problems with LTL simplifications*

A special class of interesting cases consists of formulae where formula refinement leads to bigger automata. Such cases are surprisingly often connected with issues in the earliest phases of LTL to automata translation, namely in formula parsing or simplification. For example, LTL3BA implements several specific formula reduction rules applied after all standard formula reductions. If such a rule is applied, the reduced formula is checked again for possible application of some reduction rule, but in LTL3BA version 1.1.2 it was checked only on the top level of the formula. Hence, some reductions were not applied when the input formula was refined with a constraint. This was a bug and was fixed in version 1.1.3.

LTL2BA has even more problems with formula simplifications as it is sensitive to superfluous parentheses. For instance, the command `ltl2ba -f '<>([]<>X p)'` generates an automaton with 2 states, while the equivalent `ltl2ba -f '<>[]<>X p'` produces an automaton with 4 states. This is due to the fact that LTL2BA runs another simplification pass in the presence of parentheses.

The operator <> represents F and [] represents G in LTL2BA.

Table 4.3 indicates that Spin's translator benefits less than the other translators from the addition of constraints. Part of the problem, it seems, is due to a change that was introduced in Spin 6 to allow LTL formulae embedding atomic propositions with arbitrary Promela conditions. As a consequence of this change, many parenthetical blocks are now considered as atomic propositions by Spin's translator, and simplifications are therefore missed. For instance, the formula $(a \mathsf{R} b) \wedge \mathsf{G}(\neg(a \wedge b))$ is translated as if $\neg(a \wedge b)$ was an independent atomic proposition. While Spin 5 translates this formula into an automaton with one state and one edge, Spin 6 outputs an automaton with two states and three edges, where the edge connecting the states has unsatisfiable label $\neg(a \wedge b) \wedge a \wedge b$.

4.6.3 *Problem with Spin*

During our experiments, we discovered a handful of cases where equivalent automata would cause Spin to produce different results: e.g., a counterexample for automata built by some tools, and no counterexamples for (equivalent) automata built by other tools. Sometimes the automata would differ only by the order in which the transitions are listed. In turned out that this bug[23] was due to a rare combination of events in the red DFS in the presence of a deadlock in the system. All the presented results are computed by compiling the Spin 6.3.2 verifier with `-DNOSTUTTER`, which effectively means that we ignore deadlock scenarios, and we are safe from this bug.

[23] http://spinroot.com/fluxbb/viewtopic.php?pid=3316, fixed by Spin 6.4.4

4.7 FINAL REMARKS

We only considered incompatibilities between atomic propositions that denote a process being in different locations in our evaluation. More sources of incompatibilities could be considered, such as atomic propositions that refer to different variable values. We could also extend the principle to more than just incompatible propositions: for instance from the system we could extract information about the validity of atomic propositions in the initial state, the order of locations in a process, or learn the fact that some variable will always be updated in a monotonous way (e.g., can only be increased). All these information can be used to produce stricter property automata that ignore these impossible behaviours, and we think these automata should offer more opportunity for simplifications, and should also contribute to better sanity checks.

We demonstrated the usefulness of refinement in model checking. We believe it should also be useful in other contexts like probabilistic model checking or controller synthesis.

	effect on automata	ratio	Spin	LTL2BA	LTL3BA	LTL3BA-det	Spot	Spot-det	All
without error	+states	false	0	0	0	0	0	0	0
		<0.95	8	4	0	2	0	0	14
		[0.95,1.05]	254	6	3	109	2	2	376
		>1.05	41	14	0	7	0	0	62
	-states	false	0	54	367	375	377	378	1551
		<0.95	21	415	396	258	292	289	1671
		[0.95,1.05]	86	548	372	528	475	482	2491
		>1.05	0	20	3	5	16	14	58
	=states,+edges	false	0	0	0	0	0	0	0
		<0.95	0	2	0	0	0	0	2
		[0.95,1.05]	16	8	0	0	3	3	30
		>1.05	5	2	0	0	0	0	7
	=states,-edges	false	0	0	0	0	0	0	0
		<0.95	1	17	5	4	5	5	37
		[0.95,1.05]	16	179	153	156	187	185	876
		>1.05	0	0	0	0	0	0	0
	=states,=edges,+trans	false	0	0	0	0	0	0	0
		<0.95	0	7	0	0	0	0	7
		[0.95,1.05]	95	114	167	75	163	163	777
		>1.05	4	4	4	0	4	4	20
	=states,=edges,-trans	false	0	0	0	0	0	0	0
		<0.95	0	17	0	0	3	3	23
		[0.95,1.05]	754	284	363	343	333	333	2410
		>1.05	0	8	0	0	0	0	8
	no size change	false	0	7	7	7	7	7	35
		<0.95	0	0	0	0	0	0	0
		[0.95,1.05]	36	40	43	44	50	50	263
		>1.05	0	0	0	0	0	0	0
with error	+states	false	0	0	0	0	0	0	0
		<0.95	9	8	2	21	0	2	42
		[0.95,1.05]	150	5	5	7	9	10	186
		>1.05	52	4	5	2	2	3	68
	-states	false	0	0	0	0	0	0	0
		<0.95	15	174	167	135	190	189	870
		[0.95,1.05]	43	179	258	321	269	270	1340
		>1.05	3	92	116	101	103	98	513
	=states,+edges	false	0	0	0	0	0	0	0
		<0.95	1	3	0	0	0	4	8
		[0.95,1.05]	12	2	0	0	3	1	18
		>1.05	3	0	0	0	3	2	8
	=states,-edges	false	0	0	0	0	0	0	0
		<0.95	0	18	10	24	22	21	95
		[0.95,1.05]	15	82	95	116	96	98	502
		>1.05	11	41	30	26	35	35	178
	=states,=edges,+trans	false	0	0	0	0	0	0	0
		<0.95	1	13	3	0	4	4	25
		[0.95,1.05]	50	59	104	90	100	100	503
		>1.05	3	14	5	8	9	9	48
	=states,=edges,-trans	false	0	0	0	0	0	0	0
		<0.95	1	71	12	6	28	28	146
		[0.95,1.05]	432	268	472	449	414	413	2448
		>1.05	39	137	52	50	71	71	420
	no size change	false	0	0	0	0	0	0	0
		<0.95	0	1	0	0	0	0	1
		[0.95,1.05]	6	7	8	10	11	11	53
		>1.05	0	1	0	0	0	0	1

Table 4.12: More precise data about formula refinement impact on automata and on model checking. For each combination of effect on automata (+states means increase number of states) and each category of ratio of visited transitions *(with refinement/without refinement)* we show number of corresponding cases for each LTL-to-BA translator. As usual, we keep the cases with error (counterexample) and without error (the whole product explored) apart.

	ratio	effect on automata	Spin	LTL2BA	LTL3BA	LTL3BA-det	Spot	Spot-det	All
without error	false	+states	0	0	0	0	0	0	0
		-states	0	54	367	375	377	378	1551
		=states,+edges	0	0	0	0	0	0	0
		=states,-edges	0	0	0	0	0	0	0
		=states,=edges,+trans	0	0	0	0	0	0	0
		=states,=edges,-trans	0	0	0	0	0	0	0
		no size change	0	7	7	7	7	7	35
	<0.95	+states	8	4	0	2	0	0	14
		-states	21	415	396	258	292	289	1671
		=states,+edges	0	2	0	0	0	0	2
		=states,-edges	1	17	5	4	5	5	37
		=states,=edges,+trans	0	7	0	0	0	0	7
		=states,=edges,-trans	0	17	0	0	3	3	23
		no size change	0	0	0	0	0	0	0
	[0.95,1.05]	+states	254	6	3	109	2	2	376
		-states	86	548	372	528	475	482	2491
		=states,+edges	16	8	0	0	3	3	30
		=states,-edges	16	179	153	156	187	185	876
		=states,=edges,+trans	95	114	167	75	163	163	777
		=states,=edges,-trans	754	284	363	343	333	333	2410
		no size change	36	40	43	44	50	50	263
	>1.05	+states	41	14	0	7	0	0	62
		-states	0	20	3	5	16	14	58
		=states,+edges	5	2	0	0	0	0	7
		=states,-edges	0	0	0	0	0	0	0
		=states,=edges,+trans	4	4	4	0	4	4	20
		=states,=edges,-trans	0	8	0	0	0	0	8
		no size change	0	0	0	0	0	0	0
with error	false	+states	0	0	0	0	0	0	0
		-states	0	0	0	0	0	0	0
		=states,+edges	0	0	0	0	0	0	0
		=states,-edges	0	0	0	0	0	0	0
		=states,=edges,+trans	0	0	0	0	0	0	0
		=states,=edges,-trans	0	0	0	0	0	0	0
		no size change	0	0	0	0	0	0	0
	<0.95	+states	9	8	2	21	0	2	42
		-states	15	174	167	135	190	189	870
		=states,+edges	1	3	0	0	0	4	8
		=states,-edges	0	18	10	24	22	21	95
		=states,=edges,+trans	1	13	3	0	4	4	25
		=states,=edges,-trans	1	71	12	6	28	28	146
		no size change	0	1	0	0	0	0	1
	[0.95,1.05]	+states	150	5	5	7	9	10	186
		-states	43	179	258	321	269	270	1340
		=states,+edges	12	2	0	0	3	1	18
		=states,-edges	15	82	95	116	96	98	502
		=states,=edges,+trans	50	59	104	90	100	100	503
		=states,=edges,-trans	432	268	472	449	414	413	2448
		no size change	6	7	8	10	11	11	53
	>1.05	+states	52	4	5	2	2	3	68
		-states	3	92	116	101	103	98	513
		=states,+edges	3	0	0	0	3	2	8
		=states,-edges	11	41	30	26	35	35	178
		=states,=edges,+trans	3	14	5	8	9	9	48
		=states,=edges,-trans	39	137	52	50	71	71	420
		no size change	0	1	0	0	0	0	1

Table 4.13: More precise data about formula refinement impact on automata and on model checking, in comparison to Table 4.12 the columns *effect on automata* and *ratio* are swapped.

Part II

LTL TO DETERMINISTIC AUTOMATA

5 Translation of LTL Fragments into Generalized Rabin Automata

This chapter presents a translation of an LTL fragment into deterministic automata. The translation is influenced by the successful LTL to NBA translation algorithm of LTL2BA[1], however, it avoids the notoriously difficult determinization of Büchi automata. The inspiration is reflected in our two-step approach.

1. A given LTL formula φ is translated into a *linear alternating automaton (LAA)*[2] $\mathcal{A}_\varphi$ as in LTL2BA. For the considered fragment, the LAA satisfies an additional structural condition; we call such automata *may/must alternating automata (MMAA)*.
2. The MMAA $\mathcal{A}$ is translated into a *deterministic generalized Rabin automaton* $\mathcal{G}$ with marks on transitions.

We also show that with just a little tweak, the construction is correct even for a slightly larger fragment.

Chatterjee et al. showed that it mostly pays off to use the generalized form of Rabin automata.[3] However, for the sake of completeness, we offer a procedure that translates our DTGRA into the commonly used Rabin automata with marks on states in Section 5.6.

LTL Fragments. In this chapter, we consider two LTL fragments. We start with the fragment $\mathrm{LTL}(\mathsf{F_s}, \mathsf{G_s})$ whose formulae are built with temporal operators $\mathsf{F_s}$, $\mathsf{G_s}$, F, and G only ($\mathsf{F}\varphi$ and $\mathsf{G}\varphi$ can be seen as abbreviations for $\varphi \vee \mathsf{F_s}\varphi$ and $\varphi \wedge \mathsf{G_s}\varphi$, respectively). Later we show that our translation is correct also for the fragment $\mathrm{LTL}{\setminus}\mathsf{G}(\mathsf{U}, \mathsf{X})$. The name of the fragment comes from the fact that there is no U and X in the scope of any G and the fragment is defined as

$$\varphi \ ::= \ \psi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mathsf{X}\varphi \mid \varphi \, \mathsf{U} \, \varphi,$$

where ψ ranges over $\mathrm{LTL}(\mathsf{F_s},\mathsf{G_s})$. This fragment is strictly more expressive than $\mathrm{LTL}(\mathsf{F_s},\mathsf{G_s})$.

Remark on related work. We discuss other fragments and other related translations of LTL (or its fragments) into deterministic automata in the next chapter.

5.1 Alternating automata and their subclasses

Alternating automata. An *alternating automaton* $\mathcal{A} = (S, \Sigma, \Delta, I, M, \mu, \Phi)$ is a tuple where S, Σ, M, μ and Φ have the same meaning as in ω-automata, $I \subseteq 2^S$ is a non-empty set of *initial configurations*, and $\Delta \subseteq S \times \Sigma \times 2^S$ is an *alternating transition relation*. In general, subsets $C \subseteq S$ are called *configurations*. We use analogous terminology for transitions as in ω-automata. Moreover, for

The meaning of μ is the same in the sense that it places marks on states and transitions. However, while in ω-automata the type of μ is $\mu\colon M \to 2^{S \cup \delta}$, here it is $\mu\colon M \to 2^{S \cup \Delta}$.

[1] Gastin and Oddoux (2001), “Fast LTL to Büchi Automata Translation”, [31].

[2] Also known as *very weak*, *1-weak*, or *self-loop* alternating automata.

[3] Chatterjee, Gaiser, and Křetínský (2013), “Automata with Generalized Rabin Pairs for Probabilistic Model Checking and LTL Synthesis”, [48].

a transition $t = (s, \alpha, C)$ we call s the *source state* and C the *target configuration* of t. The transition is *looping* (or simply a *loop*) if $s \in C$ and t is a *self-loop* if $C = \{s\}$. A *semitransition* of t is every triple (s, α, s') such that $s' \in C$.

Linear alternating automata. The alternating automaton $\mathcal{A}$ is *linear (LAA)* if there exists a partial order relation on the set of states such that for every transition $(s, \alpha, C) \in \Delta$ it holds that all states in C are lower or equal to s. In other words, there are no simple cycles with more than one transition.

A simple cycle does not visit any state except the first one twice.

Visualization. Figure 5.1 shows a linear alternating automaton that accepts the language of the LTL formula $\varphi = \mathsf{G}(\mathsf{F_s}a \wedge \mathsf{F_s}b) \vee \mathsf{G}b$. Transitions are depicted by branching edges, each branch of such edge corresponds to a semitransition. If a target configuration is empty, the corresponding edge leads to an empty space. Transitions that differ only in labels are grouped in the same way as in ω-automata. Each initial configuration is represented by a possibly branching unlabelled edge leading from an empty space to the states of the configuration.

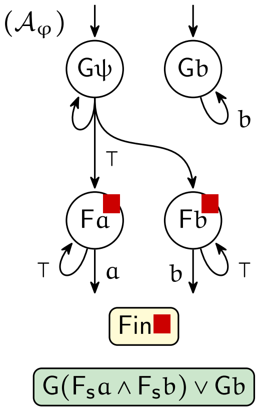


Figure 5.1: An LAA (and also MMAA) $\mathcal{A}_\varphi$ for $L(\varphi)$; $\varphi = \mathsf{G}(\mathsf{F_s}a \wedge \mathsf{F_s}b) \vee \mathsf{G}b$.

Multitransitions. A multitransition T under α is a set of transitions under α such that the source states of the transitions are pairwise different. The *source configuration* $\mathsf{source}(T)$ of T is the set of source states of transitions in T, the *target configuration* $\mathsf{target}(T)$ of T is the union of the target configurations of the transitions, and $\lambda(T) = \alpha$ is the *label* of T. The set of all multitransitions is denoted by $\Gamma^{\mathcal{A}}$, and $\Gamma^{\mathcal{A}}_\alpha$ stands for all multitransitions of $\mathcal{A}$ under α. We write Γ and Γ_α when $\mathcal{A}$ is clear from the context. Further, we use $C_1 \xrightarrow{\alpha}_{\mathcal{A}} C_2$ to denote that there is a multitransition $T \in \Gamma^{\mathcal{A}}_\alpha$ such that $C_1 = \mathsf{source}(T)$ and $C_2 = \mathsf{target}(T)$. Again, we leave out the $\mathcal{A}$ if $\mathcal{A}$ is clear from the context.

Runs. A run of $\mathcal{A}$ over a word $u = u_0u_1 \ldots \in \Sigma^\omega$ is an infinite sequence $\pi = T_0T_1 \ldots \in \Gamma^\omega$ of multitransitions such that $\mathsf{source}(T_0) \in I$ and for all $i \geq 0$ we have $\lambda(T_i) = u_i$ and $\mathsf{target}(T_i) = \mathsf{source}(T_{i+1})$. A branch b of π is a maximal (finite or infinite) sequence of consecutive semitransitions $b = (s_0, u_0, s_1)(s_1, u_1, s_2) \ldots$ where $s_{i+1} \in C_i$ for the transition $(s_i, u_i, C_i) \in T_i$ starting in s_i. The semitransitions have the marks of their parent transitions, and analogously to a run of ω-automata, the set $\mathsf{marks}(b)$ is the set of recurrent marks of b. A branch b satisfies Inf● if ● $\in \mathsf{marks}(b)$ and it satisfies Fin■ if ■ $\notin \mathsf{marks}(b)$. The run is accepting iff all its infinite branches satisfy Φ. The language of $\mathcal{A}$ is the set $L(\mathcal{A})$ of all words $u \in \Sigma^\omega$ such that $\mathcal{A}$ has an accepting run over u.

A multitransition T has exactly one transition for each $s \in \mathsf{source}(T)$.

A branch of a run of an alternating automaton is reminiscent of a run of an ω-automaton.

Runs visualization. Runs of alternating automata can be visualized as a directed acyclic graphs (DAG). Figure 5.2 shows a run of $\mathcal{A}_\varphi$ over the word $(\{a\}\emptyset\{b\}\{a,b\})^\omega$. The dotted lines divide the DAG into segments corresponding to multitransitions. Each transition of a multitransition is represented by edges leading across the corresponding segment from the source state to states of the target configuration. Branches in the DAG correspond to branches of the run. State of an LAA can be ordered in a way that all edges in the DAG go only to the same or a lower row.

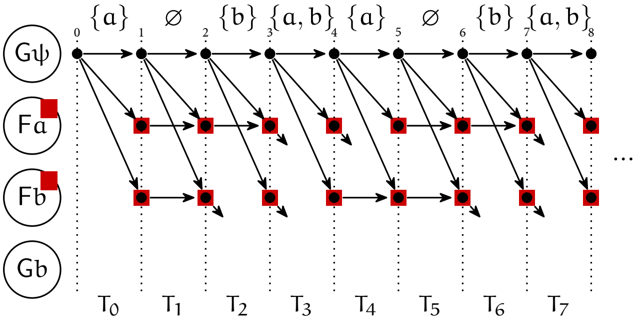


Figure 5.2: A run of the LAA $\mathcal{A}_\varphi$ from Figure 5.1 over $(\{a\}\emptyset\{b\}\{a,b\})^\omega$.

May/must alternating automata. An LAA is a *may/must alternating automaton (MMAA)* if each state fits into one of the following three categories:

1. *May-states* – states with a self-loop for each $\alpha \in \Sigma$ and at least one non-looping transition.

A run that enters such a state *may* wait in the state for an arbitrary number of steps.

2. *Must-states* – states with at least one transition and with looping transitions only.

A run that enters such a state can never leave it. In other words, the run *must* stay there.

3. *Loopless states* – states that have no looping transitions and no predecessors. They can appear only in initial configurations (or they are unreachable).

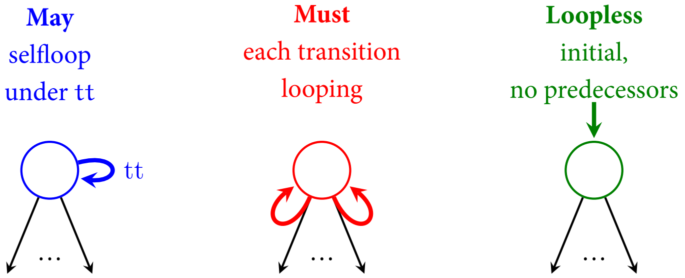


Figure 5.3: Illustration of state types of MMAA. The specific properties of the types are highlighted by distinct colors.

The automaton of Figure 5.1 is an MMAA with may-states Fa and Fb, must-states Gψ and Gb, and no loopless states.

In this thesis we consider only MMAA with marks on states and with co-Büchi acceptance; that is automata with a unique mark ■ and the acceptance formula $\mathrm{Fin}\blacksquare$. Moreover, we always set $\mu(\blacksquare)$ to the set of all may-states of the automaton. This is justified by the following observations:

- There are no looping transitions of loopless states. Hence, removing all loopless states from $\mu(\blacksquare)$ has no effect on the acceptance of any run.

Each branch of a run can visit at most one loopless state.

- All transitions leading from must-states are looping. Hence, if a run contains a must-state that is in $\mu(\blacksquare)$, then the run is non-accepting. Removing all must-states in $\mu(\blacksquare)$ together with their adjacent transitions from an MMAA has no effect on its accepting runs.
- Every may-state has self-loops for all $\alpha \in \Sigma$. If such a state is not in $\mu(\blacksquare)$, we can always apply these self-loops without violating acceptance of any run. We can also remove these states from all the target configurations of all transitions of an MMAA without affecting its language.

The class of MMAA with co-Büchi acceptance and with marks on states is expressively equivalent to the LTL fragment LTL($\mathsf{F_s}$,$\mathsf{G_s}$).

5.2 TRANSLATION OF LTL($\mathsf{F_s}$,$\mathsf{G_s}$) TO MMAA

Our translation follows the standard translation of LTL to LAA implemented in the tool LTL2BA.[4] Here we present a restriction of their translation to the fragment LTL($\mathsf{F_s}, \mathsf{G_s}$) only. In this section, we treat the transition relation $\Delta \subseteq S \times \Sigma \times 2^S$ of an LAA as a function $\Delta : S \times \Sigma \to 2^{2^S}$, where $C \in \Delta(s, \alpha)$ means $(s, \alpha, C) \in \Delta$. Further, we consider $\mathsf{G}\psi$ and $\mathsf{F}\psi$ to be subformulae of $\mathsf{G_s}\psi$ and $\mathsf{F_s}\psi$, respectively.

$\mathsf{G_s}\psi \equiv \mathsf{XG}\psi$ and $\mathsf{F_s}\psi \equiv \mathsf{XF}\psi$

Let φ be an LTL($\mathsf{F_s}, \mathsf{G_s}$) formula in positive normal form. An equivalent LAA is constructed as $\mathcal{A}_\varphi = (S, \Sigma, \Delta, I, \{$🟥$\}, \mu, \mathsf{Fin}$🟥$)$, where

- S is the set of subformulae of φ,
- $\Sigma = 2^{AP(\varphi)}$,
- Δ deserves more space and explanation and is thus defined below,
- $I = \overline{\varphi}$ where $\overline{\psi}$ represents a disjunctive normal form of ψ in a set notation that we compute for ψ as

$$\begin{aligned} \overline{\psi} &= \{\{\psi\}\} \text{ if } \psi \text{ is a temporal formula} \\ \overline{\psi_1 \vee \psi_2} &= \overline{\psi_1} \cup \overline{\psi_2} \\ \overline{\psi_1 \wedge \psi_2} &= \{C_1 \cup C_2 \mid C_1 \in \overline{\psi_1} \text{ and } C_2 \in \overline{\psi_2}\}, \text{ and} \end{aligned}$$

- μ maps 🟥 to the set of all subformulae of the form $\mathsf{F}\psi$ in S.

Transition function. Configurations in $\Delta(\psi, \alpha)$ stand for conjunctions of subformulae that, for ψ to be satisfied, have to hold in the next step if α holds now. Each configuration $\Delta(\psi, \alpha)$ is one possible way to satisfy ψ.

$$\begin{aligned} \Delta(\top, \alpha) &= \{\emptyset\} \\ \Delta(\neg\top, \alpha) &= \emptyset \\ \Delta(a, \alpha) &= \begin{cases} \{\emptyset\} & \text{if } a \in \alpha \\ \emptyset & \text{otherwise} \end{cases} \\ \Delta(\neg a, \alpha) &= \begin{cases} \{\emptyset\} & \text{if } a \notin \alpha \\ \emptyset & \text{otherwise} \end{cases} \\ \Delta(\psi_1 \vee \psi_2, \alpha) &= \Delta(\psi_1, \alpha) \cup \Delta(\psi_2, \alpha) \\ \Delta(\psi_1 \wedge \psi_2, \alpha) &= \{C_1 \cup C_2 \mid C_1 \in \Delta(\psi_1, \alpha) \text{ and } C_2 \in \Delta(\psi_2, \alpha)\} \\ \Delta(\mathsf{G_s}\psi, \alpha) &= \{\{\mathsf{G}\psi\}\} \\ \Delta(\mathsf{F_s}\psi, \alpha) &= \{\{\mathsf{F}\psi\}\} \\ \Delta(\mathsf{G}\psi, \alpha) &= \{C \cup \{\mathsf{G}\psi\} \mid C \in \Delta(\psi, \alpha)\} \\ \Delta(\mathsf{F}\psi, \alpha) &= \{\{\mathsf{F}\psi\}\} \cup \Delta(\psi, \alpha) \end{aligned}$$

All branches that follow the transition into $\emptyset$ (which happens when the branches are in s and $\emptyset \in \Delta(s, u_i)$ for the next u_i) terminate and become finite. A run where all branches are finite is accepting.

A sequence of multitransitions that hits a state s with empty $\Delta(s, u_i)$ for the next u_i blocks and does not form a run.

States for conjunctions and disjunction are never reachable.

States for $\mathsf{F_s}\psi$ and $\mathsf{G_s}\psi$ as well as states for $\top$, a, and their negations are reachable if and only if they are in I.

$\mathsf{G}\psi \equiv \psi \wedge \mathsf{XG}\psi$

$\mathsf{F}\psi \equiv \psi \vee \mathsf{XF}\psi$

Figure 5.4 shows this translation applied to formula $\varphi \mathsf{G}(\mathsf{F_s}a \wedge \mathsf{F_s}b) \vee \mathsf{G}b$.

[4] Gastin and Oddoux (2001), [31].

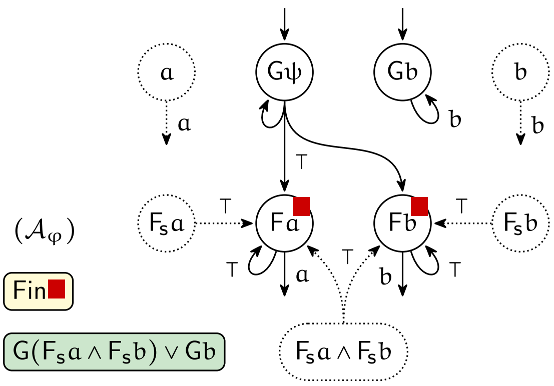


Figure 5.4: An MMAA $\mathcal{A}_\varphi$ for the formula $\varphi = \mathsf{G}\psi \vee \mathsf{G}b$ for $\psi = \mathsf{F_s}a \wedge \mathsf{F_s}b$ before removing the unreachable (dotted) states.

Using the partial order "is a subformula of" on states, one can easily prove that $\mathcal{A}_\varphi$ is an LAA. Moreover, all the states of the form $\mathsf{G}\psi$ are must-states and all the states of the form $\mathsf{F}\psi$ are may-states. States of other formulae are loopless, and they are unreachable unless they appear in I. Hence, the constructed automaton is also an MMAA. Figure 5.1 shows an MMAA produced by the translation of formula $\mathsf{G}(\mathsf{F_s}a \wedge \mathsf{F_s}b) \vee \mathsf{G}b$.

Theorem 5.1. *For each formula* $\varphi \in \mathrm{LTL}(\mathsf{F_s}, \mathsf{G_s})$, *we can construct an MMAA* $\mathcal{A}_\varphi$ *with at most* $|\varphi|$ *states such that* $L(\varphi) = L(\mathcal{A}_\varphi)$.

$|\varphi|$ denotes the length of φ.

We have shown that the translation yields an MMAA. The correctness of the construction was proved by Oddoux in his PhD thesis.[5]

5.3 TRANSLATION OF MMAA TO LTL($\mathsf{F_S}$,$\mathsf{G_S}$)

In this section, we show the reverse translation to the one of the previous section – from MMAA to $\mathrm{LTL}(\mathsf{F_s}, \mathsf{G_s})$. We assume that may-states have no looping transitions except self-loops. The assumption is valid as any application of a looping transition that is not a self-loop can always be replaced by an application of a self-loop with the same label; this change cannot transform an accepting run into a non-accepting one and thus the looping transitions of may-states that are not self-loops can be removed without altering the language of the automaton.

Let $\mathcal{A} = (S, 2^{AP'}, \Delta, I, \{■\}, \mu, \mathsf{Fin}■)$ be an MMAA with a propositional alphabet. For each $\alpha \in 2^{AP'}$ we define ψ_α to be a formula satisfied exactly by all the words starting with α:

$$\psi_\alpha = \Big(\bigwedge_{a\in\alpha} a\Big) \wedge \Big(\bigwedge_{a\in AP'\smallsetminus\alpha} \neg a\Big)$$

Now we inductively define a formula φ_s for each state $s \in S$. The formula φ_s is satisfied by any word for which there is an accepting run of $\mathcal{A}$ starting in the configuration $\{s\}$. The inductive definition is admissible because $\mathcal{A}$ is an LAA and thus there is a partial order on S such that transitions of a state s can lead only to s or states that are lower than s.

$$\varphi_s = \begin{cases} \mathsf{F} \bigvee_{\substack{(s,\alpha,C)\in\Delta \\ C\neq\{s\}}} \big(\psi_\alpha \wedge \bigwedge_{q\in C} \mathsf{X}\varphi_q\big) & \text{if } s \text{ is a may-state} \\ \mathsf{G} \bigvee_{(s,\alpha,C)\in\Delta} \big(\psi_\alpha \wedge \bigwedge_{q\in C\smallsetminus\{s\}} \mathsf{X}\varphi_q\big) & \text{if } s \text{ is a must-state} \\ \bigvee_{(s,\alpha,C)\in\Delta} \big(\psi_\alpha \wedge \bigwedge_{q\in C} \mathsf{X}\varphi_q\big) & \text{if } s \text{ is a loopless state} \end{cases}$$

The conjunction of an empty set of conjuncts is $\top$ while the disjunction of an empty set of disjuncts is $\neg\top$.

[5] Oddoux (2003), "Utilisation des Automates Alternants pour un Model-Checking Efficace des Logiques Temporelles Linéaires", [49].

Finally, we define the formula $\varphi_{\mathcal{A}}$ equivalent to the whole automaton $\mathcal{A}$ as

$$\varphi_{\mathcal{A}} = \bigvee_{C \in I} \bigwedge_{s \in C} \varphi_s.$$

Each temporal operator X in the definition of φ_s is in front of F or G. If we replace all occurrences of XF by $\mathsf{F_s}$ and all occurrences of XG by $\mathsf{G_s}$ in φ_s for all states s we always get that $\varphi_{\mathcal{A}}$ is an LTL($\mathsf{F_s}, \mathsf{G_s}$) formula. Hence, we have shown that the following theorem holds.

Theorem 5.2. *For each MMAA $\mathcal{A}$ with a propositional alphabet, we can construct an LTL$(\mathsf{F_s}, \mathsf{G_s})$ formula $\varphi_{\mathcal{A}}$ such that $\mathsf{L}(\mathcal{A}) = \mathsf{L}(\varphi_{\mathcal{A}})$.*

5.4 TRANSLATION OF MMAA TO DETERMINISTIC AUTOMATA

Let $\mathcal{A} = (S, \Sigma, \Delta, I, \{$🟥$\}, \mu, \mathsf{Fin}$🟥$)$ be an MMAA. First, we build a deterministic semiautomaton $\mathcal{T}$ that follows all possible runs of $\mathcal{A}$. Subsequently, we equip $\mathcal{T}$ with an acceptance condition and build a deterministic generalized Rabin automaton $\mathcal{D}$ such that $\mathsf{L}(\mathcal{D}) = \mathsf{L}(\mathcal{A})$.

5.4.1 Semiautomaton $\mathcal{T}$

The idea behind the construction of the deterministic semiautomaton is based on a double powerset construction: the run σ of the semiautomaton $\mathcal{T}$ over a word u tracks all runs of $\mathcal{A}$ over u. More precisely, the state of $\mathcal{T}$ reached after reading a finite input consists of all possible configurations in which $\mathcal{A}$ can be after reading the same input. Hence, states of the semiautomaton are sets of configurations of $\mathcal{A}$ and we call them *macrostates*.

One powerset construction is for dealternation, and the other is for determinization of the MMAA.

We use $s, s_1, s_2, \ldots$ to denote states of $\mathcal{A}$; $C, C_1, C_2, \ldots$ to denote configurations of $\mathcal{A}$; and $m, m_1, m_2, \ldots$ to denote macrostates of $\mathcal{T}$. Further, we use $t, t_1, t_2 \ldots$ to denote the transitions of $\mathcal{A}$; $T, T_0, T_1 \ldots$ to denote multitransitions of $\mathcal{A}$; and $r, r_1, r_2 \ldots$ to denote the transitions of $\mathcal{T}$. Finally, we use $\sigma(u)$ to denote the unique run of $\mathcal{T}$ over u.

Formally, we define the deterministic semiautomaton $\mathcal{T} = (Q, \Sigma, \delta, m_I)$ for $\mathcal{A}$ as follows:

- $Q \subseteq 2^{2^S}$ is the set *macrostates*, restricted to those reachable from the initial macrostate m_I by δ,
- $(m_1, \alpha, m_2) \in \delta$ iff $m_2 = \{C_2 \mid C_1 \in m_1, C_1 \xrightarrow{\alpha} C_2\}$
- $m_I = I$ is the *initial macrostate*.

For each $m_1 \in Q$ and $\alpha \in \Sigma$, there is a single transition to a macrostate m_2 that consists of target configurations of multitransitions labelled by α with source configurations in m_1. We say that (m_1, α, m_2) covers these multitransitions.

Figure 5.5 depicts the semiautomaton $\mathcal{T}$ for the MMAA of Figure 5.1. Each line in a macrostate represents one configuration.

5.4.2 Generalized Rabin Automaton $\mathcal{D}$

Now we are heading towards a deterministic generalized Rabin automaton $\mathcal{D} = (Q, \Sigma, \delta, m_I, M, \mu', \Phi)$. On top of the semiautomaton $\mathcal{T}$ we add a set of marks, place the marks on transitions, and define the acceptance formula. Finally, we will prove the equivalence of $\mathcal{D}$ to $\mathcal{A}$.

We need some more notation here. For a run π of $\mathcal{A}$, by $\mathsf{Rec}_s(\pi)$ we denote the set of states that appear recurrently in the run. For any configuration

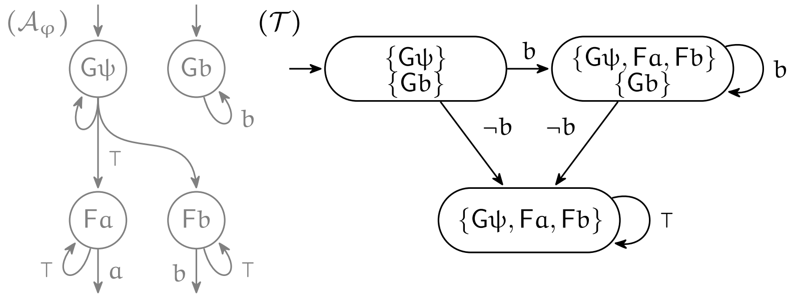


Figure 5.5: The semiautomaton $\mathcal{T}$ (right) for the MMAA $\mathcal{A}_\varphi$ of Figure 5.1. The structure of $\mathcal{A}_\varphi$ is drawn again in grey on the left.

$Z \subseteq S$;[6] by $\mathsf{must}(Z)$ we denote the set of must-states in Z. Finally, we say that the run π is bounded by Z iff $\mathsf{Rec}_s(\pi) \subseteq Z$ and $\mathsf{must}(\mathsf{Rec}_s(\pi)) = \mathsf{must}(Z)$. For example, the run of Figure 5.2 is bounded by $Z = \{G\psi, Fa, Fb\}$.

For every configuration $Z \subseteq$ we define a set $AC_Z \subseteq 2^S$ of *allowed configurations* as follows:

$$AC_Z = \{C \subseteq Z \mid \mathsf{must}(C) = \mathsf{must}(Z)\}$$

For a run to be bounded by Z it is allowed to visit only configurations from AC_Z from some point on.

Further, a set $AT_Z \subseteq \delta$ is a set of *allowed transitions* that contains transitions of $\mathcal{T}$ such that they cover some multitransition to AC_Z. It is defined as follows:

$$AT_Z = \{(m_1, \alpha, m_2) \in \delta \mid \exists C_1 \in AC_Z, C_2 \in (m_2 \cap AC_Z) \text{ and } C_1 \xrightarrow{\alpha} C_2\}$$

A definition of AT_Z with $C_1 \in (m_1 \cap AC_Z)$ might seem more intuitive. It would be correct; however, it is also less effective in practice.

Lemma 5.3. *If $\mathcal{A}$ has a run over u bounded by Z, then the run $\sigma(u)$ of $\mathcal{T}$ over u contains a suffix made of transitions from AT_Z.*

Proof. Let π be a run of $\mathcal{A}$ over u bounded by Z. Then it has a suffix with configurations from AC_Z only. As $\sigma(u)$ tracks all runs of $\mathcal{A}$ over u, it also tracks π and hence has a suffix where for each transition (m_i, u_i, m_{i+1}) there exist configurations $C_1 \in (m_i \cap AC_Z)$ and $C_2 \in (m_{i+1} \cap AC_Z)$ such that $C_1 \xrightarrow{\alpha} C_2$. That implies that $\sigma(u)$ has a suffix containing only transitions from AT_Z. □

$\sigma(u)$ is the unique run of $\mathcal{T}$ over u.

In fact, the other direction can be proved as well: if $\sigma(u)$ contains a suffix of transitions from AT_Z then $\mathcal{A}$ has a run over u bounded by Z.

s-escaping multitransitions. Let $s \in S \cap \mu($🟥$)$ be a marked state of $\mathcal{A}$. We say that a multitransition T is s-*escaping* if it contains a non-looping transition of s. The importance of escaping multitransitions is expressed by the following lemma.

Lemma 5.4. *The run $\pi = T_0T_1\ldots$ of $\mathcal{A}$ over a word u is accepting if and only if for all $s \in S \cap \mu($🟥$) \cap \mathsf{Rec}_s(\pi)$ it holds that π contains infinitely many s-escaping multitransitions.*

Proof. Assume for contradiction that π is accepting and that there is a state $s \in S \cap \mu($🟥$) \cap \mathsf{Rec}_s(\pi)$ such that π contains only finitely many s-escaping multitransitions. Let T_iT_{i+1} be a suffix of π without s-escaping multitransitions such that $s \in \mathsf{source}(T_i)$. As we only removed a finite prefix, s still appears infinitely often in the suffix. Then there is a branch $b = (s, u_i, s)^\omega$ in the suffix which does not satisfy $\mathsf{Fin}($🟥$)$ and thus π cannot be accepting.

[6] We use Z as a name for the configurations here to distinguish them from those we used for the construction of $\mathcal{T}$.

Conversely, all branches that are currently in the state s leave s at every s-escaping multitransition. As $\mathcal{A}$ is an LAA, the branches can never reach the state s again. As π contains infinitely many s-escaping multitransitions for all $s \in \mu(\blacksquare) \cap \mathrm{Rec}_s(\pi)$, no branch can stay in a state marked by $\blacksquare$ and thus π is accepting. □

Clearly, we need to detect runs of $\mathcal{A}$ with bounding configuration Z that contain infinitely many s-escaping multitransitions for each $s \in Z \cap \mu(\blacksquare)$. However, the multitransitions should not leave Z. For each $Z \subseteq S$ and each $s \in Z \cap \mu(\blacksquare)$ we define the set ET_Z^s of s-*escaping transitions* of $\mathcal{T}$ as follows.

$$\mathrm{ET}_Z^s = \{(m_1, \alpha, m_2) \in \delta \mid \exists (s, \alpha, C) \in \Delta \text{ such that } s \notin C \text{ and } C \subseteq Z\}$$

The set ET_Z^s contains transitions such that there exists a non-looping transition of s in $\mathcal{A}$ not leaving Z. Note that all transitions of $\mathcal{T}$ with the same label belong to the set or none of them does.

Now we are ready to build the set of marks M_Z, place the marks on transitions of $\mathcal{T}$ and describe the acceptance formula Φ_Z for each configuration $Z \subseteq S$ in a way that Φ_Z will be satisfied by $\sigma(u)$ if and only if there exists an accepting run of $\mathcal{A}$ over u bounded by Z.

$$M_Z = \{\boxed{Z}\} \cup \{\textcircled{Z}_s \mid s \in Z \cap \mu(\blacksquare)\} \qquad \mu'(\boxed{Z}) = \delta \smallsetminus \mathrm{AT}_Z$$
$$\Phi_Z = \mathrm{Fin}\boxed{Z} \wedge \bigwedge_{s \in Z \cap \mu(\blacksquare)} \mathrm{Inf}\textcircled{Z}_s \qquad \mu'(\textcircled{Z}_s) = \mathrm{AT}_Z \cap \mathrm{ET}_Z^s$$

Satisfying $\mathrm{Fin}\boxed{Z}$ ensures that $\sigma(u)$ has a suffix of transitions allowed for Z and $\mathrm{Inf}\textcircled{Z}_s$ ensures that $\sigma(u)$ has infinitely many s-escaping transitions for Z.

Subsequently, $\sigma(u)$ should be accepting if there exists some $Z \subseteq S$ such that there is an accepting run of $\mathcal{A}$ over u bounded by Z.

$$M = \bigcup_{Z \subseteq S} M_Z \qquad \Phi = \bigvee_{Z \subseteq S} \Phi_Z$$

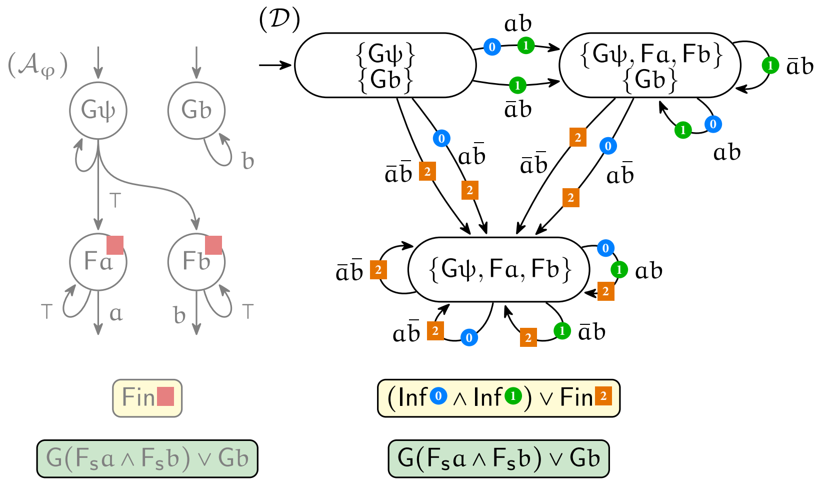

$(\mathcal{D})$
$(\mathcal{A}_\varphi)$
$G\psi$
Gb
Fa
Fb
b
$\top$
a
b
$\top$
$\top$
$\{G\psi\}$ $\{Gb\}$
$\{G\psi, Fa, Fb\}$ $\{Gb\}$
$\{G\psi, Fa, Fb\}$
ab
$\bar{a}b$
$\bar{a}\bar{b}$
$a\bar{b}$
$\mathrm{Fin}\blacksquare$
$(\mathrm{Inf}\,⓪ \wedge \mathrm{Inf}\,①) \vee \mathrm{Fin}\boxed{2}$
$G(F_s a \wedge F_s b) \vee Gb$
$G(F_s a \wedge F_s b) \vee Gb$

Figure 5.6: A deterministic automaton $\mathcal{D}$ (right) equivalent to $\mathcal{A}_\varphi$ (left, in grey). Only the two sets $P = \{G\psi, Fa, Fb\}$ and $R = \{Gb\}$ bound some runs of $\mathcal{A}$.

The mark ⓪ represents $\textcircled{P}_{Fa}$, the mark ① represents $\textcircled{P}_{Fb}$, and finally, $\boxed{2}$ is $\boxed{R}$.

Lemma 5.5. *If there is an accepting run π of $\mathcal{A}$ over u then the run $\sigma(u)$ of $\mathcal{D}$ satisfies Φ_Z for $Z = \mathrm{Rec}_s(\pi)$.*

Proof. From Lemma 5.3 immediately follows that $\sigma(u)$ has a suffix $r_i r_{i+1} \ldots$ of transitions from AT_Z and thus due to the placement of $\boxed{Z}$ marks $\sigma(u)$ satisfies $\mathrm{Fin}\boxed{Z}$.

The run $\pi = T_0 T_1 \ldots$ is accepting, thus by Lemma 5.4, it follows that π has infinitely many s-escaping multitransitions for each $s \in Z \cap \mu(\blacksquare)$. Let s be such state and let T_j for $j > i$ be an s-escaping multitransition of π. Since $j > i$,

The index i comes from the first transition r_i of the suffix from above.

it is clear that the corresponding transition r_j is in ET^s_Z and also in AT_Z, and thus r_j has the mark ⓩ$_s$. As there are infinitely many such indices j, we have that $\sigma(u)$ satisfies Infⓩ$_s$. □

Lemma 5.6. *If a run $\sigma(u)$ of $\mathcal{D}$ satisfies Φ_Z then $\mathcal{A}$ has an* accepting *run over u bounded by Z.*

Proof. If $\sigma(u) = r_0 r_1 \dots$ is a run of $\mathcal{D}$ satisfying Φ_Z, then it has a suffix of transitions of AT_Z and the suffix contains infinitely many transitions of ET^s_Z for each $s \in Z \cap \mu($🟥$)$. Let $r_i = (m_i, u_i, m_{i+1})$ be the first transition of the suffix. From the definition of AT_Z it follows that there is a configuration $C_{i+1} \in (m_{i+1} \cap \mathsf{AC}_Z)$. The construction of $\mathcal{T}$ guarantees that there exists a sequence of multitransitions of $\mathcal{A}$ leading to C_{i+1}. More precisely, for some initial configuration $C_0 \in I$ it holds $C_0 \xrightarrow{u_0} C_1 \xrightarrow{u_1} \dots \xrightarrow{u_{i-1}} C_i \xrightarrow{u_i} C_{i+1}$, and we denote the corresponding sequence of multitransitions by $T_0 T_1 \dots T_i$. This sequence is a prefix of an accepting run of $\mathcal{A}$ over u bounded by Z.

We inductively define a multitransition sequence $T_{i+1} T_{i+2} \dots$ completing this run. The definition relies on the suffix $r_{i+1} r_{i+2} \dots$ of $\sigma(u)$. Let us assume that $j > i$ and that $\mathsf{target}(T_{j-1})$ is a configuration of AC_Z. We define T_j to contain one transition of s for each $s \in \mathsf{target}(T_{j-1})$. Thus we get $\mathsf{source}(T_j) = \mathsf{target}(T_{j-1})$ and the full sequence forms a run. As $r_j \in \mathsf{AT}_Z$, there exists a *reference* multitransition T' labelled by u_j such that both source and target configurations of T' are in AC_Z. We copy from T' to T_j the transitions for all must-states, and for each may-state $s \in \mathsf{target}(T_{j-1})$, we have two cases. If $r_j \in \mathsf{ET}^s_Z$, then T_j contains a non-looping transition leading from s to some states in Z. The existence of such a transition follows from the definition of ET^s_Z. For the remaining may-states, T_j uses the self-loops under u_j. Formally, $T_j = \{t^s_j \mid s \in \mathsf{target}(T_{j-1})\}$, where

$$t^s_j = \begin{cases} (s, u_j, C_s) \text{ contained in } T' & \text{if } s \in \mathsf{must}(Z) \\ (s, u_j, \{s\}) & \text{if } s \in \mu(🟥) \wedge r_j \notin \mathsf{ET}^s_Z \\ (s, u_j, C_s) \text{ where } C_s \subseteq Z, s \notin C_s & \text{if } s \in \mu(🟥) \wedge r_j \in \mathsf{ET}^s_Z \end{cases}$$

One can easily check that $\mathsf{target}(T_j) \in \mathsf{AC}_Z$, and we continue by building T_{j+1}. The run constructed in this way is bounded by Z. Moreover, T_j is s-escaping whenever $r_j \in \mathsf{ET}^s_Z$ which holds infinitely often for each $s \in \mu($🟥$) \cap Z$. The constructed run of $\mathcal{A}$ over u is thus accepting. □

The previous two lemmata prove that the automaton $\mathcal{D}$ accepts the same language as $\mathcal{A}$ and the following Theorem 5.7. In conjunction with Theorem 5.2 we have also proved Theorem 5.8.

Theorem 5.7. *For each MMAA $\mathcal{A}$ with n states, we can construct a deterministic automaton $\mathcal{D}$ with at most 2^{2^n} states and $L(\mathcal{D}) = L(\mathcal{A})$.*

Theorem 5.8. *For each formula $\varphi \in \mathsf{LTL}(\mathsf{F}_s, \mathsf{G}_s)$, we can construct a deterministic automaton $\mathcal{D}_\varphi$ with at most $2^{2^{|\varphi|}}$ states such that $L(\varphi) = L(\mathcal{D}_\varphi)$.*

This upper bound is better than the bounds of all versions of Rabinizer: versions 1 and 2 use automata with marks on states which costs an additional blow-up exponential in the number of atomic propositions; versions 3 and 4 have triple exponential upper bounds for generalized Rabin automata.

5.5 MMAA in the limit and LTL∖G(U,X)

We have just shown a determinization algorithm for MMAA. In fact, our construction works correctly for a larger class of linear alternating automata called *may/must in the limit automata* (limMMAA). An LAA $\mathcal{A}$ is a limMMAA if $\mathcal{A}$ contains only must-states, states without looping transitions, and states marked by 🟥 (not exclusively may-states), and each state reachable from some must-state is either a must- or a may-state. Each accepting run of a limMMAA has a suffix that contains either only empty configurations, or configurations consisting of must-states and may-states reachable from some must-states. Hence, the determinization construction produces correct results also for limMMAA under an additional condition: marks and Φ_Z are constructed only for bounding configurations Z that contain only must-states and may-states reachable from them.

A state s is reachable in $\mathcal{A}$ from a state s_0 iff $\{s_0\} \rightarrow C_1 \rightarrow \ldots \rightarrow C$ such that $s \in C$.

If we translate formulae of the fragment $\mathrm{LTL}\setminus\mathrm{G}(\mathrm{U},\mathrm{X})$ by the translation of LTL2BA, we obtain limMMAA. The translation of $\mathrm{LTL}\setminus\mathrm{G}(\mathrm{U},\mathrm{X})$ into limMMAA places 🟥 marks on all states for subformulae of the form $\psi_1 \,\mathrm{U}\, \psi_2$. The rules for δ for U and X that are needed for the translation and were not given in Section 5.2 follows.

$$\begin{aligned}\delta(\mathrm{X}\psi, \alpha) &= \{\{\psi\}\}\\ \delta(\psi_1 \,\mathrm{U}\, \psi_2, \alpha) &= \overline{\delta}(\psi_2, \alpha) \cup \{C \cup \{\psi_1 \,\mathrm{U}\, \psi_2\} \mid C \in \overline{\delta}(\psi_1, \alpha)\}\end{aligned}$$

$\psi_1 \,\mathrm{U}\, \psi_2 \equiv \psi_2 \vee (\psi_1 \wedge \mathrm{X}(\psi_1 \,\mathrm{U}\, \psi_2))$

5.6 Degeneralization for Rabin automata

Some algorithms that require deterministic automata cannot handle the generalized Rabin acceptance condition and require Rabin automata, often even with marks on states. Generalized Rabin automata have acceptance formula of the form $\bigvee_{k\in K}\left(\mathsf{Fin}\,\boxed{k} \wedge \bigwedge_{j\in J_k}\mathsf{Inf}\,\textcircled{k}^{j}\right)$. In order to get a Rabin automaton, we need to reduce the number of circle marks for each k to one. Our construction is based on a standard degeneralization method for generalized Büchi automata.

We first illustrate the idea on a generalized Rabin automaton $\mathcal{G}$ with K that is a singleton and with h circle marks, that is with acceptance formula Fin🟥 $\wedge \bigwedge_{1\le j\le h}$ Inf🟢^{j}, and we create a Rabin automaton $\mathcal{R}$ with the two marks (placed on states), 🟥 and 🟢, only.

The automaton $\mathcal{R}$ consists of $h+2$ copies of $\mathcal{G}$. The copies are called *levels*. We start at the level 1. Intuitively, being at a level j for $1 \le j \le h$ means that we are waiting for a transition marked by 🟢^{j} in $\mathcal{G}$. Whenever a transition marked by 🟥 appears, we reset and move to the level 0. A transition r without the square mark gets us from level j to the maximal level $l \ge j$ such that $r \in \mu($🟢$^{j'})$ for each $j \le j' < l$. The levels 0 and $h+1$ have the same transitions (including target levels) as the level 1. A run of $\mathcal{G}$ is accepting if and only if the corresponding run of $\mathcal{R}$ visits the level 0 only finitely often and it visits the level $h+1$ infinitely often. Hence all states of level 0 are marked by 🟥 and all states of level $h+1$ are marked by 🟢.

If $r \notin \mu($🟢$^j)$ then there has to be no j' between l and j and therefore, $l = j$.

In the general case, we track the levels for all $k \in K$ simultaneously. Given a DTGRA $\mathcal{G} = (S, \Sigma, \delta, s_I, \bigcup_{k \in K} M_k, \mu, \bigvee_{k \in K} \Phi_k)$ where

$$M_k = \{\boxed{k}\} \cup \{\textcircled{k}^j \mid 1 \leq j \leq h_k\} \text{ and } \Phi_k = \mathsf{Fin}\boxed{k} \wedge \bigwedge_{1 \leq j \leq h_k} \mathsf{Inf}\textcircled{k}^j,$$

we construct an equivalent DRA as $\mathcal{R} = \big(Q, \Sigma, \delta_{\mathcal{R}}, q_I, M, \mu_{\mathcal{R}}, \Phi\big)$, where

- $Q = S \times \{0, 1, \dots, h_1{+}1\} \times \dots \times \{0, 1, \dots, h_{|K|}{+}1\}$,
- $\big((s, l_1, \dots, l_{|K|}), \alpha, (s', l'_1, \dots, l'_{|K|})\big) \in \delta_{\mathcal{R}}$ iff $r = (s, \alpha, s') \in \delta$ and for each $1 \leq k \leq |K|$ it holds

$$l'_k = \begin{cases} 0 & \text{if } r \in \mu(\boxed{k}) \\ \mathsf{level}(r, k, l_k) & \text{if } r \notin \mu(\boxed{k}) \text{ and } 1 \leq l_k \leq h_k \\ \mathsf{level}(r, k, 1) & \text{if } r \notin \mu(\boxed{k}) \text{ and } l_k \in \{0, h_k{+}1\} \end{cases}$$

  where $\mathsf{level}(r, k, i) = \max\left\{l \mid l \leq h_k{+}1 \text{ and } r \in \bigcap_{i \leq j < l} \mu(\textcircled{k}^j)\right\}$,

The intersection of zero sets contains all transitions of $\mathcal{G}$ and thus $1, \dots i$ are always arguments of the maximum and therefore, if $r \notin \mu(\textcircled{k}^i)$ then $\mathsf{level}(r, k, i) = i$.

- $q_I = (m_I, 1, \dots, 1)$,
- $M = \{\boxed{k}, \textcircled{k} \mid k \in K\}$,
- $\mu_{\mathcal{R}}(\boxed{k}) = \{(s, l_1, \dots, l_{|K|}) \in Q \mid l_k = 0\}$,
- $\mu_{\mathcal{R}}(\textcircled{k}) = \{(s, l_1, \dots, l_{|K|}) \in Q \mid l_k = h_k{+}1\}$, and
- $\Phi = \bigvee_{k \in K} (\mathsf{Fin}\boxed{k} \wedge \mathsf{Inf}\textcircled{k})$.

Complexity. We have to multiply the state space of $\mathcal{G}$ by $(h_k + 2)$ for each $k \in K$ in the worst case. Thus $|Q| \leq |S| \cdot (h_1 + 2) \cdot \ldots \cdot (h_{|K|} + 2)$. If we start with an LTL$\setminus$G(U, X) formula φ of length n, we can create an equivalent limMMAA $\mathcal{A}_\varphi$ with n states and generalized Rabin automaton $\mathcal{D}_\varphi$ with at most 2^{2^n} states. To obtain the deterministic Rabin automaton $\mathcal{R}_\varphi$, we multiply the state space of $\mathcal{D}_\varphi$ by at most $|Z| + 2$ for each configuration $Z \subseteq S$ of $\mathcal{A}$, where the number of states in Z is bounded by n. Altogether, we can derive an upper bound on the number of states $|Q|$ of the Rabin automaton as follows.

$$\begin{gathered} |Q| \leq 2^{2^n} \cdot (n+2)^{2^n} = \\ 2^{2^n} \cdot 2^{2^n \cdot \log_2(n+2)} = \\ 2^{2^n} \cdot 2^{2^{n + \log_2 \log_2(n+2)}} \in 2^{\mathcal{O}(2^{n + \log\log n})}. \end{gathered}$$

5.7 IMPLEMENTATION AND TRANSLATION IMPROVEMENTS

We have implemented our translation in a tool called LTL3DRA. The tool is built on top of the LTL to Büchi automata translator LTL3BA[7] and is available at https://github.com/xblahoud/ltl3dra. The two tools share the code for formulae parsing, simplification of LTL formulae, and translation to LAA. The performance of LTL3DRA is evaluated and compared to the performance of other tools in the next chapter. For LTL3DRA to produce a reasonably small automaton we have implemented several optimizations to the core translation, namely we:

- simplify the input formula,
- reduce the state-spaces of automata,
  - remove unreachable states in each step,
  - merge equivalent states in each step,
  - remove redundant transitions of the LAA,
  - reduce macrostates that contain the configuration $\emptyset$, and
  - remove the initial macrostate if found superfluous, and
- simplify the acceptance condition
  - before we compute the placement of the marks (based on the LAA) and
  - after the deterministic automaton is built (based on marks' placement).

Before we describe the optimizations in details, we fix names for the input formula and for the automata and their parts used on the way. The input formula is φ. The corresponding linear alternating automaton is called $\mathcal{A}$ and we set $\mathcal{A} = (S, \Sigma, \Delta, I, \{\blacksquare\}, \mu, \mathsf{Fin}\blacksquare)$. The equivalent deterministic automaton is $\mathcal{D} = (Q, \Sigma, \delta, m_I, M_{\mathcal{Z}}, \mu', \Phi_{\mathcal{Z}})$ where $\mathcal{Z} \subseteq 2^S$ is some set of *bounding configurations*, $M_{\mathcal{Z}} = \bigcup_{Z \in \mathcal{Z}} M_Z$, and $\Phi_{\mathcal{Z}} = \bigvee_{Z \in \mathcal{Z}} \Phi_Z$. Finally, the name for the Rabin automaton is $\mathcal{R}$.

Formula simplifications. On top of the reduction rules of LTL3BA, we add one more – we rewrite subformulae $\mathsf{GF}\psi$ and $\mathsf{FG}\psi$ to equivalent formulae $\mathsf{GF_s}\psi$ and $\mathsf{FG_s}\psi$, respectively. The deterministic automata for formulae with strict temporal operators are often smaller than those without this reduction.[8]

Unreachable states. In each step $(\mathcal{A}, \mathcal{D}, \mathcal{R})$, we always keep and compute only states that are reachable from some initial configuration or from the initial state.

Equivalent states. In each step we iteratively merge equivalent states. Two states of linear automata are equivalent if they have the same transitions and the same marks. Two states of the deterministic automata are equivalent if they have the same marks and for each $\alpha \in \Sigma$ their transitions under α lead to the same state and have the same marks.

[7] Babiak et al. (2012), "LTL to Büchi Automata Translation: Fast and More Deterministic", [33].

[8] In fact, the resulting automata are usually of the same size due to the subsequent state-space reductions. However, the rewriting rule saves the tool from computing many equivalent states only to merge them later. This rewriting rule can be deactivated by the `-X` option.

Redundant transitions. A transition $t_2 = (s, \alpha, C_2) \in \Delta$ is redundant if there is another transition $t_1 = (s, \alpha, C_1) \in \Delta$ of s such that $C_1 \subset C_2$. If we alter an accepting run of $\mathcal{A}$ that uses t_2 to use t_1 instead, it will remain accepting (the change would only remove some branches).

Macrostates with $\emptyset$. If a macrostate m of $\mathcal{D}$ contains the configuration $\emptyset$, we remove all other configurations from m. This modification is clearly correct – if a run π of $\mathcal{A}$ reaches the configuration $\emptyset$ then all subsequent multitransitions of the run are empty and thus π is accepting.[9]

Superfluous initial macrostate. If the initial macrostate m_I of $\mathcal{D}$ does not have any self-loop, we check its equivalence to other states not taking acceptance marks into account. Marks on transitions that are taken at most once by any run are irrelevant.

Bounding configurations. We reduce the number of bounding configurations that we take into account in two ways. First, we consider only configurations that bound some run that we call *modest*. Intuitively, modest runs minimize their sets of recurrent states ($\mathsf{Rec}_S(\pi)$). Formally, a run is modest if it uses for each may-state $s \in S$ only the self-loop of s and exclusively one of its non-looping transitions. For each word $u \in L(\mathcal{A})$ there exists an accepting run that is modest.

Let π be a modest run of $\mathcal{A}$ and let $s \in \mathsf{Rec}_S(\pi)$ be some state visited infinitely often by π. If s is a may-state π can choose the non-looping transition repeatedly. For s being a must-state, however, π does not always have the choice as must-states do not have the self-loop. Therefore, π can be forced by u to use all of its transitions repeatedly. With this in mind, we define a function $\mathsf{mod\text{-}rec}\colon 2^S \to 2^{2^S}$ that recursively computes, for a given configuration Z, the set of configurations that can bound some modest run π that visits the states of Z infinitely often, which is when $Z \subseteq \mathsf{Rec}_S(\pi)$. Before the formal definition of mod-rec we define an auxiliary operation $\otimes$ that when applied to two sets of configurations $W_1, W_2 \subseteq 2^S$ creates a set of combinations of their configurations, formally

$$W_1 \otimes W_2 = \bigcup_{\substack{Z_1 \in W_1 \\ Z_2 \in W_2}} \{Z_1 \cup Z_2\}$$

and an auxiliary function $\mathsf{onestep}\colon S \to 2^{2^S}$ that for a given state s computes the set of configurations that arise by removing s from configurations reachable from s in one step; formally

$$\mathsf{onestep}(s) = \left\{ C \in 2^{S \setminus \{s\}} \mid (s, \alpha, C \cup \{s\}) \in \delta_{\mathcal{A}}, \alpha \in \Sigma \right\}.$$

Finally, the formal definition of mod-rec follows.

[9] In this case, there is no infinite branch in π and therefore all infinite branches satisfy whatever acceptance formula.

$$\mathsf{mod\text{-}rec}(Z) = \begin{cases} \emptyset & \text{if } Z = \emptyset \\ \{\{s\}\} \otimes \bigcup\limits_{\substack{(s,\alpha,C)\in\Delta,\\ s\notin C}} \mathsf{mod\text{-}rec}(C) & \text{if } Z = \{s\} \text{ and } s \notin \mathsf{must}(S) \\ \{\{s\}\} \otimes \bigcup\limits_{W\subseteq \mathsf{onestep}(s)} \mathsf{mod\text{-}rec}(\bigcup\limits_{C\in W} C) & \text{if } Z = \{s\} \text{ and } s \in \mathsf{must}(S) \\ \bigotimes\limits_{s\in C} \mathsf{mod\text{-}rec}(\{s\}) & \text{otherwise} \end{cases}$$

Further, we can eliminate from $\mathcal{Z}$ all configurations that contain a state that is not reachable from some must-state. Indeed, for every accepting run π the set $\mathsf{Rec}_S(\pi)$ contains only states reachable from must-states. Indeed, the other states are left by all branches sooner or later.

In order to find bounding configurations with this property, we define the function $\mathsf{mod\text{-}one}\colon 2^S \to 2^{2^S}$ that recursively computes, for a given configuration Z, the set of configurations that can bound some modest run that ever visits states of Z. As all states reachable from must-states can be visited infinitely often by accepting runs, $\mathsf{mod\text{-}one}(Z) = \mathsf{mod\text{-}rec}(Z)$ for Z that contains such states only.

A run $T_0T_1\ldots$ visits states of Z if for each $s \in Z$ there is some T_i such that $s \in \mathsf{source}(T_i)$.

$$\mathsf{mod\text{-}one}(Z) = \begin{cases} \emptyset & \text{if } Z = \emptyset \\ \bigcup\limits_{\substack{(s,\alpha,C)\in\delta_{\mathcal{A}}\\ s\notin C}} \mathsf{mod\text{-}one}(C) & \text{if } Z = \{s\} \text{ and } s \notin \mathsf{must}(S) \\ \mathsf{mod\text{-}rec}(Z) & \text{if } Z = \{s\} \text{ and } s \in \mathsf{must}(S) \\ \bigotimes\limits_{s\in Z} \mathsf{mod\text{-}one}(\{s\}) & \text{otherwise} \end{cases}$$

Finally, the set $\mathcal{Z}$ of bounding configurations consists of mod-one for all initial configurations of $\mathcal{A}$; that is $\mathcal{Z} = \bigcup_{Z\in I} \mathsf{mod\text{-}one}(Z)$.

Acceptance simplifications. After we place the acceptance marks, we revise the marks and the acceptance formula again. In particular, we perform the following three simplifications.

1. We remove M_Z and $\Phi_Z = \mathsf{Fin}\boxed{Z} \wedge \bigwedge_{j\leq J_Z} \mathsf{Inf}\textcircled{Z}^j$ from M and Φ if no run can satisfy Φ_Z, which is when $\mu'(\boxed{Z}) = \delta$ or if some $\mu'(\textcircled{Z}^j) = \emptyset$.

2. We remove the mark $\textcircled{Z}^{j_1}$ (and the corresponding conjunct in Φ_Z) if there is some $\textcircled{Z}^{j_2}$ such that $\mu'(\textcircled{Z}^{j_2}) \subseteq \mu'(\textcircled{Z}^{j_1})$.

3. If the fact that a run π satisfies Φ_{Z_1} implies that π also satisfies Φ_{Z_2} we remove Φ_{Z_1} and the corresponding M_{Z_1}.

6

LTL to Deterministic Automata Translators: Experimental Evaluation

This chapter evaluates state-of-the-art translators of LTL into deterministic automata in the means of exhaustive experiments. The chapter is inspired by our previous work,[1] but it has been written entirely from scratch. We consider the following three translation approaches (listed in the order of historical appearance).

1. determinization of nondeterministic automata
2. direct translations
3. determinization of cut-deterministic automata

Determinization of Büchi automata. Safra developed the first optimal[2] determinization procedure for Büchi automata in his seminal paper from 1988.[3] His construction takes a Büchi automaton with n states and produces a deterministic Rabin automaton with at most $2^{\mathcal{O}(n \log n)}$ states and at most $2n$ Rabin pairs (which needs $4n$ acceptance marks). Researchers proposed several optimizations since 1988,[4] some of them can take generalized Büchi automata on input, and some of them can even produce parity automata. Parity automata are more desirable for synthesis as solving parity games is more efficient than solving Rabin games.

From the implementation point of view, we have two choices nowadays. For more than ten years, the tool *ltl2dstar*[5] was a synonym for Safra's construction – it is an efficient implementation that includes several optimizations. In 2016, the authors of Spot implemented a determinization based on Redziejowski's construction[6] that takes a Büchi automaton with marks on transitions on input and creates an equivalent deterministic parity automaton on output. The determinization in Spot also implements optimizations based on SCC and on simulation.[7]

Direct translations. The recent boom of direct translations of LTL into deterministic automata was started by Křetínský and Esparza and their construction implemented in *Rabinizer*[8] for the fragment LTL(F,G) in 2012.[9] We have presented the translation of the previous chapter that works for a slightly larger fragment LTL$\setminus\mathsf{G}(\mathsf{U},\mathsf{X})$ in the following year.[10] Our translation is implemented in the tool *LTL3DRA* and it was the first translation that produced generalized Rabin automata with marks on transitions. At the same time, *Rabinizer 2*[11] extended the fragment even more to LTL$\setminus\mathsf{G}(\mathsf{U})$. Finally, in 2014 Esparza and Křetínský finished their effort by providing a translation of the full LTL[12] that was implemented in *Rabinizer 3*[13] and improved in *Rabinizer 4*.[14] All the translations have in common that the output automata have a generalized Rabin acceptance; LTL3DRA, Rabinizer 3 and Rabinizer 4 use marks on transitions.

[1] Blahoudek, Křetínský, and Strejček (2013), "Comparison of LTL to Deterministic Rabin Automata Translators", [14].

[2] singly exponential

[3] Safra (1988), "On the Complexity of Omega-Automata", [50].

[4] Schewe (2009), [4]; Piterman (2007), [51]; Redziejowski (2012), [52].

[5] Klein (2005), "Linear Time Logic and Deterministic ω-Automata", [53]; Klein and Baier (2006), "Experiments with Deterministic Omega-Automata for Formulas of Linear Temporal Logic", [54].

[6] Redziejowski (2012), "An Improved Construction of Deterministic Omega-Automaton Using Derivatives", [52].

[7] Duret-Lutz et al. (2016), "Spot 2.0 - A Framework for LTL and ω-Automata Manipulation", [55].

[8] [57] Gaiser, Křetínský, and Esparza (2012).

[9] Křetínský and Esparza (2012), "Deterministic Automata for the (F, G)-Fragment of LTL", [56].

[10] Babiak et al. (2013), "Effective Translation of LTL to Deterministic Rabin Automata: Beyond the (F, G)-Fragment", [13].

[11] Křetínský and Ledesma-Garza (2013), "Rabinizer 2: Small Deterministic Automata for LTL∖GU", [58].

[12] Esparza and Křetínský (2014), "From LTL to Deterministic Automata: A Safraless Compositional Approach", [59].

[13] Komárková and Křetínský (2014), [60].

[14] Rabinizer 4 was not yet published by the date of submitting the thesis. See Table 6.1 for a reference.

The aforementioned translations that work for some fragment only share a double exponential complexity while the translation of Rabinizer 3 and 4 has a triple exponential upper bound.

Determinization of cut-deterministic Büchi automata. In 2017, Esparza et al. presented a construction that takes a cut-deterministic Büchi automaton and converts it into a deterministic parity automaton with a single exponential blow-up. The construction is based on coloring of runs.[15] The construction can be improved if it is chained together with a translation of LTL into cut-deterministic automata by the same authors.[16] The result of these two constructions chained together is a double exponential translation from full LTL into DPA. This approach was implemented in the tool *ltl2dpa*. Ltl2dpa has initially been a part of the owl library,[17] and now it is also distributed as a part of the yet unpublished Rabinizer 4.

To reproduce the evaluation (or to run it with new versions of the tools) visit https://github.com/xblahoud/LTL2DA-comparison. You can find a collection of scripts, files with the used LTL formulae,[18] and Jupyter notebooks that can repeat all computations performed for this chapter on this page. The notebooks also generate all tables and figures used here and they also include some additional data. In the interactive Jupyter notebooks you can explore the data and look for information of your interest if you miss it here.

[15] Esparza et al. (2017), "From LTL and Limit-Deterministic Büchi Automata to Deterministic Parity Automata", [61].

[16] Sickert et al. (2016), "Limit-Deterministic Büchi Automata for Linear Temporal Logic", [62].

[17] available at https://www7.in.tum.de/~sickert/projects/owl/

[18] including the scripts used to generate them

6.1 EVALUATED TOOLS

Altogether, we evaluate 22 tool chains that convert LTL formulae into deterministic automata. In particular, we have 14 tool chains that rely on Safra-like determinization, 6 variants of tools for direct translations, and finally 2 translations performed by ltl2dpa. Figure 6.1 gives an overview of most of the used tool chains. Homepages and versions of all tools used for this evaluation are listed in Table 6.1.

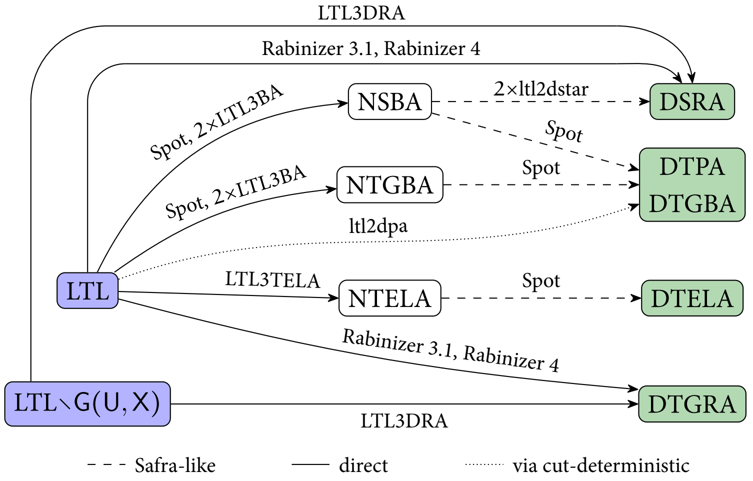


Figure 6.1: Evaluated tool chains for translation of LTL to deterministic automata. The blue boxes are LTL fragments, the white boxes represent types of intermediate automata, the green boxes represent the type of output automata (two types indicate that the tool chain produces one type for some formulae and the other for the rest, the paragraph on Safra-based translations on the next page explains these cases), and finally, the type of line denotes the type of translation.

Naming. We reference each tool chain by a triple *(main, intermediate, acc)*. We use *main* for the tool that outputs the final deterministic automaton, *intermediate* indicates which LTL to nondeterministic automata translator was used in case of the Safra's approach or which mode of ltl2dpa was used, and finally *acc* provides details about acceptance conditions of the resulting automata. For the Safra-based tool chains, *acc* consists of two parts divided by a dot: the first part is the acceptance of the intermediate nondeterministic automaton (SB, TGB, TEL) and the second part is the acceptance condition of the output automaton (SR, TP, TEL).[19]

We use the tool ltlcross from the Spot library to run all the tool chains and gather the information of interest about the resulting automata. You can find the exact ltlcross command and the reference name for each tool chain in Table 6.2. In the following text, we comment on the choice of tool chains, provide more details about some, and comment explicitly on the two tool chains (Spot, —, TP) and (ltl2dpa, Rabinizer, TP) that cannot be found in Figure 6.1.

Direct translations. We have chosen three tools that translate LTL formulae directly into deterministic automata: *LTL3DRA*, *Rabinizer 3.1*, and, with the kind permission of its authors, *Rabinizer 4*. We have excluded *Rabinizer* and *Rabinizer 2* for the following reasons; the tools work for fragments only, they do not support HOA format, they are no longer maintained, they contain many bugs, and they give larger automata than their successor Rabinizer 3.1 in most cases. We have run two variants of each of these tools: one that outputs DTGRA and one that outputs DSRA (not shown in Figure 6.1) which we have included mainly to provide some output comparable to the output of ltl2dstar.

[19] We denote the cases where the resulting automaton can be either a DTPA or a DTGBA by *TP* as the parity acceptance is in some sense more complex than generalized Büchi. TGBA cannot be converted into TPA without changing the structure of the automaton; however, neither emptiness check nor game solving is harder for TGBA than for TPA.

Safra-based translations. We evaluate both ltl2dstar and Spot for determinization of nondeterministic automata. The tool ltl2dstar offers two input interfaces: *ltl2dstar (NBA)* reads a Büchi automaton directly, and *ltl2dstar (LTL)* reads an LTL formula together with instructions how to call some LTL to NBA translator. The knowledge of the original LTL formula can enable some optimizations (for example for stutter-invariant properties).[20]

While ltl2dstar determinizes Büchi automata with marks on states, autfilt– the determinization tool of Spot – can also process automata with marks on transitions. Moreover, Spot can convert arbitrary automaton into a TBA internally and thus it can input TGBA (all considered LTL-to-BA translators can output TGBA) or even TELA directly. If the input automaton is deterministic, autfilt only simplifies it and we get a DTGBA or a DTELA on output, in other cases, Spot converts the automata into DTPA. You can see the workflow of Spot in Figure 6.2. Besides autfilt (denoted as *Spot (autfilt)*), Spot offers another way to get a deterministic automaton that starts with an LTL formula directly. This approach (referenced as *(Spot, —, TP)*) uses the same algorithms as *(Spot (autfilt), Spot, TGB.TP)*, however, it may produce different automata in some cases.[21]

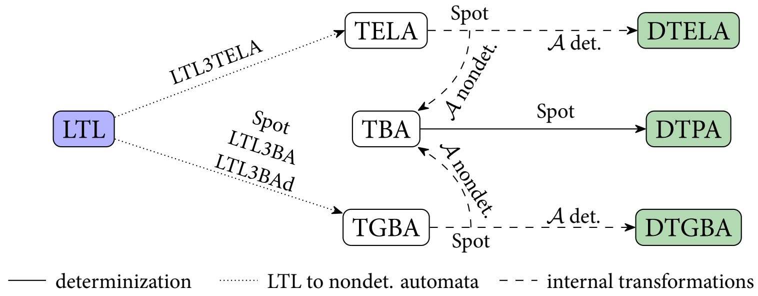


Figure 6.2: Workflow of Spot for determinization of automata that do not have Büchi acceptance condition with marks on states.

LTL to nondeterministic automata. Our previous evaluation[22] suggests using only Spot and LTL3BA for translation of LTL into BA for ltl2dstar. We again run LTL3BA in 2 configurations: one prefers to output automata as small as possible (*LTL3BA*) while the other has a preference to create potentially bigger but more deterministic automata (*LTL3BAd*). Spot offers a similar choice between smaller and more deterministic automata, however, the previous evaluation revealed that this choice makes only a negligible difference in the size of the deterministic automata. As autfilt can take arbitrary automaton on input, we also consider the tool chain[23] that employs the tool LTL3TELA which translates LTL into TELA.

Other (ltl2dpa). We use the version of ltl2dpa from the Rabinizer 4 tool set. It offers two modes of conversion of LTL formulae into deterministic parity automata. The modes differ in the core translation used in the first step. The default option *(ltl2dpa, ltl2ldba TP)* uses an LTL to cDGBA translation of the tool *ltl2ldba*.[24] If the cut-deterministic automaton (TGBA) is already deterministic, ltl2dpa outputs it directly; otherwise it uses a construction based on runs' coloring to produce a DTPA. The second option *(ltl2dpa, Rabinizer, TP)* relies on the LTL to Rabin automata translation of Rabinizer 4 to build an intermediate Rabin automaton with marks on transitions that is converted into a parity automaton by a construction based on improved index appearance record.[25]

[20] Klein and Baier (2007), "On-the-Fly Stuttering in the Construction of Deterministic Omega-Automata", [63].

[21] (i) The knowledge of the input formula allows Spot to treat obligation properties more efficiently and (ii) the acceptance marks that are outside of SCC of the intermediate TGBA are not removed (which is the default for non-deterministic automata in Spot) in this case as they are often beneficial to the determinization algorithm of Spot; however, sometimes they cause that the resulting automaton is larger.

[22] Blahoudek, Křetínský, and Strejček (2013), "Comparison of LTL to Deterministic Rabin Automata Translators", [14].

[23] (Spot (autfilt), LTL3TELA, TEL.TEL)

[24] Sickert et al. (2016), "Limit-Deterministic Büchi Automata for Linear Temporal Logic", [62].

[25] Křetínský et al. (2017), "Index Appearance Record for Transforming Rabin Automata into Parity Automata", [64].

tool	version	webpage
LTL3BA	1.1.3	https://sourceforge.net/p/ltl3ba/
LTL3TELA	1.1.1	https://github.com/jurajmajor/ltl3tela
Spot(ltl2tgba)	2.5	https://spot.lrde.epita.fr/
ltl2dstar	0.5.4	http://ltl2dstar.de/
Spot(autfilt)	2.5	https://spot.lrde.epita.fr/
LTL3DRA	0.2.6	https://github.com/xblahoud/ltl3dra
Rabinizer 3	3.1	https://www7.in.tum.de/~kretinsk/rabinizer3.html
Rabinizer 4	15. 2. 2018	https://www7.in.tum.de/~kretinsk/rabinizer4.html
ltl2dpa	15. 2. 2018	https://www7.in.tum.de/~kretinsk/rabinizer4.html

Table 6.1: References for tools used in the experimental evaluation of LTL to deterministic automata translators. The first part contains tools that convert LTL formulae into nondeterministic automata, the second part lists tools that can determinize these nondeterministic automata, the third part shows tools for direct translations, and finally, the last part gives a reference for ltl2dpa. For Rabinizer 4 and ltl2dpa we give the date of download as they do not use any minor version numbers. Note that we also use ltl2tgba to convert LTL to deterministic automata.

type	name	interm.	acc	ltlcross command
direct	LTL3DRA	—	TGR	`ltl3dra -f %s > %O`
	LTL3DRA	—	SR	`ltl3dra -H3 -f %s > %O`

	Rabinizer 3	—	TGR	`java -jar Rab3/rabinizer3.1.jar -silent \`
				`-format=hoa -out=std %[eiRWM]f > %O`

	Rabinizer 3	—	SR	`java -jar Rab3/rabinizer3.1.jar -silent \`
				`-format=hoa -out=std -auto=sr %[eiRWM]f > %O`

	Rabinizer 4	—	TGR	`Rab4/bin/ltl2dgra %f > %O`
	Rabinizer 4	—	SR	`Rab4/bin/ltl2dra %f | autfilt --sbacc > %O`
Safra	ltl2dstar (LTL)	LTL3BA	SB.SR	`ltl2dstar -H -t "ltl3ba -M0 -H3 -f %%s > %%H" %L %O`
	ltl2dstar (LTL)	LTL3BAd	SB.SR	`ltl2dstar -H -t "ltl3ba -M1 -H3 -f %%s > %%H" %L %O`
	ltl2dstar (LTL)	Spot	SB.SR	`ltl2dstar -H -t "ltl2tgba -B -f %%s > %%H" %L %O`
	ltl2dstar (NBA)	LTL3BA	SB.SR	`ltl3ba -M0 -H3 -f %s | ltl2dstar -B -H - - > %O`
	ltl2dstar (NBA)	LTL3BAd	SB.SR	`ltl3ba -M1 -H3 -f %s | ltl2dstar -B -H - - > %O`
	ltl2dstar (NBA)	Spot	SB.SR	`ltl2tgba -B -f %f | ltl2dstar -B -H - - > %O`
	Spot (autfilt)	LTL3BA	TGB.TP	`ltl3ba -M0 -H2 -f %s | autfilt -DG > %O`
	Spot (autfilt)	LTL3BA	SB.TP	`ltl3ba -M0 -H3 -f %s | autfilt -DG > %O`
	Spot (autfilt)	LTL3BAd	TGB.TP	`ltl3ba -M1 -H2 -f %s | autfilt -DG > %O`
	Spot (autfilt)	LTL3BAd	SB.TP	`ltl3ba -M1 -H3 -f %s | autfilt -DG > %O`
	Spot (autfilt)	LTL3TELA	TEL.TEL	`ltl3tela -f %f | autfilt -DG > %O`
	Spot (autfilt)	Spot	TGB.TP	`ltl2tgba -f %f | autfilt -DG > %O`
	Spot (autfilt)	Spot	SB.TP	`ltl2tgba -B -f %f | autfilt -DG > %O`
	Spot	—	TP	`ltl2tgba -DG -f %f > %O`
other	ltl2dpa	ltl2ldba	TP	`Rab4/bin/ltl2dpa --mode=ldba %f > %O`
	ltl2dpa	Rabinizer	TP	`Rab4/bin/ltl2dpa --mode=rabinizer %f > %O`

Table 6.2: All considered tool chains with the corresponding commands passed to ltlcross. The tool chains are divided into three parts regarding the type of translation they rely on.

Discovered bugs. During preparation of this chapter, I have found several confirmed bugs in some of the considered tools. Namely, 5 bugs in (the preliminary versions) of Rabinizer 4 and ltl2dpa, 5 bugs in the Spot library, 3 bugs in Rabinizer 3.1, one segmentation fault bug in LTL3TELA, and one bug in ltl2dstar. Authors of the touched tools (except Rabinizer 3.1) have already fixed most of the bugs and released new versions of the tools. I am grateful for their prompt response (even during the Christmas holidays).

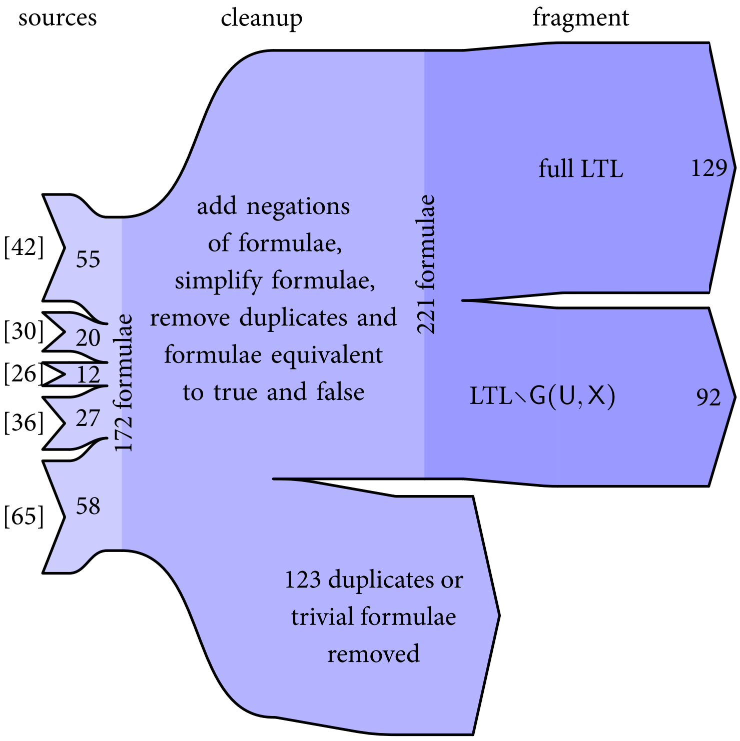

sources
cleanup
fragment
[42]
55
[30]
20
[26]
12
[36]
27
[65]
58
172 formulae
add negations of formulae, simplify formulae, remove duplicates and formulae equivalent to true and false
221 formulae
full LTL
129
LTL∖G(U, X)
92
123 duplicates or trivial formulae removed

Figure 6.3: Preparation of the formulae from the literature, and classification according to fragments of LTL.

6.2 BENCHMARK FORMULAE

We use benchmark formulae from two sources – formulae from literature and randomly generated formulae. We consider both *concrete* and *parametric* formulae that were already used for benchmarking LTL translators in the literature. We further divide the formulae based on LTL fragments. We use the word *benchmark* to denote each of the considered sets of LTL formulae.

Concrete formulae from literature. We have collected many formulae that were already used in the literature[26] to evaluate the performance of LTL translators. For each formula from the listed sources, we added its negation into our set, simplified all the formulae by ltlfilt,[27] removed duplicates and formulae equivalent to true or false. The resulting collection contains 221 formulae. As LTL3DRA works correctly for a fragment of LTL only, we have further separated the set of formulae based on their presence to $\mathrm{LTL}\smallsetminus\mathsf{G}(\mathsf{U},\mathsf{X})$. The Figure 6.3 shows that there are 92 formulae from the fragment $\mathrm{LTL}\smallsetminus\mathsf{G}(\mathsf{U},\mathsf{X})$ and 129 formulae outside $\mathrm{LTL}\smallsetminus\mathsf{G}(\mathsf{U},\mathsf{X})$ (referenced as *full LTL*). There are 42 formulae that are both in $\mathrm{LTL}\smallsetminus\mathsf{G}(\mathsf{U},\mathsf{X})$ and LTL(F,G). We have not separated the benchmark of $\mathrm{LTL}\smallsetminus\mathsf{G}(\mathsf{U},\mathsf{X})$ into two due to its small size.

Random formulae. We have used the tool randltl from the Spot library to generate 500 formulae from each relevant fragment of LTL randomly. We consider three fragments: *full LTL*, $\mathrm{LTL}\smallsetminus\mathsf{G}(\mathsf{U},\mathsf{X})$, and LTL($\mathsf{F}$,$\mathsf{G}$). The fragment LTL($\mathsf{F}$,$\mathsf{G}$) was included because the direct translations can deal with the operators F and G much easier than with other temporal operators. The sets for the full LTL and for $\mathrm{LTL}\smallsetminus\mathsf{G}(\mathsf{U},\mathsf{X})$ are disjoint, while $\mathrm{LTL}\smallsetminus\mathsf{G}(\mathsf{U},\mathsf{X})$ shares 50 formulae with LTL(F,G). Further, 7 formulae from the full LTL and 14 formulae from $\mathrm{LTL}\smallsetminus\mathsf{G}(\mathsf{U},\mathsf{X})$ are both in the random and in the literature benchmark.

Table 6.3 gives the total number of formulae in each benchmark. The *mixed* source shows the number of distinct formulae after the corresponding literature and random benchmarks were merged together. The second column of the table shows the number of formulae for which all tool chains were able to produce a correct deterministic automaton within given constraints. See the next section for the constraints and for the reasons that prevented the tool chains from creating the desired automata.

Table 6.3: Concrete formulae benchmarks.

source	fragment	total count	all finished
literature	full LTL	129	122
	$\mathrm{LTL}\smallsetminus\mathsf{G}(\mathsf{U},\mathsf{X})$	92	89
random	full LTL	500	479
	$\mathrm{LTL}\smallsetminus\mathsf{G}(\mathsf{U},\mathsf{X})$	500	494
	LTL(F,G)	500	500
mixed	full LTL	622	594
	$\mathrm{LTL}\smallsetminus\mathsf{G}(\mathsf{U},\mathsf{X})$	578	569

[26] Etessami and Holzmann (2000), [26]; Pelánek (2007), [30]; Somenzi and Bloem (2000), [36]; Dwyer, Avrunin, and Corbett (1998), [42]; Holeček et al. (2004), [65].

[27] Duret-Lutz (2013), “Manipulating LTL Formulas Using Spot 1.0”, [41].

Parametric formulae from literature. Besides concrete formulae, we evaluate the tool chains on parametric formulae. We have used formulae that were already used to benchmark LTL to automata translators by Gastin and Oddoux,[28] Geldenhuys and Hansen,[29] Müller and Sickert,[30] formulae that witness the double exponential blow-up of the LTL to deterministic automata translation described by Kupferman and Rosenberg,[31] and finally two formulae from LTL(F,G).

We have used the tool genltl from the Spot library to generate all instances of the parametric formulae, and we adopt the naming of the formulae used by this tool. For the formulae used already in the literature, the first part of the name is a lowercase acronym of the authors' names (for example *go* for Gastin and Oddoux). The two additional formulae from LTL(F,G) are called *and-fg* and *or-fg*. For the witness formulae of the double exponential blow-up (*kr-n* and *kr-nlogn*), please consult the original paper. The list of the other formulae follows.

$$
\begin{array}{rlrl}
\text{gh-e}(n) = & \bigwedge_{i=1}^{n} \mathsf{F} a_i & \text{gh-q}(n) = & \bigwedge_{i=1}^{n} (\mathsf{F} a_i \vee \mathsf{G} a_{i+1}) \\
\text{gh-s}(n) = & \bigvee_{i=1}^{n} \mathsf{G} a_i & \text{gh-r}(n) = & \bigwedge_{i=1}^{n} (\mathsf{GF} a_i \vee \mathsf{FG} a_{i+1}) \\
\text{gh-c1}(n) = & \bigvee_{i=1}^{n} \mathsf{GF} a_i & \text{gh-u}(n) = & (\ldots((a_1 \,\mathsf{U}\, a_2) \,\mathsf{U}\, a_3) \,\mathsf{U} \ldots) \,\mathsf{U}\, a_n \\
\text{gh-c2}(n) = & \bigwedge_{i=1}^{n} \mathsf{GF} a_i & \text{gh-u2}(n) = & a_1 \,\mathsf{U}(a_2 \,\mathsf{U}(\ldots(a_{n-1} \,\mathsf{U}\, p_n)\ldots)) \\
\text{and-fg}(n) = & \bigwedge_{i=1}^{n} \mathsf{FG} a_i & \text{ms-phi-h}(n) = & \bigvee_{i=0}^{n} (\mathsf{FG}(\neg^i a \vee \mathsf{X}^i b)) \\
\text{or-fg}(n) = & \bigvee_{i=1}^{n} \mathsf{FG} a_i & \text{go-theta}(n) = & \neg\left(\left(\bigwedge_{i=1}^{n} \mathsf{GF} a_i\right) \rightarrow \mathsf{G}(b \rightarrow \mathsf{F} c)\right) \\
\text{ms-phi-r}(0) = & \mathsf{FG} a_0 \wedge \mathsf{GF} b_0 & \text{ms-phi-r}(i+1) = & \mathsf{FG} a_{i+1} \wedge \mathsf{GF} b_{i+1} \vee \text{ms-phi-s}(i) \\
\text{ms-phi-s}(0) = & \mathsf{FG} a_0 \vee \mathsf{GF} b_0 & \text{ms-phi-s}(i+1) = & \mathsf{FG} a_{i+1} \vee \mathsf{GF} b_{i+1} \wedge \text{ms-phi-r}(i)
\end{array}
$$

[28] Gastin and Oddoux (2001), "Fast LTL to Büchi Automata Translation", [31].

[29] Geldenhuys and Hansen (2006), "Larger Automata and Less Work for LTL Model Checking", [66].

[30] Müller and Sickert (2017), "LTL to Deterministic Emerson-Lei Automata", [67].

[31] Kupferman and Rosenberg (2010), "The Blowup in Translating LTL to Deterministic Automata", [68].

6.3 HARDWARE, BENCHMARK SETTINGS, AND ERRORS

All experiments ran on a workstation with the Intel i7-3770 3.40GHz CPU and with 8GB DDR3 1333MHz RAM. We set up a time limit of 120 seconds for non-parametric and 300 seconds for parametric benchmarks. We did not set any explicit memory limit. We rely on ltlcross from the Spot library to run the translations. Spot can only process automata that use up to 32 different acceptance marks and thus we had to remove all automata violating the limit of 32 marks from our results. This limitation affected mostly automata produced by Rabinizer 3.

Tables 6.4 and 6.5 give the summary of automata that we were not able to analyze or were flawed and thus we removed them from the final results. The unique case where some tool crashed was again due to the Spot's limitation on the number of marks – as LTL3TELA relies internally on Spot, in case it needs more than 32 marks it exits and produces no automaton. The total numbers of formulae for which some tool failed to produce a correct automaton are 7 and 3 for the literature benchmarks and 21 and 6 for the random benchmarks, respecting the order of fragments from the tables; the LTL(F,G) benchmark of random formulae contains no error (see Table 6.3).

Table 6.4: Errors summary for formulae from literature.

			full LTL		LTL$\smallsetminus$G(U, X)	
main	interm.	acc	>32 marks	timeout	>32 marks	timeout
LTL3DRA	—	TGR	—	—	1	—
Rabinizer 3	—	SR	6	—	1	1
Rabinizer 3	—	TGR	—	—	—	1
Spot (autfilt)	LTL3TELA	TEL.TEL	—	1	—	1

Table 6.5: Errors summary for random formulae.

			full LTL		LTL$\smallsetminus$G(U, X)	
main	interm.	acc	>32 marks	crash	incorrect	>32 marks
Rabinizer 3	—	SR	21	—	1	5
Rabinizer 3	—	TGR	6	—	1	1
Spot (autfilt)	LTL3TELA	TEL.TEL	—	1	—	—

The only tool whose bugs remained unfixed until submission of the thesis is Rabinizer 3. The tool sometimes outputs an automaton with an empty acceptance condition (equivalent to false). This causes the two 1 in the column for incorrect automata. The incorrect automata were removed from further analysis.

6.4 RESULTS: NON-PARAMETRIC BENCHMARKS

We first describe how we present the measured data and then we offer some observations based on the data. We use mainly the following three types of data visualization. Additionally, we use scatter and quantile plots to investigate some phenomena more deeply.

Numbers of minimal automata. Figure 6.4 show for each benchmark and each approach (*direct*, *Safra*-like, and *ltl2dpa*) the number of formulae from the benchmark for which some tool chain of the given approach can produce automaton with the minimal number of states from all considered tool chains. Figures 6.6 (*literature* benchmarks) and 6.7 (*random* benchmarks) refine the information from Figure 6.4 to the level of tool chains. There are two plots on each of these figures. The right-hand side presents the numbers after all automata were converted to use marks on states exclusively (using autfilt).

Cumulative numbers. Tables 6.6 (literature) and 6.7 (random) present cumulative results of the evaluated tool chains on the non-parametric benchmarks. For each benchmark and each tool chain, the tables show the cumulative number of states and acceptance marks[32] of the produced automata. The third column sums the time needed to compute all automata from the benchmark by the corresponding tool chain. The green color highlights the best (minimal) values for each column. The thick lines divide translation types.[33] We include only formulae where all tools finished their computation without any error in this comparison;[34] the number in brackets following a fragment name shows the number of such formulae for the benchmark. Note that these tables mix automata with different acceptance conditions.

Cross-comparison. Cross-comparison tables compare tool chains within some logical groups in more detail. More precisely, we compare the tool chains against each other on individual formulae and count the number of *victories* on formulae from each benchmark. Let us consider two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ produced for a formula φ by the tool chains t_1 and t_2, respectively. We say that t_1 *wins* against t_2 on φ if

1. t_2 violates some of the given limits (time, >32 acceptance marks, incorrect automaton) and t_1 succeeds to build a correct $\mathcal{A}_1$, or
2. $\mathcal{A}_1$ has less states than $\mathcal{A}_2$, or
3. if the numbers of states of $\mathcal{A}_1$ and $\mathcal{A}_2$ agree and $\mathcal{A}_1$ uses fewer acceptance marks than $\mathcal{A}_2$.

The tables show the number of victories for all pairs of tool chains from the corresponding groups. We assign a number to each tool chain in the first column (#), this number is used to reference the tool chain in the columns header. A cell on the row r in column c contains the number of victories of the tool chain r over c. Finally, the last column of the tables sums the number of victories for the tool chain in each row. We use brown boxes around fragments name to distinguish the random benchmarks (box) from the literature benchmarks (no box).We present the cross-comparison for tools that perform direct translation (Table 6.8), tool chains of ltl2dstar (Table 6.9), tool chains of Spot (Table 6.10), and finally for ltl2dpa together with the tools Spot and Rabinizer 4 selected based on the previous results (Table 6.11).

[32] Here we consider the number of distinct marks (the size of the set M), not the count of marks placed on individual states and transitions.

[33] direct, Safra-like, and ltl2dpa

[34] See Tables 6.4 and 6.5 for reasons for formula exclusions.

Observations based on the numbers of minimal automata. The numbers of minimal automata by approaches indicate that the determinization-based approach (denoted as *Safra*) is best suited for formulae outside $\text{LTL}\setminus\mathsf{G}(\mathsf{U},\mathsf{X})$ (full LTL) while the direct translations excel on the LTL(F,G) fragment. However, Figure 6.5 shows that none of the approaches is truly dominant to the others.[35]

If we look at the level of tools, the first observation is no longer valid for random formulae outside $\text{LTL}\setminus\mathsf{G}(\mathsf{U},\mathsf{X})$, where Rabinizer 4 wins with its direct approach. However, for the formulae from literature, Rabinizer 4 cannot compete with Spot. Overall, Rabinizer 4 seems to be the most successful tool that performs a direct translation and (Spot, —, TP) is the best Safra-based tool chain.

It is particularly interesting to look at the numbers of minimal automata with marks on states for the fragment LTL(F,G) in Figure 6.7. (i) It is the only benchmark where ltl2dstar seems to be the better option as the determinization tool than Spot (please keep in mind that it is no longer the case if you do not request marks on states). (ii) It is the only benchmark where (ltl2dpa, ltl2ldba TP) is the tool chain that hits the minimum size most often, and the lead is outstanding (96 cases).

Pushing marks from transitions to states was very harmful to all tools using the generalized Rabin condition[36] and for Spot with LTL3TELA while it helped ltl2dpa at the same time. The algorithm that moves the marks creates, for each state s and each combination of marks that can be found on transitions leading to s, a unique state. Parity automata have at most one mark on each edge while the TGRA usually place marks on edges in numerous combinations – each such the combination leads to a unique state after marks are pushed to states.

On the random benchmarks with restricted fragments Spot is most successful with LTL3TELA. We suspect that many of these cases are when the TELA produced by LTL3TELA is already deterministic. However, if we consult Table 6.7 we cannot confirm the dominance of this tool chain over other tool chains that use Spot for determinization in cumulative sizes of automata.

Observations based on the cumulative numbers. The cumulative numbers reveal that Rabinizer 4 produces (on average) the smallest automata for the restricted fragments. On the fragment LTL(F,G), we can observe that the determinization-based tools are a bad choice. On the other hand, the numbers are in favour of Spot for the formulae outside $\text{LTL}\setminus\mathsf{G}(\mathsf{U},\mathsf{X})$.

Please note that the high numbers of states for ltl2dstar can be influenced by the fact that it produces automata with marks of states while the other tool chains usually place marks on transitions. We discuss ltl2dstar in more details later on and we address this issue when we compare ltl2dstar and autfilt.

Spot is the right choice for situations where a simple acceptance condition is desirable – all tool chains that involve some other tool than Spot need far more acceptance marks than Spot alone for each of the five benchmarks.

Finally, Rabinizer 3, Rabinizer 4, ltl2dpa, and partially LTL3TELA do exhibit long computation times. However, the computation time of the automaton is usually the minor problem in comparison to the analysis of the product with some system that follows in many cases.

[35] It is often the case that at least two approaches achieve the same result.

[36] LTL3DRA, Rabinizer 3, and Rabinizer 4

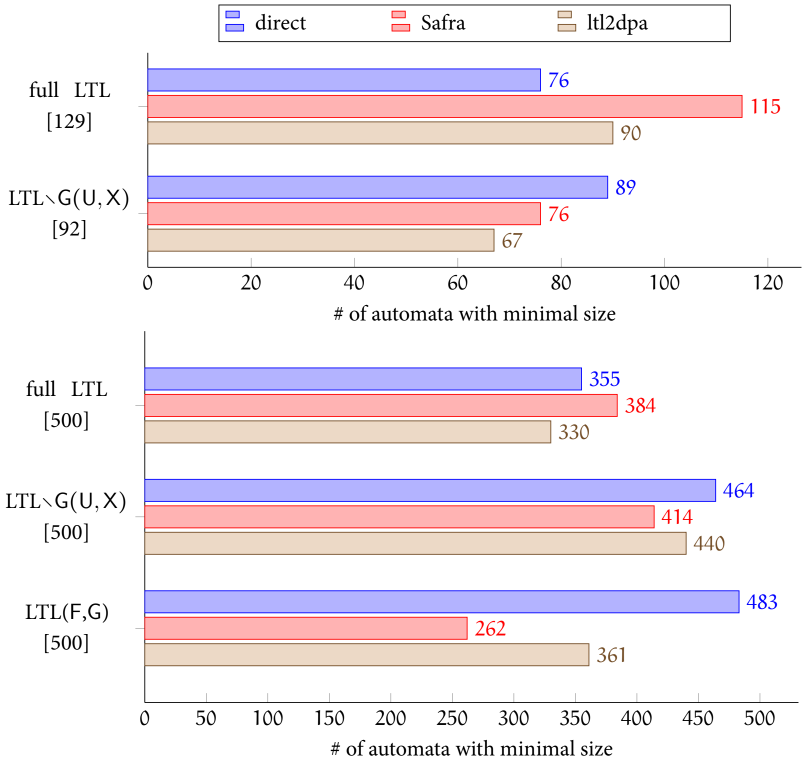


Figure 6.4: Numbers of minimal automata by approaches for the *literature* (top) and the *random* (bottom) benchmarks. For each benchmark and each approach we show in how many cases some tool that uses the considered approach produced an automaton with the minimal number of states that was achieved for a given formula. The numbers in brackets below the fragment names show the total number of formulae in the benchmark.

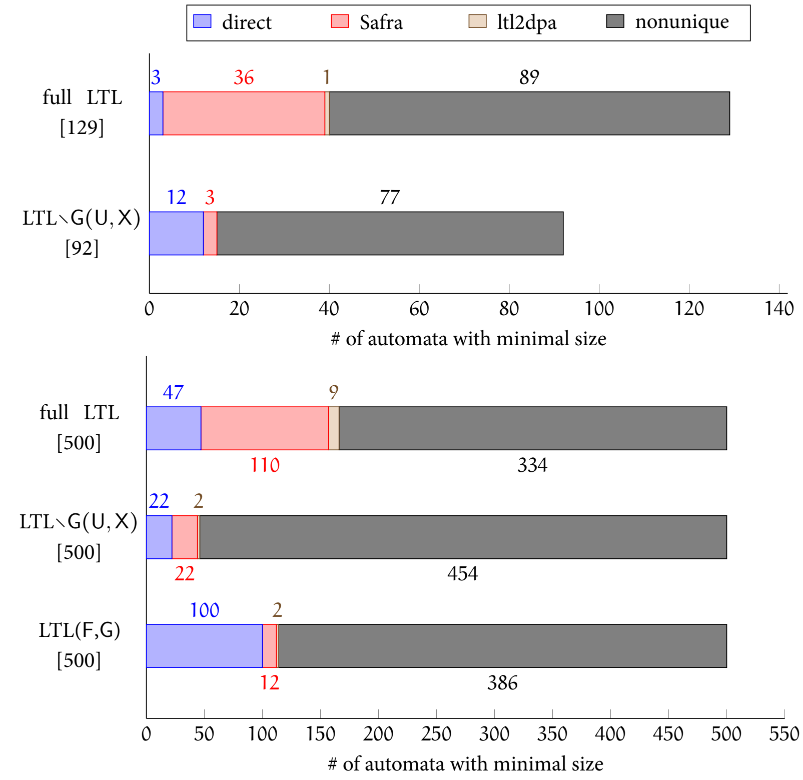


Figure 6.5: Numbers of *unique* minimal automata by approaches for the *literature* (top) and the *random* (bottom) benchmarks. We consider only cases where no other approach reached the same size of automata; the number of cases where at least two approaches managed to produce some minimal automaton is shown by the *nonunique* (grey) bar.

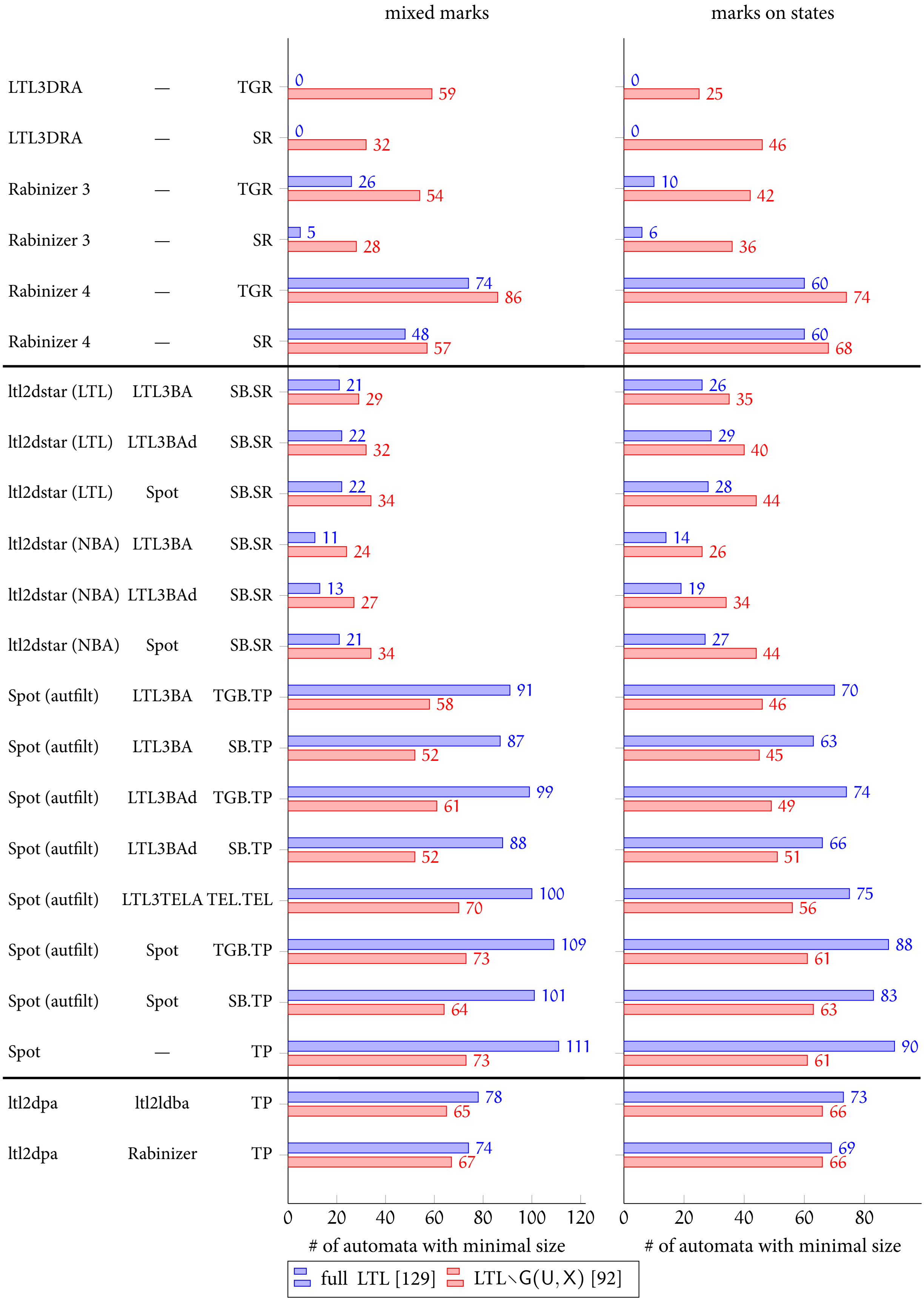


Figure 6.6: The numbers of minimal automata for *literature* benchmarks.

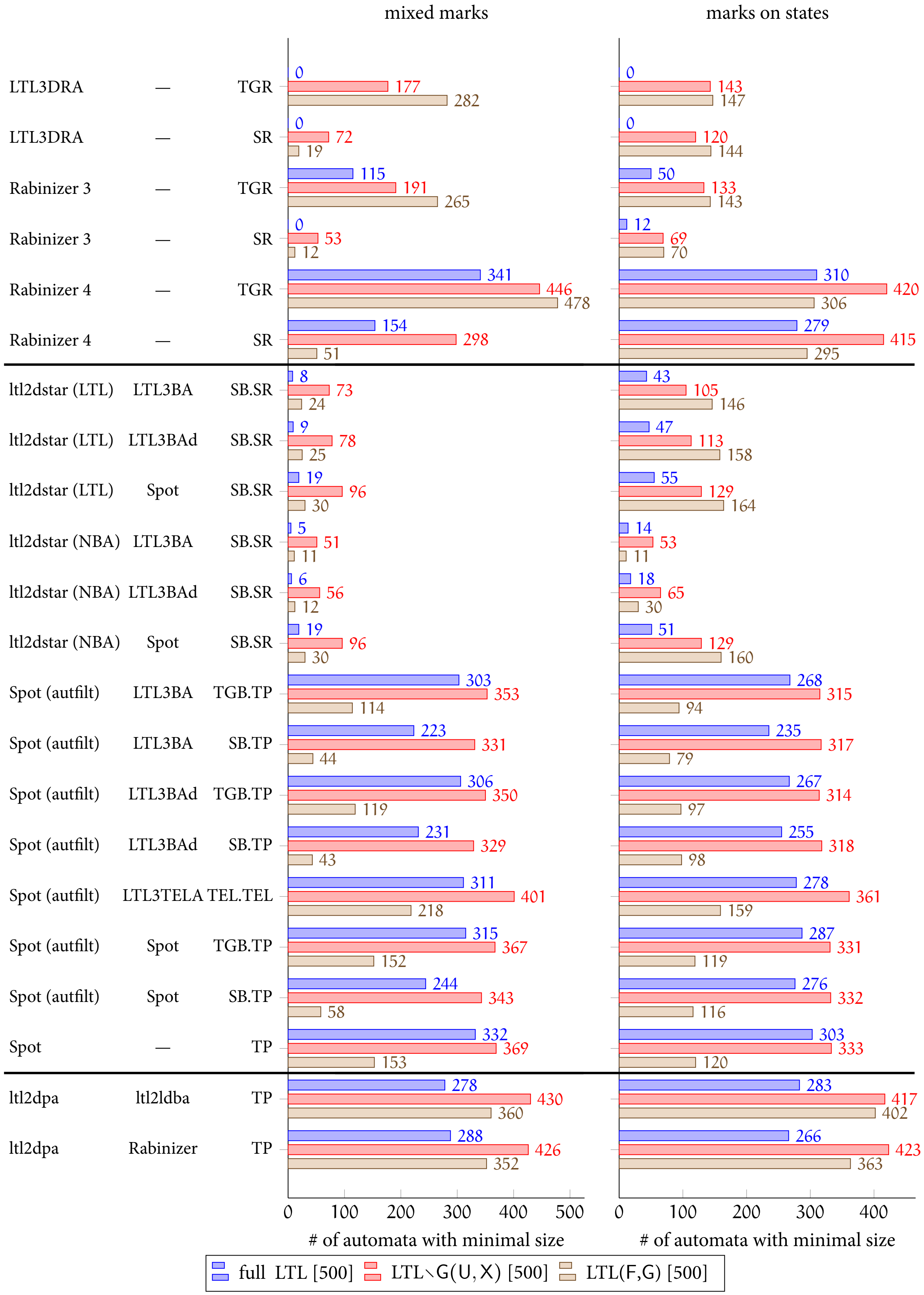


Figure 6.7: The numbers of minimal automata for *random* benchmarks.

Table 6.6: The cumulative numbers for the *literature* benchmarks.

			full LTL [122]			LTL$\smallsetminus\mathsf{G}(\mathsf{U},\mathsf{X})$ [89]		
main tool	intermediate	acc	states	acc	time	states	acc	time
LTL3DRA	—	TGR	—	—	—	315	172	2
		SR	—	—	—	451	276	3
Rabinizer 3	—	TGR	869	426	304	870	191	138
		SR	1707	980	424	1623	548	156
Rabinizer 4	—	TGR	841	375	104	268	196	51
		SR	1350	402	106	386	218	51
ltl2dstar LTL	LTL3BA	SB.SR	1992	398	4	59559	276	24
	LTL3BAd	SB.SR	1189	376	3	59515	274	25
	Spot	SB.SR	1032	338	3	59712	264	24
ltl2dstar NBA	LTL3BA	SB.SR	2170	396	2	78671	264	27
	LTL3BAd	SB.SR	1254	374	1	78644	264	28
	Spot	SB.SR	1034	334	2	59714	242	22
Spot autfilt	LTL3BA	TGB.TP	987	250	2	452	152	1
		SB.TP	1211	283	2	495	148	1
	LTL3BAd	TGB.TP	706	255	1	437	148	1
		SB.TP	738	254	1	482	141	1
	LTL3TELA	TEL.TEL	753	250	122	428	159	122
	Spot	TGB.TP	684	211	1	420	134	1
		SB.TP	688	213	2	448	127	1
Spot	—	TP	680	207	1	420	134	2
ltl2dpa	ltl2ldba	TP	754	327	95	294	201	50
	Rabinizer	TP	901	349	108	414	202	55

Table 6.7: The cumulative numbers for the *random* benchmarks.

			full LTL [479]			LTL$\smallsetminus\mathsf{G}(\mathsf{U},\mathsf{X})$ [494]			LTL(F,G) [500]		
main tool	intermediate	acc	states	acc	time	states	acc	time	states	acc	time
LTL3DRA	—	TGR	—	—	—	3359	1040	4	1359	1571	4
		SR	—	—	—	3962	1826	6	2918	2022	5
Rabinizer 3	—	TGR	4269	2839	145	2766	1546	82	1488	1710	77
		SR	10609	6072	276	4298	3892	97	4165	4012	87
Rabinizer 4	—	TGR	3753	1544	371	2465	1089	329	1045	1468	312
		SR	5704	1582	376	2833	1182	337	2452	1630	317
ltl2dstar LTL	LTL3BA	SB.SR	39058	2008	14	4117	1638	11	6912	2056	10
	LTL3BAd	SB.SR	9238	1830	12	4099	1576	12	7697	2038	11
	Spot	SB.SR	8111	1716	15	4074	1504	12	8563	1980	11
ltl2dstar NBA	LTL3BA	SB.SR	49841	1956	9	4952	1556	5	15223	2016	5
	LTL3BAd	SB.SR	11700	1772	4	4945	1508	4	15522	1984	4
	Spot	SB.SR	8247	1658	6	4116	1428	5	8867	2008	5
Spot autfilt	LTL3BA	TGB.TP	3853	1026	7	2646	817	5	2021	1312	4
		SB.TP	4979	1146	6	2833	833	5	3183	1388	4
	LTL3BAd	TGB.TP	3809	1052	5	2658	833	4	2054	1302	4
		SB.TP	4279	1049	5	2850	817	5	2863	1330	5
	LTL3TELA	TEL.TEL	4346	1071	10	2666	819	6	2270	1329	5
	Spot	TGB.TP	3695	984	6	2611	767	7	1934	1251	7
		SB.TP	4022	986	6	2717	769	5	2428	1263	5
Spot	—	TP	3625	925	5	2606	743	6	1931	1250	6
ltl2dpa	ltl2ldba	TP	6388	1299	359	2536	1115	320	1199	1468	308
	Rabinizer	TP	4391	1323	390	2505	1119	348	1244	1492	327

Cross-comparison (direct). Table 6.8 shows that Rabinizer 4 overall beats the other tools for direct translations. However, there are still many cases where either LTL3DRA or Rabinizer 3 produces a better[37] automaton (see the column 5 which lists the number of losses of Rabinizer 4).

The results of Rabinizer 3 that produces DSRA are particularly bad. They are partially caused by the fact that Rabinizer 3 usually uses a lot of unnecessary acceptance marks; it also suggests that the degeneralization procedure can be improved.

[37] Note that in case of equally sized automata we also compare the numbers of acceptance marks.

Table 6.8: Cross-comparison of the tools that perform a direct translation.

	main tool	intermediate	acc	#	1	2	3	4	5	6	V
full LTL [129]	LTL3DRA	—	TGR	1	—	—	—	—	—	—	—
			SR	2	—	—	—	—	—	—	—
	Rabinizer 3	—	TGR	3	—	—	—	129	21	60	210
			SR	4	—	—	0	—	6	10	16
	Rabinizer 4	—	TGR	5	—	—	101	123	—	74	298
			SR	6	—	—	56	119	3	—	178
LTL∖G(U,X) [92]	LTL3DRA	—	TGR	1	—	91	31	89	30	58	299
			SR	2	1	—	8	84	2	10	105
	Rabinizer 3	—	TGR	3	9	80	—	91	28	55	263
			SR	4	2	5	0	—	1	4	12
	Rabinizer 4	—	TGR	5	37	65	46	91	—	42	281
			SR	6	30	43	34	88	0	—	195
full LTL [500]	LTL3DRA	—	TGR	1	—	—	—	—	—	—	—
			SR	2	—	—	—	—	—	—	—
	Rabinizer 3	—	TGR	3	—	—	—	494	50	210	754
			SR	4	—	—	0	—	2	10	12
	Rabinizer 4	—	TGR	5	—	—	426	498	—	295	1219
			SR	6	—	—	280	489	17	—	786
LTL∖G(U,X) [500]	LTL3DRA	—	TGR	1	—	500	219	474	81	189	1463
			SR	2	0	—	26	434	9	26	495
	Rabinizer 3	—	TGR	3	93	468	—	498	69	203	1331
			SR	4	26	64	0	—	5	8	103
	Rabinizer 4	—	TGR	5	340	444	406	495	—	204	1889
			SR	6	264	401	290	492	2	—	1449
LTL(F,G) [500]	LTL3DRA	—	TGR	1	—	499	208	493	70	397	1667
			SR	2	0	—	22	416	1	50	489
	Rabinizer 3	—	TGR	3	114	466	—	500	48	374	1502
			SR	4	6	81	0	—	0	28	115
	Rabinizer 4	—	TGR	5	309	490	363	500	—	457	2119
			SR	6	65	319	87	472	0	—	943

ltl2dstar. The following three paragraphs combine observations from the cumulative numbers and the cross-comparison in Table 6.9 that was created for the ltl2dstar tool chains. We first address the question: *Which LTL-to-BA translator should we use with ltl2dstar?*

From the cross-comparison it seems that the NBA interface combined with Spot is the best choice. The cumulative numbers confirm this for the number of acceptance marks, but slightly favour the LTL interface for the number of states. The tool chain of LTL interface combined with LTL3BAd offers a very good alternative to the tool chains with Spot; it produces the minimal number of states on the LTL(F,G) benchmark and on the $\mathsf{LTL}\smallsetminus\mathsf{G}(\mathsf{U},\mathsf{X})$ formulae from literature.

The difference between the LTL and NBA interface combined with Spot is negligible, while it is significant with LTL3BA at the same time. The reason for this is that Spot sets the stutter-invariance property in its HOA representation of the produced nondeterministic automata while LTL3BA does not. This property is read by ltl2dstar which can then employ optimizations for stutter-invariant properties.

Spot. The message of the comparison of the tool chains that use Spot for determinization is clear: use (Spot, —, TP). However, there are still some cases where Spot works better with LTL3BA (see the column 8 in Table 6.10); even more, on the fragment LTL(F,G), (Spot (autfilt), LTL3TELA, TEL.TEL) wins the battle in the cross-comparison (but not in the cumulative numbers).

This is expected as LTL3TELA works best on LTL(F,G) formulae and often produces a deterministic automaton. On the other hand, in cases where the intermediate TELA is not deterministic, the tool chain has to employ an expensive transformation into a TBA and produces large automata. These inconsistent results of the tool chain with LTL3TELA can be observed on the quantile plot in Figure 6.8. The figure shows automata sizes of three selected tool chains based on Spot's determinization. For each of these tool chains we have sorted the automata sizes independently and plotted the results. The green line shows the minimal automata achieved by some Spot's tool chain.[38] We can see that (Spot (autfilt), LTL3TELA, TEL.TEL) produced both most of the smallest and also the largest automata from the selected tool chains.

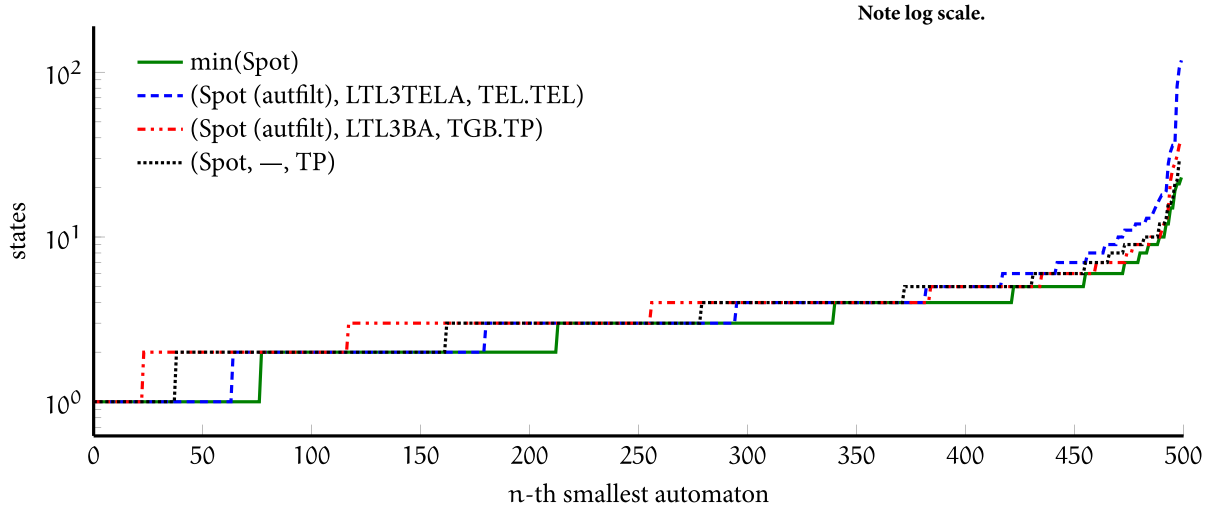


Figure 6.8: Quantile plot of automata sizes of selected tool chains that use Spot for determinization on the LTL(F,G) benchmark. **Note log scale.**

[38] minimal from all tool chains based on Spot, not only from the selected ones

Table 6.9: Cross-comparison of tool chains that use ltl2dstar for determinization of nondeterministic automata.

	main tool	intermediate	acc	#	1	2	3	4	5	6	V
full LTL [129]	ltl2dstar LTL	LTL3BA	SB.SR	1	—	7	6	25	25	7	70
		LTL3BAd	SB.SR	2	55	—	4	70	19	5	153
		Spot	SB.SR	3	66	27	—	79	41	1	214
	ltl2dstar NBA	LTL3BA	SB.SR	4	0	7	5	—	8	5	25
		LTL3BAd	SB.SR	5	51	0	3	57	—	3	114
		Spot	SB.SR	6	67	28	1	79	41	—	216
LTL∖G(U,X) [92]	ltl2dstar LTL	LTL3BA	SB.SR	1	—	3	2	38	32	3	78
		LTL3BAd	SB.SR	2	17	—	3	49	31	3	103
		Spot	SB.SR	3	23	10	—	51	33	1	118
	ltl2dstar NBA	LTL3BA	SB.SR	4	3	2	2	—	4	1	12
		LTL3BAd	SB.SR	5	16	4	5	18	—	3	46
		Spot	SB.SR	6	33	20	10	51	33	—	147
full LTL [500]	ltl2dstar LTL	LTL3BA	SB.SR	1	—	34	44	142	121	53	394
		LTL3BAd	SB.SR	2	189	—	54	269	129	80	721
		Spot	SB.SR	3	229	105	—	303	206	33	876
	ltl2dstar NBA	LTL3BA	SB.SR	4	22	29	29	—	30	25	135
		LTL3BAd	SB.SR	5	165	16	41	204	—	36	462
		Spot	SB.SR	6	225	112	19	292	185	—	833
LTL∖G(U,X) [500]	ltl2dstar LTL	LTL3BA	SB.SR	1	—	24	27	152	141	37	381
		LTL3BAd	SB.SR	2	72	—	21	191	132	33	449
		Spot	SB.SR	3	117	61	—	214	163	15	570
	ltl2dstar NBA	LTL3BA	SB.SR	4	24	36	38	—	30	22	150
		LTL3BAd	SB.SR	5	79	24	37	86	—	15	241
		Spot	SB.SR	6	140	94	36	208	158	—	636
LTL(F,G) [500]	ltl2dstar LTL	LTL3BA	SB.SR	1	—	57	81	455	399	120	1112
		LTL3BAd	SB.SR	2	101	—	81	462	411	126	1181
		Spot	SB.SR	3	130	83	—	451	402	52	1118
	ltl2dstar NBA	LTL3BA	SB.SR	4	5	8	19	—	87	20	139
		LTL3BAd	SB.SR	5	42	10	22	154	—	22	250
		Spot	SB.SR	6	123	84	3	449	400	—	1059

Table 6.10: Cross-comparison of tool chains that use Spot for determinization of nondeterministic automata.

	main tool	intermediate	acc	#	1	2	3	4	5	6	7	8	V
full LTL [129]	Spot autfilt	LTL3BA	TGB.TP	1	—	38	18	31	28	22	25	22	184
			SB.TP	2	16	—	4	2	12	0	2	0	36
		LTL3BAd	TGB.TP	3	23	33	—	22	22	6	13	6	125
			SB.TP	4	26	29	6	—	18	3	3	3	88
		LTL3TELA	TEL.TEL	5	20	27	8	23	—	2	12	2	94
		Spot	TGB.TP	6	45	45	27	36	36	—	12	0	201
			SB.TP	7	44	43	26	30	35	2	—	2	182
	Spot	—	TP	8	49	49	31	40	42	6	16	—	233
LTL∖G(U,X) [92]	Spot autfilt	LTL3BA	TGB.TP	1	—	18	3	16	12	4	14	4	71
			SB.TP	2	2	—	0	3	2	1	3	1	12
		LTL3BAd	TGB.TP	3	4	17	—	16	11	2	13	2	65
			SB.TP	4	4	11	0	—	2	1	2	1	21
		LTL3TELA	TEL.TEL	5	15	24	12	24	—	5	16	5	101
		Spot	TGB.TP	6	22	28	19	28	16	—	15	0	128
			SB.TP	7	18	26	15	18	8	1	—	1	87
	Spot	—	TP	8	22	28	19	28	16	0	15	—	128
full LTL [500]	Spot autfilt	LTL3BA	TGB.TP	1	—	223	48	204	150	79	199	64	967
			SB.TP	2	35	—	32	20	62	31	36	26	242
		LTL3BAd	TGB.TP	3	26	210	—	183	129	48	177	31	804
			SB.TP	4	50	105	37	—	74	28	35	23	352
		LTL3TELA	TEL.TEL	5	65	193	57	159	—	43	149	41	707
		Spot	TGB.TP	6	117	251	104	223	165	—	163	2	1025
			SB.TP	7	116	183	103	128	133	26	—	23	712
	Spot	—	TP	8	178	314	165	291	233	75	231	—	1487
LTL∖G(U,X) [500]	Spot autfilt	LTL3BA	TGB.TP	1	—	110	41	113	66	41	99	39	509
			SB.TP	2	12	—	20	17	15	8	19	6	97
		LTL3BAd	TGB.TP	3	15	91	—	82	48	10	71	10	327
			SB.TP	4	20	31	11	—	23	4	10	3	102
		LTL3TELA	TEL.TEL	5	74	107	80	112	—	52	87	51	563
		Spot	TGB.TP	6	93	145	84	137	88	—	76	0	623
			SB.TP	7	76	107	70	88	67	2	—	2	412
	Spot	—	TP	8	115	170	107	162	113	26	101	—	794
LTL(F,G) [500]	Spot autfilt	LTL3BA	TGB.TP	1	—	321	35	313	124	61	245	60	1159
			SB.TP	2	16	—	26	55	36	16	41	15	205
		LTL3BAd	TGB.TP	3	14	312	—	305	123	44	238	43	1079
			SB.TP	4	18	109	19	—	38	17	39	16	256
		LTL3TELA	TEL.TEL	5	182	354	186	345	—	158	295	156	1676
		Spot	TGB.TP	6	148	334	155	333	149	—	287	0	1406
			SB.TP	7	71	224	78	182	83	14	—	14	666
	Spot	—	TP	8	149	336	156	334	150	2	289	—	1416

Determinization tool chains with marks on states. We have already mentioned that the large automata produced by ltl2dstar in comparison to other tool chains are at least partially due to the placement of marks to states. Therefore, we offer a fair comparison of ltl2dstar and Spot in Figure 6.9. The scatter plots compare the tool chain t_x = (Spot, —, TP) with the tool chain t_y = (ltl2dstar (LTL), Spot, SB.SR) after all marks were pushed to states. A dot on coordinates (x, y) represents the fact that there exists a formula φ in the benchmark such that t_x created an automaton with x states for φ and t_y produced an automaton with y states for the same formula. The color of the dot indicates the number of such formulae. We have merged the literature and random benchmarks to reduce the number of figures.

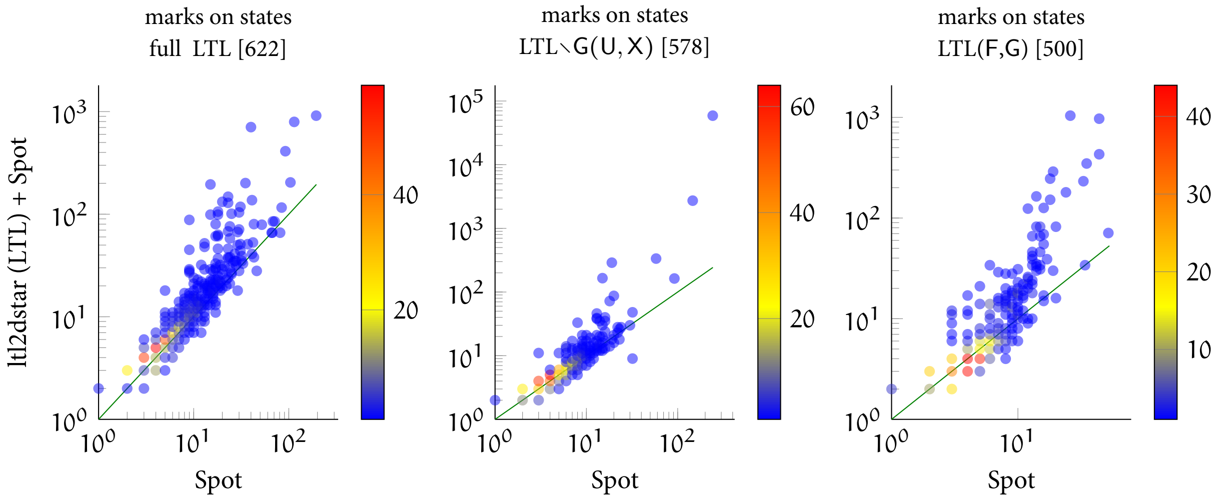


Figure 6.9: Scatter plots comparing ltl2dstar and Spot on automata with marks on states. **Note log scale.**

We can observe that without any doubts Spot is better even after the marks were pushed on states on the full LTL and LTL∖G(U, X) benchmarks and avoids really large automata on LTL(F,G). However, on LTL(F,G) Spot is no longer so dominant (note the red dots below the green line), though, it is still preferable.

Rabinizer 4, Spot, and ltl2dpa. Now we compare the best direct tool and the best determinization tool chain with the two tool chains of ltl2dpa. You can find the cross-comparison in Table 6.11. Spot wins on all benchmarks except the LTL(F,G) benchmark. On the LTL(F,G) benchmark not only Rabinizer 4 clearly wins, but also ltl2dpa beats Spot.

The cross-comparison uses the number of acceptance marks into account for automata of equal size. The scatter plots in Figure 6.10 show how Spot competes with Rabinizer 4 on states only. We have, again, merged the random and literature benchmarks. For the full LTL and LTL∖G(U, X) benchmarks, we zoom (below) into the dense parts of the plots indicated by the red boxes. The dominance of Spot is no longer present here; on the contrary, Rabinizer 4 seems to be slightly preferable.

Table 6.11: Cross-comparison of Rabinizer 4, Spot, and ltl2dpa.

	main tool	intermediate	acc	#	1	2	3	4	V
full LTL [129]	Rabinizer 4	—	TGR	1	—	15	46	63	124
	Spot	—	TP	2	100	—	105	104	309
	ltl2dpa	ltl2ldba	TP	3	45	12	—	43	100
		Rabinizer	TP	4	20	12	28	—	60
LTL∖G(U,X) [92]	Rabinizer 4	—	TGR	1	—	24	41	41	106
	Spot	—	TP	2	59	—	66	67	192
	ltl2dpa	ltl2ldba	TP	3	2	17	—	7	26
		Rabinizer	TP	4	1	15	3	—	19
full LTL [500]	Rabinizer 4	—	TGR	1	—	145	244	209	598
	Spot	—	TP	2	310	—	369	358	1037
	ltl2dpa	ltl2ldba	TP	3	119	118	—	103	340
		Rabinizer	TP	4	109	122	121	—	352
LTL∖G(U,X) [500]	Rabinizer 4	—	TGR	1	—	139	137	136	412
	Spot	—	TP	2	311	—	348	336	995
	ltl2dpa	ltl2ldba	TP	3	52	118	—	28	198
		Rabinizer	TP	4	40	126	27	—	193
LTL(F,G) [500]	Rabinizer 4	—	TGR	1	—	354	341	349	1044
	Spot	—	TP	2	111	—	135	137	383
	ltl2dpa	ltl2ldba	TP	3	73	347	—	33	453
		Rabinizer	TP	4	67	339	7	—	413

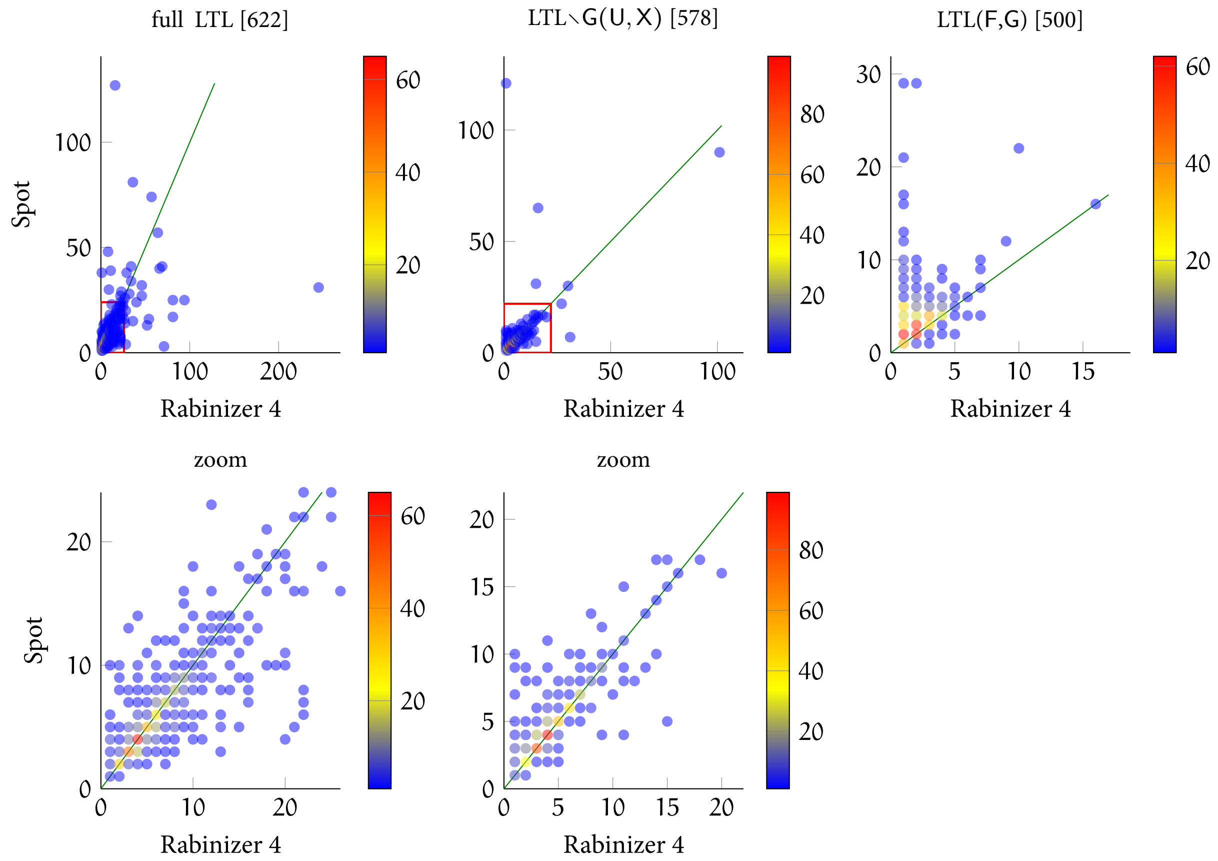


Figure 6.10: Scatter plots comparing Rabinizer 4 and Spot with zooms into dense parts of the plots.

In the cross-comparison of Table 6.10, the tool chain (Spot, —, TP) is only beaten by (Spot (autfilt), LTL3TELA, TEL.TEL) on the LTL(F,G) benchmark. The scatter plots in Figure 6.11 confirms the victory of Rabinizer 4 over Spot even against the tool chain with LTL3TELA.

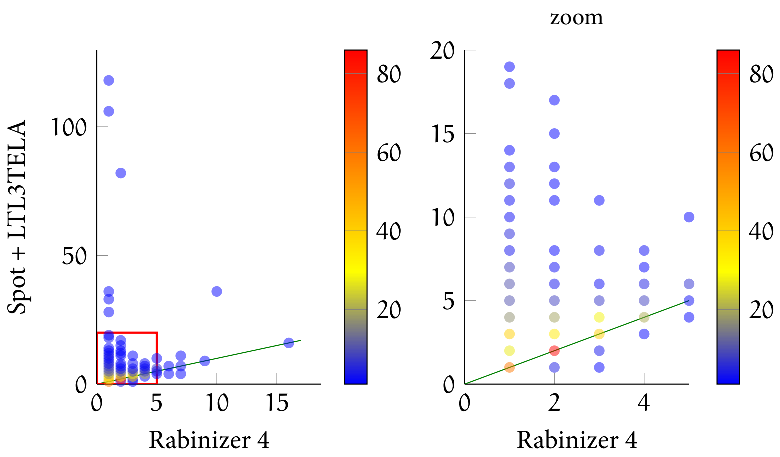


Figure 6.11: Scatter plot comparing Rabinizer 4 against Spot combined with LTL3TELA.

6.5 RESULTS: THE PARAMETRIC BENCHMARKS

We present the results achieved by the considered tool chains on the parametric formulae in Tables 6.12, 6.13, 6.14, and 6.15. For each parametric formula $\varphi(i)$ and each tool chain t, we show two or three numbers. The column *max* shows the maximal parameter i for which t was able to compute an automaton for $\varphi(i)$. The heading $n = j$ for some formula φ means that all tools were able to compute automata for $\varphi(j)$ and that some timeout occurred for $\varphi(j+1)$. If the timeout was preceded by some error (violating the limit of 32 marks), we use two columns named n_e and n_t instead of n, where the value for n_e is the maximum parameter without error and n_t is the maximum parameter without any timeout. The value for t in the column for n shows the number of states of the automaton produced by t for $\varphi(n)$, and analogously for n_e and n_t. The best values (minimal for n, n_e, or n_t and maximal for *max*) for each formula are highlighted in green. Note that all errors encountered here were related to the limit on acceptance marks.

The tables confirm that Rabinizer 4 often produce small automata, but also that it often requires a lot of time. The DTGRA setting reached the maximal n for a given formula only in one case (the DSRA in one more), and it was never the unique tool[39] that achieved the maximal parameter; the same holds for the tool chains that combine ltl2dstar and Spot and the tool chains that use ltl2dpa are only slightly better in this aspect. Surprisingly, LTL3DRA succeeded to reach the maximal parameter in 3 cases as the only tool and ltl2dstar combined with LTL3BA even in 4 cases as the only tool chain. (LTL3DRA, —, SR) was three times able to reach higher parameter than (LTL3DRA, —, TGR). The reason for this unexpected behaviour is the limitation of ltlcross to automata with at most 32 acceptance marks, as (LTL3DRA, —, TGR) violated the limit for lower parameters than (LTL3DRA, —, SR). In all these cases, the TGRA setting of LTL3DRA achieved shorter run times.[40]

[39] In this situation, more tool chains that use the same combination of tools, possibly in different configurations, are considered unique.

[40] Rabinizer 4, on the contrary, does not exhibit the same difference. The run times of the TGRA setting are higher than the ones of the SRA setting.

Table 6.12: Parametric formulae benchmark (*gh* I).

			gh-e		gh-c1		gh-c2		gh-q		
main tool	intermediate	acc	$n = 9$	max	$n = 8$	max	$n = 11$	max	$n_e = 3$	$n_t = 6$	max
LTL3DRA	—	TGR	512	9	1	22	1	13	30	910	6
		SR	512	9	2	22	12	13	30	—	5
Rabinizer 3	—	TGR	512	12	1	14	1	16	18	—	4
		SR	512	10	256	8	12	12	43	—	3
Rabinizer 4	—	TGR	512	12	1	11	1	12	18	240	10
		SR	512	12	256	11	22	12	18	240	10
ltl2dstar LTL	LTL3BA	SB.SR	512	10	2	18	32	19	24	386	7
	LTL3BAd	SB.SR	512	10	2	19	21	15	24	386	8
	Spot	SB.SR	512	12	2	11	21	14	18	240	7
ltl2dstar NBA	LTL3BA	SB.SR	512	11	4	25	23	20	28	491	7
	LTL3BAd	SB.SR	512	10	2	25	12	15	27	542	8
	Spot	SB.SR	512	12	2	11	21	14	18	240	7
Spot autfilt	LTL3BA	TGB.TP	512	11	1	23	1	16	18	240	6
		SB.TP	512	11	2	22	21	11	18	240	7
	LTL3BAd	TGB.TP	512	13	1	24	1	15	18	240	7
		SB.TP	512	10	2	23	12	15	18	240	7
	LTL3TELA	TEL.TEL	512	10	1	22	1	11	18	240	6
	Spot	TGB.TP	512	13	1	11	1	14	18	240	7
		SB.TP	512	13	2	11	12	13	18	240	7
Spot	—	TP	512	14	1	11	1	13	18	240	9
ltl2dpa	ltl2ldba	TP	512	11	1	13	11	13	18	240	9
	Rabinizer	TP	512	12	1	11	11	12	18	240	10

Table 6.13: Parametric formulae benchmark (*gh* II).

			gh-r		gh-s			gh-u		gh-u2	
main tool	intermediate	acc	$n = 2$	max	$n_e = 3$	$n_t = 10$	max	$n = 8$	max	$n = 8$	max
LTL3DRA	—	TGR	1	4	8	1024	12	719	8	9	13
		SR	10	5	8	1024	12	719	8	9	13
Rabinizer 3	—	TGR	1	4	8	—	3	129	11	9	17
		SR	12	4	35	—	3	129	10	9	12
Rabinizer 4	—	TGR	1	4	7	—	8	128	12	8	14
		SR	13	5	7	—	8	128	12	8	14
ltl2dstar LTL	LTL3BA	SB.SR	756	3	8	1024	12	129	9	9	13
	LTL3BAd	SB.SR	66	4	8	1024	12	129	9	9	13
	Spot	SB.SR	286	3	8	1024	12	129	12	9	14
ltl2dstar NBA	LTL3BA	SB.SR	1304	2	9	1025	13	129	9	9	14
	LTL3BAd	SB.SR	152	3	9	1025	13	129	9	9	13
	Spot	SB.SR	286	3	9	1025	12	129	12	9	14
Spot autfilt	LTL3BA	TGB.TP	9	5	8	1024	13	128	9	8	13
		SB.TP	66	4	8	1024	13	128	9	8	13
	LTL3BAd	TGB.TP	9	5	8	1024	13	128	12	8	14
		SB.TP	13	5	8	1024	13	128	9	8	14
	LTL3TELA	TEL.TEL	16	4	8	1024	10	128	10	8	8
	Spot	TGB.TP	16	4	8	1024	13	128	12	8	14
		SB.TP	18	4	8	1024	13	128	12	8	14
Spot	—	TP	16	4	7	1023	13	128	13	8	14
ltl2dpa	ltl2ldba	TP	4	4	7	1023	11	128	11	8	14
	Rabinizer	TP	4	4	7	—	8	128	12	8	14

Table 6.14: Parametric formulae benchmark (*ms* and *go*).

			ms-phi-r		ms-phi-s		go-theta		ms-phi-h		
main tool	intermediate	acc	$n = 2$	max	$n = 1$	max	$n = 5$	max	$n_e = 2$	$n_t = 3$	max
LTL3DRA	—	TGR	1	4	1	3	2	12	—	—	—
		SR	24	5	8	5	7	12	—	—	—
Rabinizer 3	—	TGR	1	4	1	3	2	15	5	11	3
		SR	27	3	9	3	11	9	20	—	2
Rabinizer 4	—	TGR	1	4	1	3	2	10	5	19	7
		SR	33	4	11	4	11	10	13	41	7
ltl2dstar LTL	LTL3BA	SB.SR	147	2	49	1	15	17	18	194	4
	LTL3BAd	SB.SR	99	2	33	1	5444	5	18	194	4
	Spot	SB.SR	264	2	88	1	5444	5	18	194	4
ltl2dstar NBA	LTL3BA	SB.SR	10541	2	87	1	13	18	285	29635	3
	LTL3BAd	SB.SR	12012	2	94	1	8934	5	285	29452	3
	Spot	SB.SR	5418	2	82	1	5444	5	285	29452	3
Spot autfilt	LTL3BA	TGB.TP	24	3	7	3	230	7	25	199	5
		SB.TP	152	3	28	2	523	7	21	171	5
	LTL3BAd	TGB.TP	24	3	7	3	230	7	25	199	5
		SB.TP	88	3	18	3	490	6	21	171	5
	LTL3TELA	TEL.TEL	31	3	7	3	167	7	22	175	5
	Spot	TGB.TP	29	3	8	3	230	7	21	170	5
		SB.TP	76	3	16	3	436	6	21	170	5
Spot	—	TP	29	3	8	3	230	7	21	170	5
ltl2dpa	ltl2ldba	TP	6	3	4	3	6	10	7	15	11
	Rabinizer	TP	6	4	2	3	6	10	27	244	5

Table 6.15: Parametric formulae benchmark (*kr* and *other*).

			and-fg		or-fg		kr-n			kr-nlogn	
main tool	intermediate	acc	$n = 11$	max	$n = 5$	max	$n_e = 1$	$n_t = 2$	max	$n = 1$	max
LTL3DRA	—	TGR	1	22	1	20	—	—	—	—	—
		SR	2	22	32	13	—	—	—	—	—
Rabinizer 3	—	TGR	1	17	1	14	28	152	2	34	2
		SR	3	12	32	8	29	153	2	35	2
Rabinizer 4	—	TGR	1	12	1	12	23	196	2	37	1
		SR	2	12	32	12	23	196	2	37	1
ltl2dstar LTL	LTL3BA	SB.SR	2	24	32	13	13	83	3	20	3
	LTL3BAd	SB.SR	2	24	32	13	13	83	3	20	3
	Spot	SB.SR	2	11	32	12	13	83	3	20	2
ltl2dstar NBA	LTL3BA	SB.SR	3	24	77775	5	13	83	3	20	3
	LTL3BAd	SB.SR	3	24	77775	5	13	83	3	20	3
	Spot	SB.SR	2	11	58852	5	13	83	3	20	2
Spot autfilt	LTL3BA	TGB.TP	2	21	121	7	12	82	3	19	3
		SB.TP	2	21	121	7	12	82	3	19	3
	LTL3BAd	TGB.TP	2	21	121	7	12	82	3	19	3
		SB.TP	2	21	121	7	12	82	3	19	3
	LTL3TELA	TEL.TEL	1	23	121	7	12	—	1	19	1
	Spot	TGB.TP	2	11	121	7	12	82	3	19	2
		SB.TP	2	11	121	7	12	82	3	19	2
Spot	—	TP	2	11	121	7	12	82	3	19	2
ltl2dpa	ltl2ldba	TP	1	14	5	12	13	96	2	20	1
	Rabinizer	TP	1	12	120	7	23	196	2	37	1

6.6 FINAL WORDS

The situation with LTL to deterministic automata translation changed substantially since 2013. The former leading tool chains of LTL3DRA or ltl2dstar combined with Spot are now surpassed by Rabinizer 4 and Spot. However, there is still space for improvement for both Rabinizer 4 and Spot as neither of the tools dominated entirely.

General recommendations. We recommend using either (Spot, —, TP) or (Rabinizer 4, —, TGR) for translation of LTL into deterministic automata. Rabinizer 4 is preferable if the small size of automata is in the main focus. On the contrary, Spot is preferable for situations, where the complexity of the acceptance condition or computation time are an issue. Spot is also the right choice for LTL translation to automata that are further converted to games[41], as Spot produces automata with the parity acceptance condition.

Portfolio approach. As no tool dominates the others in all cases, the portfolio approach, where you run more translators and choose the best result, is also an appealing option. The portfolio of the following nine tool chains produces a minimal automaton for all considered cases except 4 randomly generated formulae outside $\mathrm{LTL} \smallsetminus \mathsf{G}(\mathsf{U}, \mathsf{X})$.

- (Rabinizer 4, —, TGR)
- (Rabinizer 3, —, TGR)
- (LTL3DRA, —, TGR)
- (Spot, —, TP)
- (Spot (autfilt), LTL3TELA, TEL.TEL)
- (Spot (autfilt), LTL3BA, TGB.TP)
- (Spot (autfilt), Spot, SB.TP)
- (ltl2dpa, ltl2ldba TP)
- (ltl2dstar (LTL), LTL3BAd, SB.SR)

Suggestions for tool developers. Spot showed the biggest weakness on the LTL(F,G) fragment. This observation can be a good starting point for the further development of determinization procedures in Spot. The developers of Rabinizer 4 could find some inspiration in its predecessor, Rabinizer 3.1, as we have identified more than 200 formulae where Rabinizer 3.1 produces better DTGRA than Rabinizer 4.

[41] synthesis of reactive systems, for example

Part III

SEMI-DETERMINISTIC AUTOMATA

7 *Semi-Determinization and Cut-Determinization of Generalized Büchi Automata*

In this chapter, we introduce semi-deterministic and cut-deterministic automata and discuss several algorithms related to them. Our focus here is on methods that convert a given nondeterministic (generalized) Büchi automaton into an equivalent semi-deterministic or cut-deterministic automaton. Semi-deterministic automata have so far only been considered with Büchi or generalized Büchi acceptance. For this reason, we omit the acceptance formula from the figures as it is always a conjunction of Inf terms for all marks present in the automaton. For the sake of clarity, we use classical alphabet throughout this and the following chapter. All results naturally apply also on automata with propositional alphabets.

7.1 SEMI-DETERMINISM AND CUT-DETERMINISM

A generalized Büchi automaton $(S, \Sigma, \delta, s_I, M, \mu, \Phi)$ is *semi-deterministic* if $S = S_N \cup S_D$ is a union of two disjoint sets S_N and S_D and the transition relation $\delta = \delta_N \cup \delta_c \cup \delta_D$ consists of three disjoint transition relations, namely

$$\delta_N : S_N \times \Sigma \times S_N, \qquad \delta_c : S_N \times \Sigma \times S_D, \text{ and } \qquad \delta_D : S_D \times \Sigma \times S_D,$$

where there is no transition from S_D to S_N[1] and δ_D is deterministic.[2] Moreover, marks are placed only on elements of S_D and δ_D. The elements of δ_c are called *cut transitions*. Figure 7.1 illustrates these conditions visually.

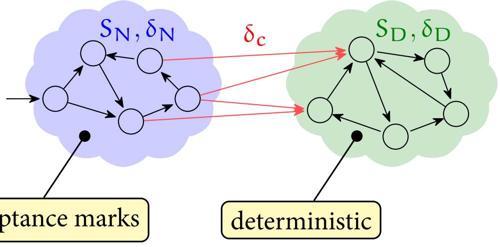


Figure 7.1: Structure of a semi-deterministic automaton. The green cloud is deterministic and contains all accepting transitions and states that are reachable from them. In a cut-deterministic automaton, the blue cloud is deterministic too.

A GBA is *cut-deterministic* if it is semi-deterministic and its δ_N is also deterministic. Intuitively, nondeterminism in a cut-deterministic automaton can be induced only by the cut transitions from S_N to S_D. The term *cut-determinism* is inspired by the graph-theoretic notion of cut (δ_c) and its purpose is to disambiguate the overloaded term *semi-determinism*.[3]

Clearly, every deterministic automaton is also cut-deterministic and every cut-deterministic automaton is also semi-deterministic. The opposite relations do not hold.

[1] $\delta \cap (S_D \times \Sigma \times S_N) = \emptyset$

[2] For each state $s \in S_D$ and each letter $a \in \Sigma$, there is at most one state s' such that $(s, a, s') \in \delta_D$.

[3] Vardi and Wolper (1986), [2]; Hahn et al. (2015), [10]; Blahoudek et al. (2016), [16].

7.2 CUT-DETERMINISM CHECK & STATE SPACE PARTITION

To check cut- or semi-determinism of a given GBA $\mathcal{A} = (S, \Sigma, \delta, s_I, M, \mu, \Phi)$ we have to (i) compute a suitable partition of S and δ into S_N, S_D, δ_N, δ_c, and δ_D, and (ii) check determinism of δ_D and δ_N. We address the two tasks in one algorithm simultaneously. The algorithm computes two partitions of states: one such that S_D is minimal and one such that S_D is maximal permissible; based on these partitions it decides whether $\mathcal{A}$ is semi- and cut-deterministic.

The knowledge of some partition of S into S_N and S_D is also needed for cut-determinization of semi-deterministic automata studied in Section 7.7 and is beneficial for complementation of semi-deterministic automata studied in the next chapter. The small size of S_D is favourable for complementation, while the converse holds for cut-determinization as cut-determinization exponentially increases the size of $S_N = S \setminus S_D$.

Topological order on SCCs. Let $\mathcal{T} = (S, \Sigma, \delta, s_I)$ be a semiautomaton and $\mathcal{C}$ the set of its maximal strongly connected components. The *SCC graph* of $\mathcal{T}$ is the directed acyclic graph $(\mathcal{C}, E)$ where

$$E = \{(S_1, S_2) \mid S_1, S_2 \in \mathcal{C} \text{ and } (s_1, a, s_2) \in \delta \text{ where } s_1 \in S_1, s_2 \in S_2, a \in \Sigma\}.$$

A *topological order on SCCs* is a total order $\preceq$ on $\mathcal{C}$ such that for each edge $(S_1, S_2) \in E$ it holds that $S_1 \preceq S_2$.

We assume that acceptance marks are placed only at transitions that are inside some SCC. This is a valid assumption because all the other marks can be removed without altering the language of $\mathcal{A}$. We further assume that we have some topological order $\preceq$ on SCCs of $\mathcal{A}$ and we order the SCCs of $\mathcal{A}$ based on $\preceq$ as $S_0 \preceq S_1 \preceq \ldots \preceq S_n$. Traversal in the topological order $\preceq$ starts with S_0 and traversal in the reverse topological order starts with S_n.

The algorithm consists of two traversals of the SCC graph of $\mathcal{A}$ and two determinism checks. During each traversal, it can move all states of the current SCC between the sets S_N and S_D. With each movement, we also update δ_N, δ_c, and δ_D accordingly. The first traversal starts with all states in S_N, which gives $S_N = S$ and $S_D = \emptyset$. In each phase, we reference the current SCC by S_i and by E we reference the edges of the SCC graph of $\mathcal{A}$.

1. **Partition with minimal S_D [topological order].** SCCs that are accepting or reachable from some accepting SCC must be in S_D by definition.
   - if S_i is accepting, move it to S_D
   - if $S_i \subseteq S_D$ move all S_j such that $(S_i, S_j) \in E$ to S_D.
   - **Semi-determinism check.** $\mathcal{A}$ is semi-deterministic iff δ_D is deterministic. If $\mathcal{A}$ is not semi-deterministic, we stop the algorithm.
2. **Partition with maximal S_D [reverse topological order].** Deterministic SCCs from which only deterministic SCCs are reachable can belong to S_D.
   - if S_i is deterministic and if all S_j such that $(S_i, S_j) \in E$ are in S_D, move S_i to S_D.
   - **Cut-determinism check.** $\mathcal{A}$ is cut-deterministic iff δ_N is deterministic.

We only add SCCs to S_D in this traversal; therefore, we skip all SCCs that are already in S_D.

7.3 SUBSET CONSTRUCTION

All semi-determinization procedures to be presented here rely in their heart on the subset construction known from the determinization of finite automata over finite words. Let $\mathcal{T} = (S, \Sigma, \delta, s_I)$ be a semiautomaton. The function $\tau_\delta : 2^S \times \Sigma \to 2^S$ computes one step of the subset construction for $\mathcal{T}$. Intuitively, $\tau_\delta(P, a)$ gives us the set of states to which we have in $\mathcal{T}$ a transition under a from some state in P. Formally, τ_δ is defined for a set $P \subseteq S$ and a letter $a \in \Sigma$ as follows.

$$\tau_\delta(P, a) = \big\{ s \in S \mid p \in P, (p, a, s) \in \delta \big\}$$

We write τ instead of τ_δ when δ is clear from context. Finally, the subset construction on $\mathcal{T}$ creates a semiautomaton $(2^S, \Sigma, \delta', \{s_I\})$ with transitions defined by τ, that is $(P, a, \tau(P, a)) \in \delta'$ for all $a \in \Sigma$ and all $P \subseteq S$. We usually consider only the sets and transitions reachable by δ' from $\{s_I\}$.

For an automaton $\mathcal{A} = (S, \Sigma, \delta, s_I, M, \mu, \Phi)$ with marks on transitions and for some of its marks $\bullet \in M$ we define a function $\overset{\bullet}{\tau}_\delta : 2^S \times \Sigma \to 2^S$ that restricts τ_δ to transitions marked by $\bullet$ only.

$$\overset{\bullet}{\tau}_\delta(P, a) = \big\{ s \in S \mid p \in P, (p, a, s) \in \delta \cap \mu(\bullet) \big\}$$

7.4 SEMI-DETERMINIZATION OF BÜCHI AUTOMATA

The first semi-determinization procedure for Büchi automata was published by Courcoubetis and Yannakakis in 1988.[4] We start with a more general version of their algorithm – we expect automata with marks on transitions while the original version used accepting states. In the transition-based view, all marks that are placed on states are moved to outgoing transitions; see an example in Figure 7.2.

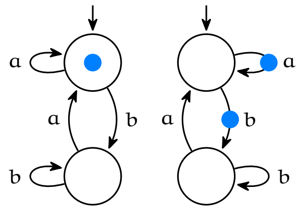


Figure 7.2: Two equivalent Büchi automata: marks on states (left) and marks on transitions (right).

Let $\mathcal{A} = (S, \Sigma, \delta, s_I, \{\bullet\}, \mu, \mathsf{Inf}\,\bullet)$ be a Büchi automaton with marks on transitions. In the following we present a basic construction of an equivalent semi-deterministic automaton $\mathcal{SD} = (Q, \Sigma, \delta_{\mathcal{SD}}, q_I, \{\bullet\}, \mu_{\mathcal{SD}}, \mathsf{Inf}\,\bullet)$ where $Q = Q_N \cup Q_D$ and $\delta_{\mathcal{SD}} = \delta_N \cup \delta_c \cup \delta_D$. The nondeterministic (blue) part N of $\mathcal{SD}$ is the semiautomaton of $\mathcal{A}$ and states of the deterministic (green) part D are formed by pairs (R, B) of subsets of S, where $B \subset R$.

$$Q_N = S \qquad q_I = s_I \qquad \delta_N = \delta \qquad Q_D \subseteq (2^S \times 2^S)$$

For each marked transition of the form (s_1, a, s_2) of $\mathcal{A}$ we have a cut transition $\big(s_1, a, (\{s_2\}, \emptyset)\big)$ from N to D.

$$\delta_c = \Big\{ \big(s_1, a, (\{s_2\}, \emptyset)\big) \mid s_2 \in \overset{\bullet}{\tau}(\{s_1\}, a) \Big\}$$

In a state (R, B) of Q_D R stands for *reachable* and tracks runs of $\mathcal{A}$ using the subset construction. The name B stands for *breakpoint* and tracks the runs that used some marked transition since the last accepting transition of $\mathcal{SD}$ was taken. This behaviour is captured by δ_1.

$$\begin{aligned} \delta_1 = \Big\{ \big((R_1, B_1), a, (R_2, B_2)\big) \mid a \in \Sigma, B_1 \subset R_1 \subseteq S,& \\ R_2 = \tau(R_1, a) \neq \emptyset, \text{ and}& \\ B_2 = \tau(B_1, a) \cup \overset{\bullet}{\tau}(R_1, a) \Big\}& \end{aligned}$$

[4] Courcoubetis and Yannakakis (1988), "Verifying Temporal Properties of Finite-State Probabilistic Programs", [9].

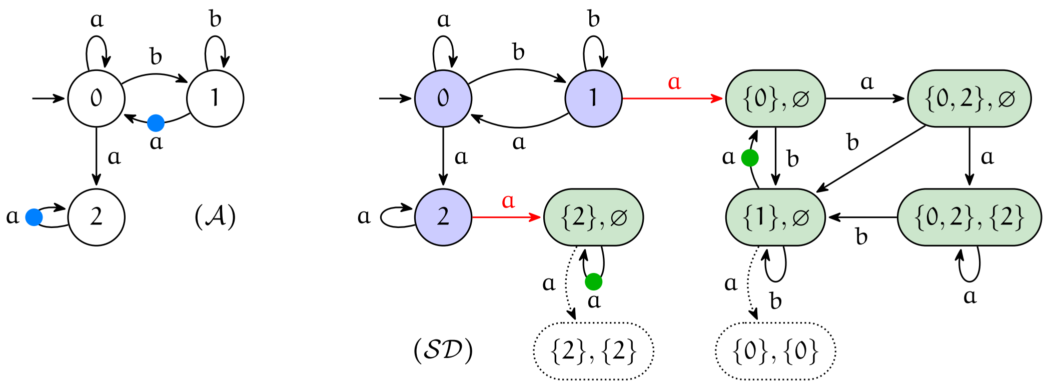


Figure 7.3: A Büchi automaton $\mathcal{A}$ (left) and the corresponding semi-deterministic Büchi automaton $\mathcal{SD}$ (right). The transitions from $\delta_1 \setminus \delta_D$ and the tuples that would become reachable by those transitions are dotted and are not in fact parts of $\mathcal{SD}$.

In some transitions of δ_1 we have $R_2 = B_2$ which means that all states in R_2 can be reached by a run of $\mathcal{A}$ that used a marked transition since B was empty. We mark such transitions by ● and reset B_2 to $\varnothing$.

$$\delta_2 = \Big\{\big((R_1, B_1), a, (R_2, \varnothing)\big) \mid \big((R_1, B_1), a, (R_2, B_2)\big) \in \delta_1 \text{ and } R_2 = B_2\Big\}$$

$$\delta_D = \Big\{\big((R_1, B_1), a, (R_2, B_2)\big) \in \delta_1 \mid R_2 \neq B_2\Big\} \cup \delta_2$$

$$\mu_{\mathcal{SD}}(\bullet) = \delta_2$$

The tuple (R_2, B_2) is not colored by green on purpose as it violates $B_2 \subset R_2$ and is not in fact a state from Q_D.

You can see the construction applied to an example automaton in Figure 7.3. For a better understanding of the reader, the figure also contains transitions from δ_1 that are not part of $\mathcal{SD}$. The automaton $\mathcal{SD}$ has 8 states.

Complexity. Let $s \in S$ be a state of $\mathcal{A}$. Then it is present in Q_N itself, and for a state $q = (R, B) \in Q_D$ it holds that either s is only in R, s is in both R and B, or s is not present in q at all. Therefore, the size of Q is bounded by $|Q| \leq |S| + 3^{|S|}$.

7.4.1 Correctness

Before we prove the correctness of our construction, we need to introduce a handful of notation, including the notions of finite words and run graphs. We will also prove an important property of run graphs in Lemma 7.1.

Finite words. A *finite word* over an alphabet Σ is a finite sequence of letters $w = w_0 w_1 \dots w_k \in \Sigma^*$. The empty word is denoted by ε. Let $u \in \Sigma^\omega$ be an infinite word. By $u_{i..j}$ for some $i \leq j$ we denote the finite word $u_i u_{i+1} \dots u_j$ and we use $u_{..j}$ as a shorthand for $u_{0..j}$ to denote the prefix of u of length j.

Extended subset function θ. The function $\theta_\delta : 2^S \times \Sigma^* \to 2^S$ extends τ_δ to finite words. It is recursively defined for a set of states $P \subseteq S$, a letter $a \in \Sigma$, and a finite word $w \in \Sigma^*$ using τ_δ as follows.

$$\theta_\delta(P, \varepsilon) = P$$

$$\theta_\delta(P, aw) = \theta_\delta\big(\tau_\delta(P, a), w\big)$$

Run graphs. Let $\mathcal{A} = (S, \Sigma, \delta, s_I, M, \mu, \Phi)$ be an automaton with marks on transitions and let $u = u_0 u_1 \ldots \in \Sigma^\omega$ be a word. A *run graph* of $\mathcal{A}$ over u is an edge-labelled directed acyclic graph $G_u^{\mathcal{A}} = (V, E, \overline{\mu})$ where $V \subseteq S \times \omega$ is a set of vertices, $E \subseteq V \times V$ is a set of edges and $\overline{\mu}\colon M \to 2^E$ is a labelling function. V, E, and $\overline{\mu}$ are defined as follows.

$$\begin{aligned} V &= \{(s_I, 0)\} \cup \{(s, i+1) \mid i \geq 0 \text{ and } s \in \theta_\delta(\{s_I\}, u_{..i})\} \\ E &= \{((s_1, i), (s_2, i+1)) \in V \times V \mid i \geq 0 \text{ and } (s_1, u_i, s_2) \in \delta\} \\ \overline{\mu}(●) &= \{((s_1, i), (s_2, i+1)) \in E \mid (s_1, u_i, s_2) \in \mu(●)\} \text{ for each } ● \in M \end{aligned}$$

The vertices on the ith level represent states reachable in $\mathcal{A}$ under the prefix of u of length i. Edges correspond to transitions and also the placement of marks is preserved.

Each infinite path in $G_u^{\mathcal{A}}$ that starts in $(s_I, 0)$ represents a run of $\mathcal{A}$ over u and conversely, each run of $\mathcal{A}$ over u is represented by a unique infinite path in $G_u^{\mathcal{A}}$.

Lemma 7.1. *Let* $\sigma = t_0 t_1 \ldots \in \delta^\omega$ *where* $t_i = (s_i, u_i, s_{i+1})$ *be a run of* $\mathcal{A}$ *over some word* u*. Then there is an index* k *such that for all* $l \geq k$ *there exists an index* $m > l$ *such that* $\theta(\{s_l\}, u_{l..m}) = \theta(\{s_k\}, u_{k..m})$.

Proof. Obviously, $\theta(\{s_l\}, u_{l..m}) \subseteq \theta(\{s_k\}, u_{k..m})$ holds as $t_k \ldots t_{l-1}$ witness that $s_l \in \theta(\{s_k\}, u_{k..l-1})$. We prove the converse by contradiction. Suppose that for all k there is an index $l \geq k$ such that for all $m > l$ it holds that $\theta(\{s_l\}, u_{l..m}) \subset \theta(\{s_k\}, u_{k..m})$. Then we have a strictly increasing sequence of indices $k_0, k_1, \ldots k_{|Q|}$ such that for each $0 \leq i < |Q|$ and all $m > k_{|Q|}$ we have $\theta(\{s_{k_{i+1}}\}, u_{k_{i+1}..m}) \subset \theta(\{s_{k_i}\}, u_{k_i..m})$. In that case $\theta(\{s_{k_{|Q|}}\}, u_{k_{|Q|}..m}) = \emptyset$ which contradicts the existence of σ. □

The idea of the construction of $\mathcal{SD}$ is that a run σ' of $\mathcal{SD}$ follows some run σ of $\mathcal{A}$ in N and nondeterministically guesses, when σ reaches the index k from the previous lemma. At this point, σ' jumps to the D with the first transition of σ marked by ●. If the guess was correct, σ' checks in D whether or not σ is accepting. The proof of the following lemma shows that $\mathcal{SD}$ can verify that σ is accepting.

Lemma 7.2. *Let* $u \in L(\mathcal{A})$ *be a word accepted by* $\mathcal{A}$*. Then* u *is also accepted by the automaton* $\mathcal{SD}$ *built for* $\mathcal{A}$*.*

Proof. Let $\sigma = t_0 t_1 \ldots \in \delta^\omega$ where $t_i = (s_i, u_i, s_{i+1})$ be an accepting run of $\mathcal{A}$ over u. Let k be the minimal index such that it satisfies Lemma 7.1 for σ and $(s_k, u_k, s_{k+1}) \in \mu(●)$. We build the run $\sigma' = t_0' t_1' \ldots$ of $\mathcal{SD}$ over u as follows. For $0 \leq j < k$ σ' stays in N, follows σ and we have $t_j' = t_j$; for k the run takes the cut transition $(s_k, u_k, (\{s_{k+1}\}, \emptyset))$ to D and the rest is deterministic.

$$\sigma' = t_0' \ldots t_{k-1}' t_k' t_{k+1}' \ldots \text{ where}$$

$$t_j' = \begin{cases} (s_j, u_j, s_{j+1}) & \text{if } j < k \\ (s_k, u_j, (\{s_{k+1}\}, \emptyset)) & \text{if } j = k \\ ((R_j, B_j), u_j, (R_{j+1}, B_{j+1})) & \text{if } j > k \end{cases}$$

From the existence of σ we know that R_j is never empty. To show that σ' is accepting we have to prove that we have infinitely many indices $i > k$ such that $((R_i, B_i), u_i, (R_{i+1}, \emptyset)) \in \delta_2$. Suppose that this is not the case and thus there exist an index $f > k$ such that for all $i > f$ we have in σ' only transitions from δ_1, which means that $R_i \supset B_i$. Let $n > f$ be an index such that (s_n, u_n, s_{n+1}) is a transition from σ with ● and thus $s_{n+1} \in B_{n+1}$. It follows from Lemma 7.1 that there is an index $m > n$ such that $R_{m+1} = \theta_\delta(\{s_{k+1}\}, u_{k+1..m}) = \theta_\delta(\{s_{n+1}\}, u_{n+1..m}) \subseteq B_{m+1}$ and thus $R_{m+1} = B_{m+1}$ which contradicts the assumption that $R_i \supset B_i$ for all $i > f$. □

Lemma 7.3. *Let $u \in L(\mathcal{SD})$ be a word accepted by $\mathcal{SD}$. Then u is also accepted by $\mathcal{A}$.*

Proof. Let σ be an accepting run of $\mathcal{SD}$ over u. Then σ has the form $\sigma = t_0 \ldots t_{k-1} t_k t_{k+1} \ldots$ where $k \geq 0$ and $t_j = (s_j, u_j, s_{j+1}) \in \delta_N$ for $0 \leq j < k$, $t_k = \big(s_k, u_k, (\{s_{k+1}\}, \emptyset)\big) \in \delta_c$ and $t_j = \big((R_j, B_j), u_j, (R_{j+1}, B_{j+1})\big) \in \delta_D$ for $j > k$ and $s_0 = s_I$. Moreover, for $j > k$ we use B'_{j+1} to denote the set such that $\big((R_j, B_j), u_j, (R_{j+1}, B'_{j+1})\big) \in \delta_1$ for the cases where $t_j \in \delta_2$.

The tuple $(R_{j+1}, B'_{j+1}) = (R_{l_i}, R_{l_i})$ is not colored by green as it is not in fact a state from Q_D.

We now prove that the run graph $G^{\mathcal{A}}_u = (V, E, \overline{\mu})$ contains an infinite path that starts in $(s_I, 0)$ and that contains infinitely many edges from $\overline{\mu}(\bullet)$; such path represents an accepting run of $\mathcal{A}$ over u.

From definitions of δ_N and δ_c it follows that there is a path from $(s_I, 0)$ to $(s_{k+1}, k+1)$ in $G^{\mathcal{A}}_u$. Now let $S_i = \{s \mid (s, i) \in V\}$ denote the set of states on level i of $G^{\mathcal{A}}_u$. By construction of $G^{\mathcal{A}}_u$ we have that $R_i \subseteq S_i$ for $i > k$. As R_i is never empty and $R_i = \theta(\{S_{k+1}\}, u_{k+1..i-1})$, there is an infinite path that starts in the vertex $(s_{k+1}, k+1)$, thus we have an infinite path from $(s_I, 0)$.

It remains to show that some infinite path contains infinitely many edges from $\overline{\mu}(\bullet)$. Let $l_0, l_1, \ldots$ be an infinite increasing sequence of indices, called *breakpoints*, such that $t_{l_i - 1} \in \mu_{\mathcal{SD}}(\bullet) = \delta_2$. For each breakpoint l_i it holds that $B'_{l_i} = R_{l_i}$ and $B_{l_i} = \emptyset$. In the following we use l to denote some l_i and l' to denote the corresponding l_{i+1}. By definition of δ_1 and $G^{\mathcal{A}}_u$, there is a path with a ● to each vertex (s', l') with $s' \in R_{l'}$ from some vertex (s, l) such that $s \in R_l$, otherwise $s' \notin B'_{l'}$ and thus l' could not be a breakpoint. Overall, we have a path in $G^{\mathcal{A}}_u$ that represents an accepting run of $\mathcal{A}$ over u. □

For each state $s' \in R_{l'}$ we have a state $s'' \in R_m$ for some $l \leq m < l'$ such that $s' \in \theta(P, u_{m+1..l'-1})$ and $P = \overset{\bullet}{\tau}(\{s''\}, u_m)$.

Theorem 7.4. *For each Büchi automaton $\mathcal{A}$ with n states and with marks on transitions we can construct a semi-deterministic Büchi automaton $\mathcal{SD}$ with at most $n + 3^n$ states such that $L(\mathcal{SD}) = L(\mathcal{A})$.*

Proof. The theorem follows from Lemmata 7.2 and 7.3. □

7.4.2 SCC-aware optimization

Let σ be an accepting run of $\mathcal{A}$. All recurrently visited transitions of σ are between states from one strongly connected component $C \subseteq S$. Therefore, we can simplify the construction of D. Namely, we can constrain Q_D to states (R, B) such that $B \subseteq R \subseteq C$ where C is some SCC of $\mathcal{A}$. The construction of δ_1 has to be corrected accordingly:

The definitions of δ_2 and δ_D depend on δ_1, so it is enough to change δ_1 only.

$$\begin{aligned}\delta_1 = \Big\{\big((R_1,B_1),a,(R_2,B_2)\big) \mid\ & a \in \Sigma, B_1 \subset R_1 \subseteq C, C \text{ is a SCC of } \mathcal{A},\\ & R_2 = \tau(R_1,a) \cap C, R_2 \neq \emptyset, \text{ and}\\ & B_2 = \big(\tau(B_1,a) \cup \overset{\bullet}{\tau}(R_1,a)\big) \cap C\Big\}\end{aligned}$$

You can see the effect of the optimization in Figure 7.4. The optimization saved 2 out of 5 states of the deterministic component of $\mathcal{SD}$ and thus the final automaton has 6 instead of 8 states. We left the now unreachable states in the figure for the reader to better see the effect of the optimization.

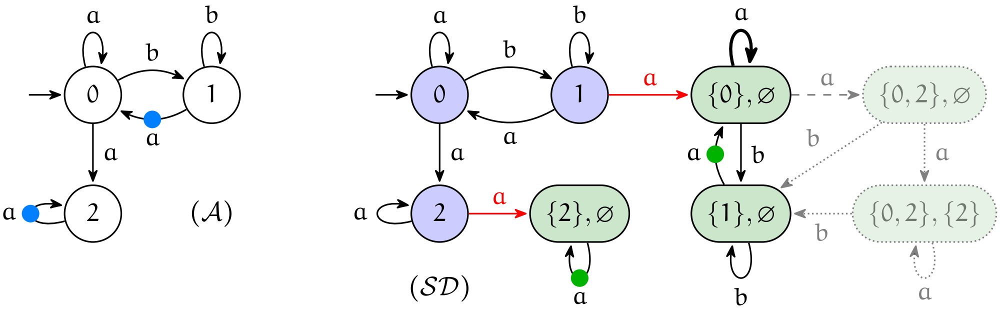


Figure 7.4: The same Büchi automaton $\mathcal{A}$ as in Figure 7.3 (left) and the corresponding semi-deterministic $\mathcal{SD}$ built using the SCC-aware optimization (right). In comparison to Figure 7.3, the dashed transition was replaced by the thick one due to the SCC-aware optimization. The dotted states and transitions became unreachable.

Correctness. With this modification, the proof of Lemma 7.3 remains unchanged. For the proof of Lemma 7.2 we only need to adjust the definition of k: *Let* C *be the SCC such that the recurrent transitions of* σ *are between states of* C. *Then let* k *be the minimal index such that it satisfies Lemma 7.1 for* σ, $(s_k, u_k, s_{k+1}) \in \mu(●)$, *and* $s_{k+1} \in C$. The rest of the proof remains the same.

7.5 CUT-DETERMINIZATION OF BÜCHI AUTOMATA

We can easily modify the above construction to build an automaton that is cut-deterministic and equivalent to $\mathcal{A}$. We define the automaton as $\mathcal{CD} = (Q', \Sigma, \delta_{\mathcal{CD}}, q'_I, \{\bullet\}, \mu_{\mathcal{SD}}, \mathsf{Inf}\bullet)$ with $Q' = Q'_N \cup Q_D$ and $\delta_{\mathcal{CD}} = \delta'_N \cup \delta'_t \cup \delta_D$ where Q_D, δ_D, and $\mu_{\mathcal{SD}}$ have the same meaning as before. On top of semi-determinization, we determinize the first part (N) of $\mathcal{SD}$ using the subset construction.

$$Q'_N \subseteq 2^S \qquad q'_I = \{s_I\} \qquad \delta'_N = \big\{\big(P, a, \tau_\delta(P, a)\big) \mid P \in Q'_N\big\}$$

For each marked transition of the form (p, a, s) of $\mathcal{A}$ and for each P such that $p \in P$ we have a cut transition $\big(P, a, (\{s\}, \varnothing)\big)$.

$$\delta'_t = \big\{\big(P, a, (\{s\}, \varnothing)\big) \mid s \in \overset{\bullet}{\tau}(P, a)\big\}$$

Figure 7.5 shows the cut-determinization of our example automaton $\mathcal{A}$. The determinization of the first component did not increase the number of states in our example, but in general, this is not always the case. Also, $\mathcal{CD}$ can have more cut transitions then the equivalent $\mathcal{SD}$.

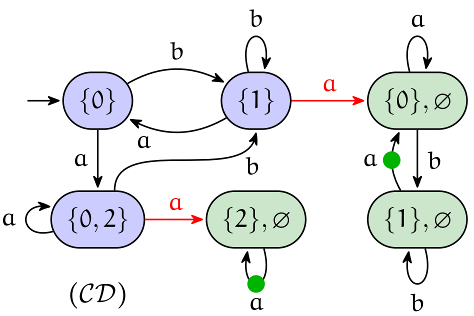


Figure 7.5: A cut-deterministic automaton $\mathcal{CD}$ equivalent to $\mathcal{A}$ from Figures 7.3 and 7.4. It is basically $\mathcal{SD}$ with a subset construction applied on the first (blue) part.

Complexity and correctness. The size of Q' is bounded by $|Q'| \le 2^{|S|} + 3^{|S|}$. For the proof of $L(\mathcal{A}) \subseteq L(\mathcal{CD})$ we can reuse the proof of Lemma 7.2 (with SCC-aware optimization). We need to change s_j by P_j in the definition of t'_j for $j \le k$ and argue that $s_k \in P_k$, which follows from definition of τ_δ. We can similarly adjust the proof of Lemma 7.3 to show that $L(\mathcal{CD}) \subseteq L(\mathcal{A})$. This is summarised in the following theorem.

Theorem 7.5. *For each Büchi automaton $\mathcal{A}$ with n states and with marks on transitions we can construct a cut-deterministic Büchi automaton $\mathcal{CD}$ with at most $2^n + 3^n$ states such that $L(\mathcal{CD}) = L(\mathcal{A})$.*

7.6 SEMI-DETERMINIZATION OF GENERALIZED BÜCHI AUTOMATA

The simplest approach to semi-determinization and cut-determinization of nondeterministic generalized Büchi automata proceeds in two steps. A given NGBA $\mathcal{A}$ with n states and marks ●$^{0}, \dots,$ ●^{h} is first converted to a nondeterministic Büchi automaton $\mathcal{B}$ with at most $n \cdot (h+1)$ states and one mark; we call this step *degeneralization*. As the second step, the NBA $\mathcal{B}$ is semi- or cut-determinized by the above construction.

Degeneralization. The degeneralization of $\mathcal{A}$ is straightforward. The automaton $\mathcal{B}$ is formed by $h+1$ copies of the semiautomaton of $\mathcal{A}$, called levels $0, \dots, h$. On level l, $\mathcal{B}$ waits for transitions marked by ●^{l} in $\mathcal{A}$, these transitions go in $\mathcal{B}$ to level $l+1$ for $l < h$ and to level 0 for $l = h$. The level l thus records that marks ●$^{0}, \dots,$ ●$^{l-1}$ have been seen since the last visit of level 0. The transitions from level h to 0 are marked by ●. Figure 7.6 shows de-generalization of a small GBA.

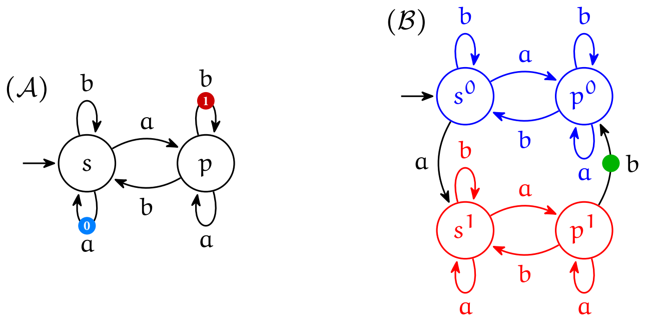
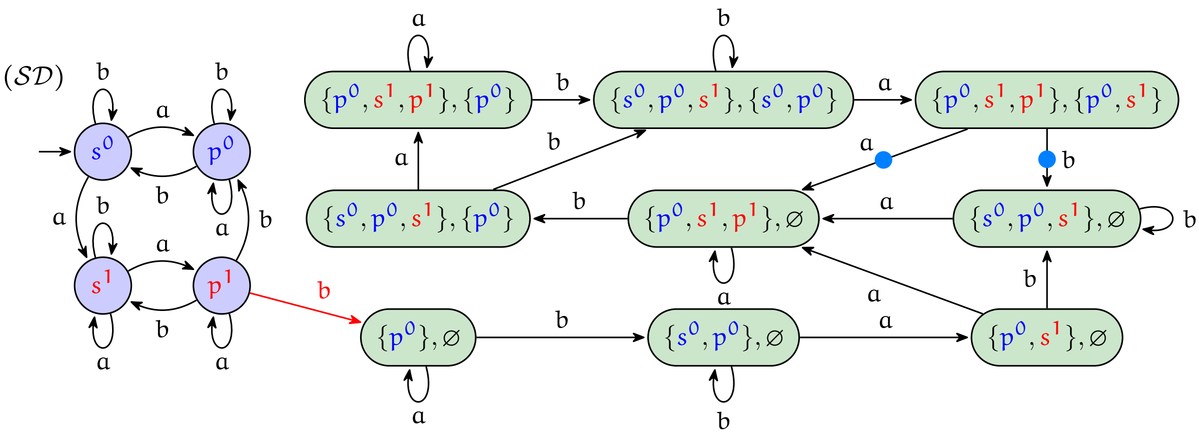


Figure 7.6: A GBA $\mathcal{A}$ and an equivalent BA $\mathcal{B}$ that was built using the standard degeneralization procedure.

Figure 7.7 shows that the two-step approach may not be the best. Consider the state $(R, B) = (\{s^0, p^0, s^1\}, \{s^0, p^0\})$ and relate it all the way back to $\mathcal{A}$. The state s of $\mathcal{A}$ is present in two variants in R — on level 0 and on level 1. Moreover, as both s^0 and p^0 are present in B, we know that both s and p are reachable by runs that crossed a ● since the last accepting transition of $\mathcal{SD}$. But $R \neq B$ because we also follow the runs that stayed behind on level 1.



($\mathcal{SD}$)
s^0
p^0
s^1
p^1
$\{p^0, s^1, p^1\}, \{p^0\}$
$\{s^0, p^0, s^1\}, \{s^0, p^0\}$
$\{p^0, s^1, p^1\}, \{p^0, s^1\}$
$\{s^0, p^0, s^1\}, \{p^0\}$
$\{p^0, s^1, p^1\}, \varnothing$
$\{s^0, p^0, s^1\}, \varnothing$
$\{p^0\}, \varnothing$
$\{s^0, p^0\}, \varnothing$
$\{p^0, s^1\}, \varnothing$
a
b

Figure 7.7: Two-step semi-determinization of $\mathcal{B}$ from Figure 7.6.

One-step semi-determinization of GBA. A direct semi-determinization of GBA was first presented by Hahn et al. in 2015.[5] It combines the level approach of degeneralization with semi-determinization and it always keeps at most one copy of a state s of the TGBA in R.

Let $\mathcal{A} = (S, \Sigma, \delta, s_I, M, \mu, \Phi)$ be a GBA with marks on transitions where $M = \{\bullet^0, \dots, \bullet^h\}$ for some $h \geq 0$ and $\Phi = \bigwedge_{0 \leq l \leq h} \mathsf{Inf}\bullet^l$. In the following, we present a SCC-aware one-step construction of an equivalent semi-deterministic automaton $\mathcal{GSD} = (Q, \Sigma, \delta_{\mathcal{GSD}}, q_I, \{\bullet\}, \mu_{\mathcal{GSD}}, \mathsf{Inf}\bullet)$ where $Q = Q_N \cup Q_D$ and $\delta_{\mathcal{GSD}} = \delta_N \cup \delta_c \cup \delta_D$. In addition to the basic semi-determinization, we augment the states of Q_D by levels, one for each accepting mark.

$$Q_N = S \quad q_I = s_I \quad \delta_N = \delta \quad Q_D \subseteq (2^S \times 2^S \times \{0, \dots, h\})$$

We add the cut transitions only for transitions marked by $\bullet^h$ in $\mathcal{A}$ and they lead to level 0.

$$\delta_c = \left\{\left(s_1, a, (\{s_2\}, \varnothing, 0)\right) \mid s_2 \in \overset{\bullet^h}{\tau}(\{s_1\}, a)\right\}$$

In Q_D we move runs to B only after the mark for the current level was seen.

$$\begin{aligned}\delta_1 = \Big\{\big((R_1, B_1, l), a, (R_2, B_2, l)\big) \mid{} & a \in \Sigma, 0 \leq l \leq h, \\ & B_1 \subset R_1 \subseteq C, C \text{ is a SCC of } \mathcal{A}, \\ & R_2 = \tau(R_1, a) \cap C, R_2 \neq \varnothing, \text{ and} \\ & B_2 = \big(\tau(B_1, a) \cup \overset{\bullet^l}{\tau}(R_1, a)\big) \cap C\Big\}\end{aligned}$$

For transitions of δ_1 where $R_2 = B_2$ we move to the next level $l' = (l + 1) \bmod (h + 1)$, reset B_2 and immediately start tracking $\bullet^{l'}$ there, and finally mark the transition by $\bullet$.

$$\begin{aligned}\delta_2 = \Big\{\big((R_1, B_1, l), a, (R_2, B_2', l')\big) \mid{} & \big((R_1, B_1, l), a, (R_2, B_2, l)\big) \in \delta_1, \\ & R_2 = B_2, \\ & B_2' = \overset{\bullet^{l'}}{\tau}(R_1, a) \cap R_2, \text{ and} \\ & l' = (l + 1) \bmod (h + 1)\Big\}\end{aligned}$$

$$\delta_D = \Big\{\big((R_1, B_1, l), a, (R_2, B_2, l)\big) \in \delta_1 \mid R_2 \neq B_2\Big\} \cup \delta_2$$

$$\mu_{\mathcal{GSD}}(\bullet) = \delta_2$$

The $\cap R_2$ in definition of B_2' ensures that B_2' contains only states from the same SCC as are in R_2.

You can find the result of this construction applied on $\mathcal{A}$ from Figure 7.6 in Figure 7.8. The resulting automaton $\mathcal{GSD}$ has $2 + 5$ states in comparison to $4 + 9$ states of $\mathcal{SD}$ from Figure 7.7.

Cut-determinization. We naturally modify the above construction to build an automaton that is cut-deterministic and equivalent to $\mathcal{A}$. We define the automaton as $\mathcal{GCD} = (Q', \Sigma, \delta_{\mathcal{GCD}}, q_I', \{\bullet\}, \mu_{\mathcal{GSD}}, \mathsf{Inf}\bullet)$ with $Q' = Q_N' \cup Q_D$ and $\delta_{\mathcal{GCD}} = \delta_N' \cup \delta_t' \cup \delta_D$ where Q_D, δ_D, and $\mu_{\mathcal{GSD}}$ have the same meaning as above. On top of semi-determinization, we determinize the first part (N) of $\mathcal{GSD}$ using the subset construction.

$$Q_N' \subseteq 2^S \quad q_I' = \{s_I\} \quad \delta_N' = \big\{\big(P, a, \tau_\delta(P, a)\big) \mid P \in Q_N'\big\}$$

[5] Hahn et al. (2015), "Lazy Probabilistic Model Checking without Determinisation", [10].

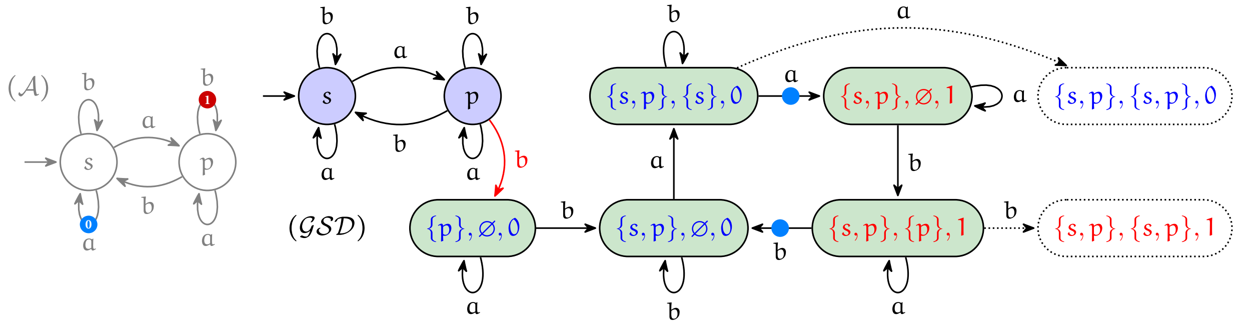


Figure 7.8: A semi-deterministic Büchi automaton $\mathcal{GSD}$ equivalent to the GBA $\mathcal{A}$ (from Figure 7.6). The transitions from $\delta_1 \setminus \delta_D$ and the tuples that would become reachable by those transitions are dotted and are not in fact parts of $\mathcal{GSD}$. The input automaton is in gray.

For each transition of the form (p, a, s) marked by ●^{h} in $\mathcal{A}$ and for each P such that $p \in P$ we have a cut transition $(P, a, (\{s\}, \varnothing, 0))$ to level 0.

$$\delta_c = \left\{ \big(P, a, (\{s\}, \varnothing, 0)\big) \mid s \in \overset{\bullet^{h}}{\tau}(P, a) \right\}$$

Complexity. In the two-step approach, the degeneralization step first creates a Büchi automaton $\mathcal{B}$ with $|M| \cdot |S|$ states. Then the size of Q is bounded by $|Q| \leq |M||S| + 3^{|M||S|}$ for semi-determinization and by $|Q| \leq 2^{|M||S|} + 3^{|M||S|}$ for cut-determinization. The one-step approach to semi-determinization of TGBA augments the states from Q_D of the semi-determinization of BA by $|M|$ levels; therefore, the bounds are $|Q'| \leq |S| + |M| \cdot 3^{|S|}$ for semi-determinization and $|Q'| \leq 2^{|S|} + |M| \cdot 3^{|S|}$ for cut-determinization.

$|M|$ is the number of marks of $\mathcal{A}$.

Correctness. The proof of Lemma 7.2 naturally extends to the one-step construction for TGBA. In the definition of σ' we now have levels in states of $\mathcal{GSD}$, that is $t'_j = \big(s_k, u_j, (\{s_{k+1}\}, \varnothing, 0)\big)$ for $j = k$ and for $j > k$ we use $t'_j = \big((R_j, B_j, l_j), u_j, (R_{j+1}, B_{j+1}, l_{j+1})\big)$. Further, we slightly modify the condition for the index n in the proof: *Let $n > f$ be the minimal index such that (s_n, u_n, s_{n+1}) is a transition from σ marked by* ●^{l_f}. The rest of the proof remains unchanged.

Note that $l_j = l_f$ for all $f \leq j \leq n$.

The modification of the proof of Lemma 7.3 is even more straightforward. For k we demand that (s_k, u_k, s_{k+1}) is marked by ●^{h} instead of ●. Further, we have to consider the levels. They change only at breakpoints and in a sequence of $h + 2$ consecutive breakpoints we can see levels for all marks of $\mathcal{A}$. On level l we follow ●^{l}, thus we have an infinite path in $G_u^{\mathcal{A}}$ with infinitely many edges from $\overline{\mu}($●$^{l})$ for each $0 \leq l \leq h$. This path represents an accepting run of $\mathcal{A}$ over u. Now we can conclude this section with the following theorem.

Theorem 7.6. *For each generalized Büchi automaton $\mathcal{A}$ with n states and h marks we can construct a semi-deterministic Büchi automaton $\mathcal{GSD}$ with at most $n + h \cdot 3^n$ states and a cut-deterministic Büchi automaton $\mathcal{GCD}$ with at most $2^n + h \cdot 3^n$ states such that* $L(\mathcal{GSD}) = L(\mathcal{GCD}) = L(\mathcal{A})$.

7.7 CUT-DETERMINIZATION OF SEMI-DETERMINISTIC GENERALIZED BÜCHI AUTOMATA

Blindly converting a semi-deterministic GBA $\mathcal{A} = (S, \Sigma, \delta, s_I, M, \mu, \Phi)$ into cut-deterministic BA using the full algorithm of the previous section is wasted effort. Indeed, we can create an equivalent cut-deterministic GBA $\mathcal{CD} = (Q, \Sigma, \delta_{\mathcal{CD}}, q_I, M, \mu, \Phi)$ with $Q = Q_N \cup Q_D$ and $\delta_{\mathcal{CD}} = \delta'_N \cup \delta'_t \cup \delta'_D$ in four steps as follows.

(i) We apply the algorithm described in Section 7.2 to partition S and δ into S_N, S_D, δ_N, δ_c, and δ_D in a way that S_D is maximal such that the partition witnesses the semi-determinism of $\mathcal{A}$. (ii) The deterministic part of $\mathcal{A}$ remains unchanged.

$$Q_D = S_D \qquad \delta'_D = \delta_D$$

(iii) We determinize S_N and δ_N using the subset construction.

$$Q_N \subseteq 2^{S_N} \quad q_I = \{s_I\} \quad \delta'_N = \left\{(P, a, P') \mid P \in Q_N \text{ and } P' = \tau_{\delta_N}(P, a)\right\}$$

Finally, (iv) we revise δ_c accordingly.

$$\delta'_t = \left\{(P, a, s) \mid s \in \tau_{\delta_c}(P, a)\right\}$$

Note that we use the generalized Büchi acceptance condition of $\mathcal{A}$ also for $\mathcal{CD}$ here, while the cut-determinization of general (not semi-deterministic) GBA presented in Section 7.6 always produces a Büchi automaton $\mathcal{GCD}$.

Complexity and correctness. The size of Q is bounded by $|Q| \le 2^{S_N} + |S_D|$ while using the algorithm of Section 7.6 yields $|Q| \le 2^{|S|} + |M| \cdot 3^{|S|}$. The correctness of the algorithm follows from the facts that the marks are placed only on elements of S_D which remained unchanged and that the subset construction exactly follows the runs of the original automaton.

7.8 IMPLEMENTATION

All algorithms described in this chapter were implemented in our tool called *Seminator*. Since version 1.2.0, Seminator uses the SCC-aware algorithms. The tool is implemented in C++ using the Spot library[6] and is distributed under the *GNU GPL v3* license. The source code, basic installation and usage instructions, and an evaluation of the tool can be found on the Seminator's web page, see Table 7.1. For reading and writing automata Seminator uses the *Hanoi Omega-Automata (HOA)* format.[7]

The tool takes a TGBA $\mathcal{A}$ as input and prints a semi-deterministic (default) or cut-deterministic automaton $\mathcal{B}$ such that $L(\mathcal{A}) = L(\mathcal{B})$ as output. Seminator does not modify $\mathcal{A}$ if it already complies with the requested type. In such cases, only the simplifications offered by Spot are applied. For cut-determinization of semi-deterministic automata Seminator uses the algorithm from Section 7.7. Moreover, before testing the input automaton for semi- or cut-determinism, we apply the following language-preserving modification of $\mathcal{A}$: *We remove all marks from transitions that are not inside any accepting SCC.* This modification itself can transform an automaton which is not semi-deterministic into a semi-deterministic one (the same holds for cut-determinism) and even if it is not

An SCC C is accepting if each mark of $\mathcal{A}$ marks a transition inside C.

[6] Duret-Lutz et al. (2016), "Spot 2.0 - A Framework for LTL and ω-Automata Manipulation", [55].

[7] Babiak et al. (2015), "The Hanoi Omega-Automata Format", [17].

the case, it can reduce the size of resulting automaton as smaller part of the original automaton has to be determinized.

The size of $\mathcal{B}$ can often be further improved by the automata reduction techniques that are implemented in Spot. They all preserve semi-determinism of TGBAs. However, the reverse simulation technique[8] does not preserve cut-determinism and thus it is not applied if a cut-deterministic automaton is requested.

By default, Seminator outputs GBA, however, BA can be requested. Note that semi-determinization and cut-determinization of automata that are not semi-deterministic always produce a BA. Seminator can produce a GBA with two or more accepting sets only when $\mathcal{A}$ is already at least semi-deterministic.

Degeneralization modes. In contrast to our expectations, for some GBA the two-step semi-determinization outperforms the one-step semi-determinization. In the two-step semi-determinization, $\mathcal{A}$ is first degeneralized into a BA and then the standard semi-determinization for BA is applied. The one-step approach performs degeneralization simultaneously with the semi-determinization. See Sections 7.4 and 7.6. The two-step approach benefits from a highly optimized degeneralization of Spot; none of the available optimizations is performed in the one-step approach. Therefore, by default Seminator tries the following three modes for dealing with degeneralization, compares the resulting automata sizes, and returns the smallest automaton out of the three.[9]

1. Convert the input GBA $\mathcal{A}$ directly.
2. Transform $\mathcal{A}$ into an equivalent BA with marks on transitions and then perform the conversion.
3. Transform $\mathcal{A}$ into an equivalent BA with marks on states and then perform the conversion.

[8] Babiak et al. (2013), “Compositional Approach to Suspension and Other Improvements to LTL Translation”, [44].

[9] Users can force to use one of the three modes.

7.9 EXPERIMENTAL EVALUATION

In this section, we compare the number of states of automata produced by Seminator to the sizes of automata produced by other tools that are able to produce semi-deterministic or cut-deterministic automata.

Tools and hardware. To our best knowledge, there are only three other relevant tools: nba2ldba, ltl2ldba, and `Büchifier`. The tool nba2ldba converts BA into semi-deterministic BA[10] and ltl2ldba translates LTL formulae directly to semi-deterministic or cut-deterministic TGBA.[11] Both tools are distributed as parts of the Owl library. `Büchifier` translates a fragment of LTL into semi-deterministic automata with a single-exponential blow-up.[12] However, we do not include it in our experiments for three reasons: (i) it works only for a fragment of LTL, (ii) it does not exhibit promising results in the authors' evaluation, and (iii) it is available only for Windows.

We use two basic configurations of Seminator in our evaluation: one is Seminator in the default setting and the other performs the two-step semi-determinization of TGBA (referenced as 2-step, uses option `--via-tba`).

Seminator simplifies the resulting automata by calling the reduction techniques of Spot. These reductions can have a strong effect on the size of the automata and thus camouflage potential weaknesses of the approach. As we aim to evaluate different approaches rather than comparing different tools, we evaluate each tool with (yes) and without (no) these reductions; we use Spot's `autfilt --small` command to apply the reduction on products of the tools from the Owl library and we use the option `-s0` to disable the reductions in Seminator.

Because ltl2ldba needs an LTL formula as input, our evaluation starts with LTL formulae. For Seminator and nba2ldba, we translate the e formulae to automata of the expected type using Spot's `ltl2tgba -D` command. The option `-D` expresses a preference towards more deterministic output.

The homepages and versions of all tools that were used for this evaluation are listed in Table 7.1. Overall, we have eight configurations of tools that produce semi-deterministic automata, see Table 7.2 for the precise commands, and six configurations that produce cut-deterministic automata, see Table 7.3. The evaluation ran on a laptop with Intel Core i7-2620M (2.70 GHz) processor and 8GB RAM. All toolchains finished the computation for all formula within one minute.

Benchmark formulae. We use two benchmark sets of LTL formulae. The first set of formulae collected from the literature was already used in Chapter 6. Figure 7.9 shows that it is very often the case that `ltl2tgba -D` produces a deterministic TGBA (●), or a non-deterministic TGBA that is already cut-

[10] It basically performs the two-step semi-determinization without the SCC-aware optimization and with marks on states.

[11] Sickert et al. (2016), "Limit-Deterministic Büchi Automata for Linear Temporal Logic", [62].

[12] Kini and Viswanathan (2017), "Optimal Translation of LTL to Limit Deterministic Automata", [69].

Table 7.1: Tools used in the experimental evaluation.

tools	version	webpage
Seminator	1.2.0	https://github.com/mklokocka/seminator/
ltl2ldba, nba2ldba	1.1.0	https://www7.in.tum.de/~sickert/projects/owl/
ltl2tgba, autfilt, ltlfilt	2.4.2	https://spot.lrde.epita.fr/

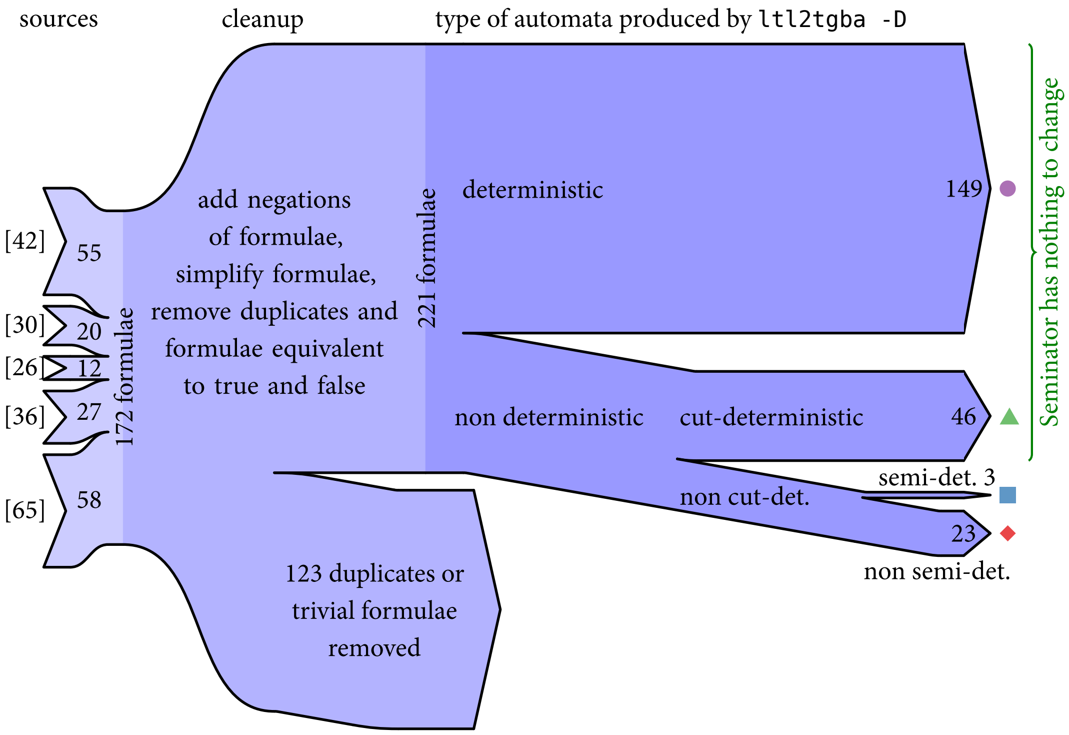

sources
cleanup
type of automata produced by ltl2tgba -D
[42]
55
[30]
20
[26]
12
[36]
27
[65]
58
172 formulae
add negations of formulae, simplify formulae, remove duplicates and formulae equivalent to true and false
221 formulae
deterministic
149
non deterministic
cut-deterministic
46
non cut-det.
semi-det. 3
23
non semi-det.
123 duplicates or trivial formulae removed
Seminator has nothing to change

Figure 7.9: Preparation of the formulae from the literature, and classification according to the four types of automata produced by `ltl2tgba -D`.

Table 7.2: Tool configurations for generating a semi-deterministic automaton from formula φ.

approach	reductions	command line
Seminator	no	`ltl2tgba -D φ | seminator -s0`
	yes	`ltl2tgba -D φ | seminator`
Seminator 2-step	no	`ltl2tgba -D φ | seminator --via-tba -s0`
	yes	`ltl2tgba -D φ | seminator --via-tba`
ltl2ldba	no	`ltl2ldba -n φ`
	yes	`ltl2ldba -n φ | autfilt --small --tgba`
nba2ldba	no	`ltl2tgba -D φ --ba | nba2ldba`
	yes	`ltl2tgba -D φ --ba | nba2ldba | autfilt --small --tgba`

Table 7.3: Tool configurations for generating cut-deterministic automata. (The `autfilt` invocation has extra options to disable reverse-simulation based reductions, since those do not preserve cut-determinism.)

approach	reductions	command line
Seminator	no	`ltl2tgba -D φ | seminator --cd -s0`
	yes	`ltl2tgba -D φ | seminator --cd`
Seminator 2-step	no	`ltl2tgba -D φ | seminator --cd --via-tba -s0`
	yes	`ltl2tgba -D φ | seminator --cd --via-tba`
ltl2ldba	no	`ltl2ldba -n φ`
	yes	`ltl2ldba -n φ | autfilt --small -x simul=1,ba-simul=1 --tgba`

deterministic (▲). Depending on its configuration, Seminator only has to perform some work on automata that are not cut-deterministic (■ and ◆) or on automata that are not semi-deterministic (◆).

As Seminator actually has to do some work only on a few formulae from the previous set, we also use a set of formulae generated randomly. For each of the four types of `ltl2tgba -D` output (●, ▲, ■, ◆) we generated exactly 100 formulae. The command used to generate all formulae used in this evaluation can be found in the GitHub repository of Seminator.

7.9.1 Results and Observations

Table 7.4 compares the sizes (number of states) of the semi-deterministic automata produced by Seminator, ltl2ldba, and nba2ldba in the configurations given in Table 7.2. Similarly, Table 7.5 compares the sizes of the cut-deterministic automata produced by Seminator and ltl2ldba.

Table 7.4: Evaluation of the tools producing semi-deterministic automata, on random LTL formulae and LTL formulae from literature classified according the type of automata produced by `ltl2tgba -D`. Each cell presents the cummulative size (number of states) of semi-deterministic automata produced by the corresponding tool without ('no') or with ('yes') reductions for the corresponding set of n formulae.

formulae			Seminator		Seminator 2-step		ltl2ldba		nba2ldba	
origin	type	n	no	yes	no	yes	no	yes	no	yes
random	● det	100	415	415	416	416	639	445	428	428
	▲ cd	100	463	463	463	463	733	539	863	634
	■ sd	100	704	704	705	705	1228	784	850	774
	◆ nd	100	1233	937	1276	987	1314	804	3657	1876
literature	● det	149	556	556	585	585	1277	855	600	600
	▲ cd	46	194	194	198	198	838	341	377	240
	■ sd	3	13	13	13	13	41	17	17	13
	◆ nd	23	472	369	514	404	666	376	869	573

Table 7.5: Evaluation of the tools producing cut-deterministic automata, on random LTL formulae and LTL formulae from literature classified according the type of automata produced by `ltl2tgba -D`. Each cell presents the cummulative size (number of states) of cut-deterministic automata produced by the corresponding tool without ('no') or with ('yes') reductions for the corresponding set of n formulae.

formulae			Seminator		Seminator 2-step		ltl2ldba	
origin	type	n	no	yes	no	yes	no	yes
random	● det	100	415	415	416	416	570	497
	▲ cd	100	463	463	463	463	732	649
	■ sd	100	734	712	735	713	1495	1275
	◆ nd	100	2028	1141	2106	1184	1387	1038
literature	● det	149	556	556	585	585	1039	809
	▲ cd	46	194	194	198	198	612	488
	■ sd	3	13	13	13	13	53	40
	◆ nd	23	656	414	698	450	470	410

Further, Figure 7.10 provides a comparison of Seminator and ltl2ldba on the level of individual semi-deterministic or cut-deterministic automata produced for the considered formulae. Both tools run without reductions to expose the difference in the core algorithms of the tools. Finally, Figure 7.11 compares the semi-deterministic automata produced by Seminator to those produced by nba2ldba. Again, both tools run without reductions. The last figure, Figure 7.12 compares Seminator with Seminator 2-step in a similar manner.

The presented results immediately lead to several observations.

1. Seminator produces nearly always the smallest semi-deterministic or cut-deterministic automaton if it gets as input a TGBA that is already semi-deterministic (which includes deterministic and cut-deterministic automata as well). Note that Seminator does not change such automata at all unless a cut-deterministic automaton is required and it gets a semi-deterministic automaton that is not cut-deterministic. In this case, Seminator just applies the subset construction on the nondeterministic part of the automaton. Hence, all these results reflect the efficiency of Spot's LTL to TGBA translation and not the efficiency of the Seminator's core algorithm.

2. When Seminator gets a TGBA that is not semi-deterministic, it produces a bigger cut-deterministic automaton than the one produced by ltl2ldba directly from the formula in many cases. When semi-deterministic automata are produced, the situation is different and it is difficult to predict which tool would produce a smaller automaton. Note that Seminator always produces a TBA in these cases, while ltl2ldba produces a TGBA.

3. The advantage of Seminator over Seminator 2-step is not very dramatic. The reasons were already touched on in the previous section.

4. Seminator clearly outperforms nba2ldba on all sets of benchmarks.

5. The numbers in Tables 7.4 and 7.5 show that reductions can save many states in the semi-deterministic and cut-deterministic automata produced by Seminator, ltl2ldba, or nba2ldba.

6. The semi-deterministic automata produced by ltl2ldba can be larger than the cut-deterministic automata produced by the same tool. This is unexpected and it indicates a potential for further improvement of the tool.

The experimental evaluation brought two main outputs. First, if someone needs to translate an LTL formula to a small semi-deterministic automaton, it pays to try to translate it by Spot. If Spot produces a semi-deterministic automaton, it is very probably smaller than what ltl2ldba would produce. The same holds when a cut-deterministic automaton is needed, but it may be necessary to run Seminator to cut-determinize the semi-deterministic automaton produced by Spot. Second, if someone needs a semi-deterministic automaton from a nondeterministic automaton rather than from an LTL formula, Seminator should be used instead of nba2ldba.

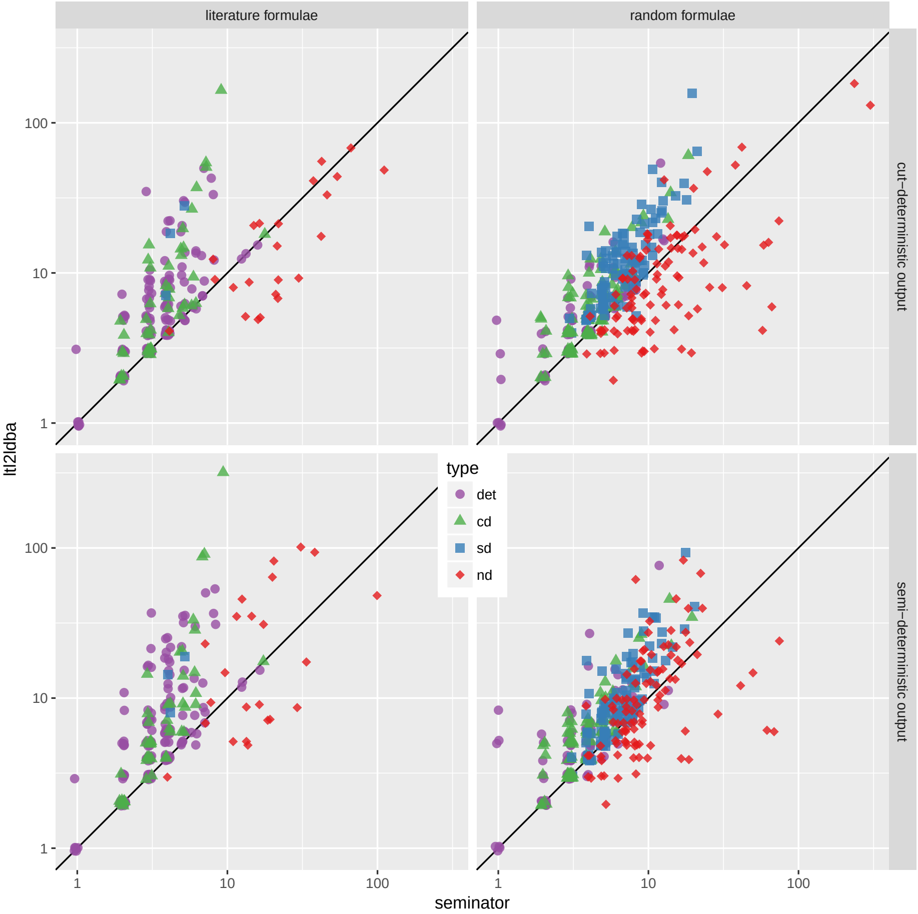


Figure 7.10: Comparison of the size of cut-deterministic automata produced by Seminator and ltl2ldba (both without reductions) on random formulae and on formulae from literature, and the analogous comparison of produced semi-deterministic automata. Scatter plots are colored according to the output type of `ltl2tgba -D`. **Note log scale.**

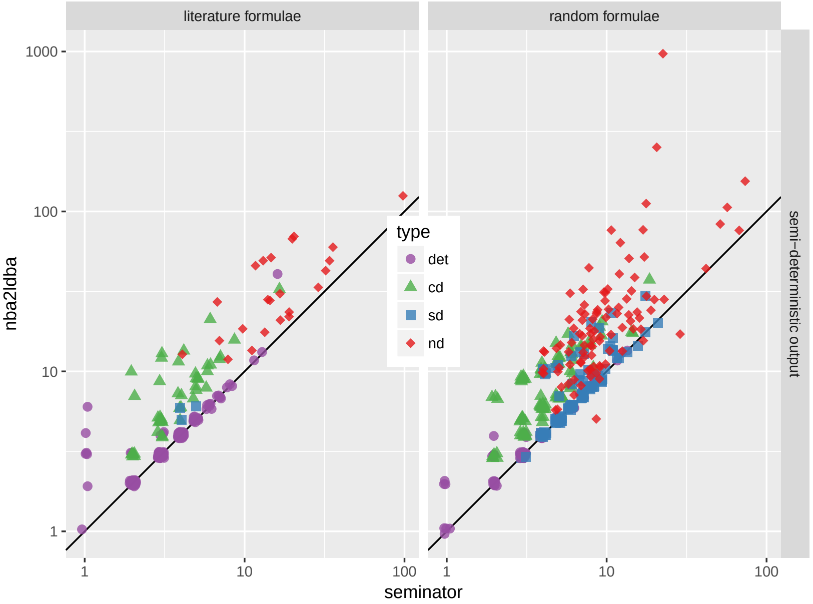


Figure 7.11: Comparison of the size of semi-deterministic automata produced by Seminator and nba2ldba (both without reductions) on random formulae and on formulae from literature. Scatter plots are colored according to the output type of `ltl2tgba -D`. **Note log scale.**

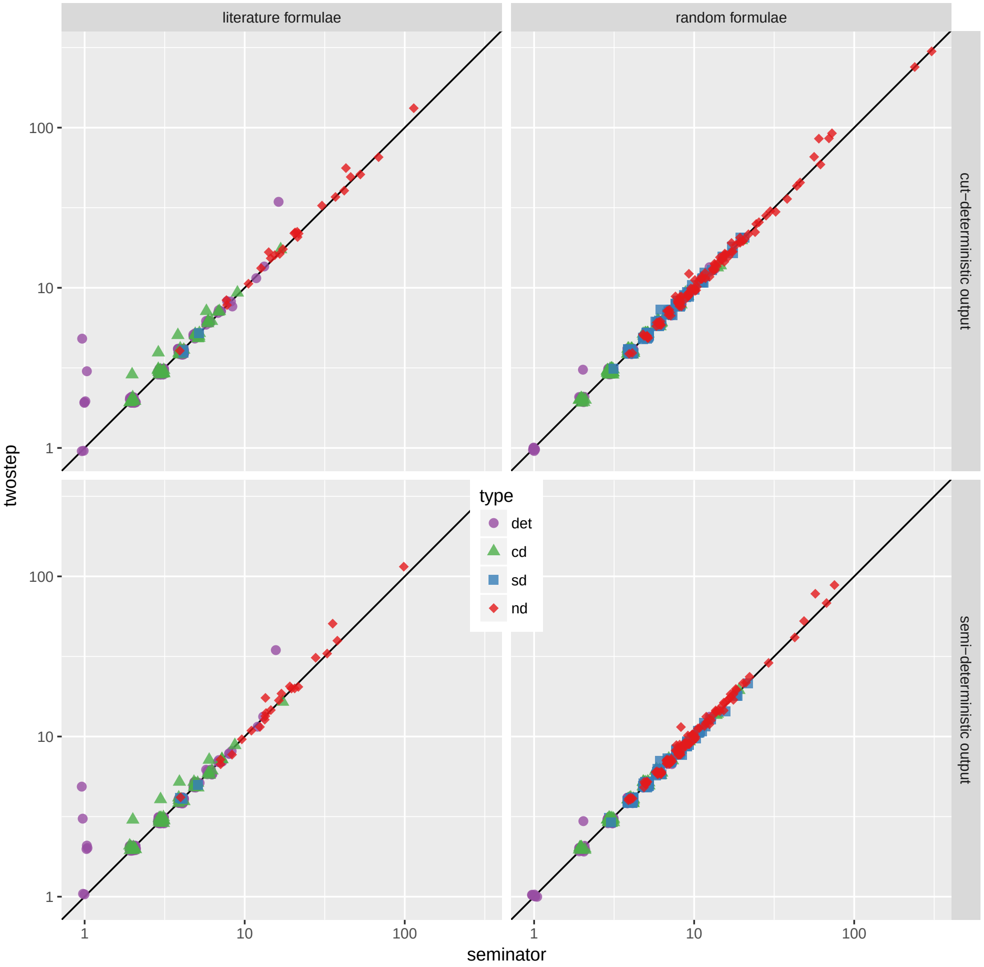


Figure 7.12: Comparison of the size of cut-deterministic automata produced by Seminator and Seminator 2-step (both without reductions) on random formulae and on formulae from literature, and the analogous comparison of produced semi-deterministic automata. Scatter plots are colored according to the output type of `ltl2tgba -D`. **Note log scale.**

8 *Complementation of Semi-Deterministic Büchi Automata*

In this chapter, we discuss a complementation procedure tailored for semi-deterministic Büchi automata. We start with a short discussion of complementation procedures for nondeterministic Büchi automata, then we discuss key observations about runs of semi-deterministic automata and use these observations to explain our construction and describe it formally. Then we reuse the notion of *ranks*[1] to prove the correctness of our algorithm, and finally, we describe our implementations, compare the performance of the construction to other algorithms that complement (nondeterministic) Büchi automata and discuss the impact of our construction on termination analysis.

Throughout this chapter, we work only with Büchi automata with marks on states (SBA). This allows us to omit acceptance labels in figures and use a condensed notation for runs in the form of an infinite sequence of states instead of transitions. As our main practical motivation for this research – termination analysis in ULTIMATE BÜCHI AUTOMIZER[2] – allows automata to have multiple initial states, in this chapter we consider a definition of automata where the initial state s_I is replaced by a set of initial states I.

Run of $\mathcal{A}$ now has to start in some initial state from I.

8.1 COMPLEMENTATION OF NBA

Complementation of a given NBA $\mathcal{A}$ over an alphabet Σ is a problem to create a Büchi automaton $\mathcal{C}$ over the same alphabet that recognizes the complement language of $\mathcal{A}$, which is $\Sigma^{\omega} \setminus L(\mathcal{A})$. It is a classic problem that has been extensively studied for more than half a century.[3] The known constructions for the complementation of NBA can be classified into the following four categories.

Ramsey-based. Historically the first complementation for NBA introduced by Büchi[4] and later improved by Sistla et al.[5] in which a Ramsey-based combinatorial argument is involved.

Determinization-based. A construction proposed by Safra[6] and enhanced by Piterman[7] in which a state of a complement is represented by a Safra tree.

Rank-based. A construction introduced by Kupferman and Vardi[8] based on ranks of run graphs for which several optimizations[9] have been proposed.

Slice-based. A construction proposed by Kähler and Wilke[10] based on reduced split trees.

The best known upper bounds on the size of the complement for these categories are in order $n^{\mathcal{O}(n)}$, $\mathcal{O}(n^{2n})$, $\mathcal{O}((0.76n)^n)$, and $\mathcal{O}((3n)^n)$ where n represents the number of states of the input NBA.

[1] Kupferman and Vardi (2001), "Weak Alternating Automata are not that Weak", [70].

[2] Heizmann, Hoenicke, and Podelski (2014), "Termination Analysis by Learning Terminating Programs", [8].

[3] See Vardi (2007), "The Büchi Complementation Saga", [71], for a survey.

[4] Büchi (1962), "On a Decision Method in Restricted Second Order Arithmetic", [1].

[5] Sistla, Vardi, and Wolper (1987), "The Complementation Problem for Büchi Automata with Appplications to Temporal Logic", [72].

[6] Safra (1988), "On the Complexity of Omega-Automata", [50].

[7] Piterman (2007), "From Nondeterministic Büchi and Streett Automata to Deterministic Parity Automata", [51].

[8] Kupferman and Vardi (2001), "Weak Alternating Automata are not that Weak", [70].

[9] Schewe (2009), [6]; Gurumurthy et al. (2003), [73]; Friedgut, Kupferman, and Vardi (2006), [74].

[10] Kähler and Wilke (2008), "Complementation, Disambiguation, and Determinization of Büchi Automata Unified", [75].

The upper bound for the rank-based complementation matches the lower bound for complementation of NBA proved by Yan[11]. Similarly as in the world of finite words, the complementation of deterministic Büchi automata is noticeably easier, it can be done with only linear blow-up. More precisely, for a DBA with n states from which a are accepting, we can build a complement DBA with $2n - a$ states.[12] Finally, complementation of semi-deterministic Büchi automata is somewhere in between – it is in $\mathcal{O}(2^n)$ as we will show in the following.

8.2 COMPLEMENTATION OF SDBA

Here we present *NCSB complementation of sDBA* which exploits the special structure of sDBA to achieve smaller complement automata. More precisely, if the deterministic part of the input sDBA $\mathcal{A}$ contains d states, including a accepting states and the nondeterministic part contains n states, the NCSB complementation produces a complement automaton $\mathcal{C}$ with at most $2^n 3^a 4^{d-a}$ states. Moreover, if $\mathcal{A}$ is deterministic ($n = 0$) $\mathcal{C}$ has $2d - a$ states, which meets the Kurshan's construction for the complementation of DBA.[13]

Besides the smaller theoretical size, the automaton $\mathcal{C}$ typically has a low degree of non-determinism when compared to results of other complementation algorithms, and is always unambiguous[14]. Moreover, the automata produced by our construction have a simple structure: they are merely an extended breakpoint construction[15] and thus are suitable for symbolic representation.

With all of these favourable properties in mind, it would be easy to think that the complementation mechanism we develop forms a class of its own. But this is not the case: the algorithm could be reformulated as an optimized version of the rank-based algorithm[16] tailored specially (and also correct only) for sDBA. However, we believe that the intuition is more clear if we focus on runs' properties rather than ranks.

Blocking. Our construction again relies on subset construction to follow all possible runs of the input automata. When handling automata that are not complete, the subset construction follows more than all runs – the set of states reached by the subset construction after reading a finite prefix of a word u may contain states that have no successors in the next step. Sequences of transitions ending in such states cannot be prolonged into infinite runs over u and we say that the corresponding runs *block*. The usage of the term *run* conflicts the fact that runs are infinite but we believe this notation simplifies the presentation.

Relation of runs to the complement. Let $\mathcal{A} = (Q, \Sigma, \delta, I, \{\bullet\}, \mu, \mathsf{Inf}\bullet)$ be an sDBA, $Q_N, \delta_N, Q_D, \delta_D, \delta_c$ be the notation introduced in Section 7.1, and $u = u_0 u_1 \ldots \in \Sigma^\omega$ be an infinite word. Each run σ of $\mathcal{A}$ over u has one of the following properties:

1. σ stays forever in Q_N,
2. σ enters Q_D and stops visiting ● at some point, or
3. σ is an accepting run.

[11] Yan (2008), "Lower Bounds for Complementation of Omega-Automata Via the Full Automata Technique", [7].

[12] Kurshan (1994), "The Complexity of Verification", [76].

[13] Kurshan (1994), "The Complexity of Verification", [76].

[14] For each $u \in L(\mathcal{C})$ there exists only one accepting run over u

[15] Miyano and Hayashi (1984), "Alternating Finite Automata on Omega-Words", [77].

[16] Kupferman and Vardi (2001), [70].

Clearly, $u \notin L(\mathcal{A})$ if and only if every run of $\mathcal{A}$ over u has one of the first two properties. In the second case, we say that σ is *safe* after visiting ● for the last time (or since the moment it enters Q_D if it does not visit any accepting state at all).

In order to check whether $u \in L(\mathcal{A})$ or not, one has to track all possible runs of $\mathcal{A}$. After reading a finite prefix of u, the states reached by the subset construction can be divided into three sets.

1. The set $N \subseteq Q_N$ represents the runs that kept out of the deterministic part so far.

N stands for *nondeterministic*

2. The set $C \subseteq Q_D$ represents the runs that have entered the deterministic part and that are not safe yet. One has to *check* if some of them will be prolonged into accepting runs in the future, or if all of the runs eventually block or become safe.

C stands for *check*

3. The set $S \subseteq (Q_D \setminus \mu(\bullet))$ represents the *safe* runs.

S stands for *safe*

Clearly, every accepting run of $\mathcal{A}$ stays in C after leaving N. On the other hand, if $w \notin L(\mathcal{A})$, every infinite run either stays in N or eventually leaves C to S and thus does not stay in C forever.

NCSB complementation of sDBA. Let $\mathcal{A} = (Q, \Sigma, \delta, I, \{\bullet\}, \mu, \mathsf{Inf}\bullet)$ be an sDBA with marks on states. The *NCSB complementation construction* creates an unambiguous Büchi automaton $\mathcal{C} = (P, \Sigma, \delta_{\mathcal{C}}, I_{\mathcal{C}}, \{\bullet\}, \mu_{\mathcal{C}}, \mathsf{Inf}\bullet)$ that recognizes the language $\Sigma^\omega \setminus L(\mathcal{A})$. The construction tracks runs of $\mathcal{A}$ using the powerset construction and guesses the right classification of runs into sets N,C, and S. Moreover, in order to check that no run stays forever in C, it uses one more set $B \subseteq C$. Therefore, states of $\mathcal{C}$ are quadruples (N, C, S, B) of subsets of Q — hence the name NCSB complementation construction.

$$P \subseteq \{(N, C, S, B) \mid N \subseteq Q_N, B \subseteq C \subseteq Q_D, S \subseteq Q_D \setminus \mu(\bullet), \text{ and } S \cap C = \emptyset\}$$

After reading only a finite prefix of the input word u, the automaton cannot know whether or not some run is already safe, as this depends on the suffix of u. The automaton $\mathcal{C}$ uses the guess-and-check strategy. Whenever a run σ in C may freshly become safe (it is leaving a marked state or it is entering Q_D via a cut transition $t \in \delta_c$), then the automaton $\mathcal{C}$ makes a nondeterministic decision to move σ to S or to leave it in C. The construction punishes every wrong decision:

- in order to preserve correctness, a run of $\mathcal{C}$ is blocked if σ is moved to S too early (runs in S are not allowed to visit marked states anymore), and

See the state $(\{0\},\{3\},\{\},\{3\})$ of Figure 8.1 and the letter b for an example.

- in order to maintain unambiguity, σ is allowed to move from C to S only when leaving a marked state. Hence, if σ misses the moment when it leaves a marked state for the last time, it will stay in C forever and this particular run of $\mathcal{C}$ will not be accepting.

See the state $(\{0\},\{2\},\{\},\{2\})$ of Figure 8.1 and the word a^ω for an example.

The set B mimics the behaviour of C with one exception: it does not adopt the runs freshly coming to C via δ_c. The size of B never increases until it becomes empty; then we say that a *breakpoint* is reached. After each breakpoint, B is set to track exactly the runs currently in C.

With the provided intuition in mind, we define the transitions of $\mathcal{C}$. We have $\big((N, C, S, B), a, (N', C', S', B')\big) \in \delta_{\mathcal{C}}$ iff

1. $N' = \tau_{\delta_N}(N, a)$ and $C' \cup S' = \tau_{\delta_c}(N, a) \cup \tau_{\delta_D}(C \cup S, a)$,

   tracing the reachable states correctly
2. $C' \cap S' = \varnothing$,

   a run in Q_D is either safe or not
3. $S' \supseteq \tau_{\delta_D}(S, a)$,

   safe runs must stay safe
4. $C' \supseteq \tau_{\delta_D}(C \smallsetminus \mu(\bullet), a)$,

   only runs leaving a marked state can be moved to S, the rest stays in C
5. for all $q \in C \smallsetminus \mu(\bullet)$ we have $\tau_{\delta_D}(\{q\}, a) \neq \varnothing$, and

   punish the wrong guess – the corresponding run should have been in S already
6. if $B = \varnothing$ then $B' = C'$ and otherwise $B' = \tau_{\delta_D}(B, a) \cap C'$.

   breakpoint construction to check that no run stays in C forever

Note that the only source of nondeterminism of $\delta_{\mathcal{C}}$ is when $\mathcal{C}$ has to guess correctly whether or not a run σ of $\mathcal{A}$ is safe. Such situations arise in two cases, namely when the current state q of the run σ satisfies

- $q \in \tau_{\delta_c}(N, a) \smallsetminus (\tau_{\delta_D}(S, a) \cup \mu(\bullet))$, and when

  σ is freshly entering Q_D
- $q \in \tau_{\delta_D}(C \cap \mu(\bullet), a) \smallsetminus (\tau_{\delta_D}(S, a) \cup \mu(\bullet))$.

  σ is leaving a marked state

All other situations are determined, including runs that are currently in a marked state (which belong to C) or in $\delta_D(S, a)$ (which belong to S).

No run stays in C if and only if B becomes empty over and over again. Thus the acceptance marks are placed on states with breakpoints.

$$\mu_{\mathcal{C}}(\bullet) = \{(N, C, S, B) \in P \mid B = \varnothing\}$$

Finally, the correct classification of runs from their start has to be reflected in the set of initial states. As some runs may start in the deterministic part of $\mathcal{A}$, we use the guess-and-check strategy again.

$$I_{\mathcal{C}} = \{(Q_N \cap I, C, S, C) \mid S \cup C = I \cap Q_D, S \cap C = \varnothing\}$$

We offer an example of the NCSB construction in Figure 8.1 where the automaton $\mathcal{C}$ with 7 states was built as a complement for the automaton $\mathcal{A}$ with 4 states. The numbers of states of complement automata built by the Ramsey-based, determinization-based, rank-based, and slice-based constructions are 54, 13, 13, and 10, respectively.

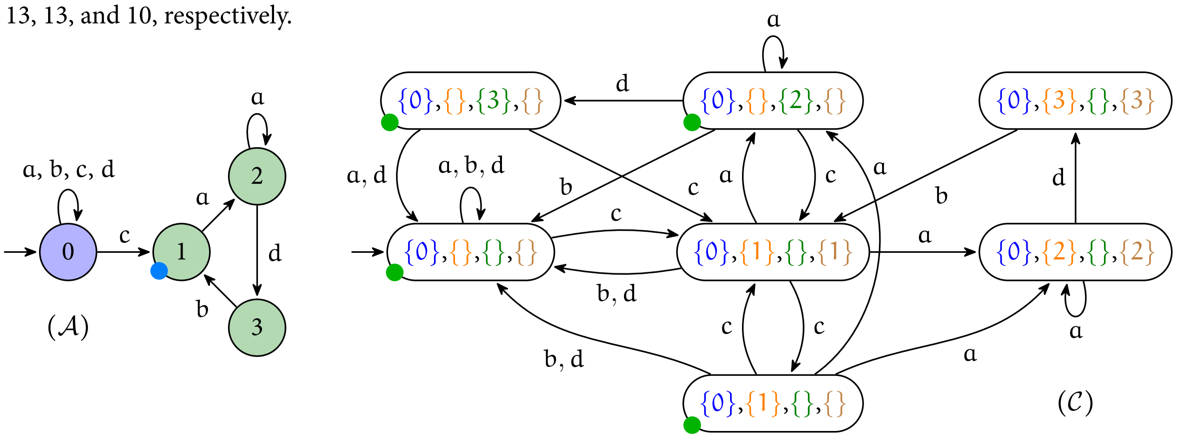


Figure 8.1: A Büchi automaton $\mathcal{A}$ (left) and its complement Büchi automaton $\mathcal{C}$ (right) built by the NCSB construction.

Complexity. Let $p = (N, C, S, B) \in P$ of $\mathcal{C}$. Then

- for a state $q_1 \in Q_N$ of $\mathcal{A}$, q_1 is either present or absent in N;

$2^{|Q_N|}$

- for $q_2 \in \mu(\bullet)$, one of the following three options holds: q_2 is only in C, q_2 is both in C and B, or q_2 is not present in p at all; and

$3^{|\mu(\bullet)|}$

- for $q_3 \in Q_D \setminus \mu(\bullet)$, one of the following four options holds: q_3 is only in S, q_3 is only in C, q_3 is both in C and B, or q_3 is not present in p at all.

$4^{|Q_D \setminus \mu(\bullet)|}$

The size of P is thus bounded by $|P| \leq 2^{|Q_N|} \cdot 3^{|\mu(\bullet)|} \cdot 4^{|Q_D \setminus \mu(\bullet)|}$.

As already mentioned before, for deterministic automata (here we assume $\mathcal{A}$ is complete and Q_N is empty), the NCSB construction leads to an automaton similar to an automaton with $2|Q| - |\mu(\bullet)|$ states produced by Kurshan's construction.[17] To see the size of the automaton produced by our construction for a DBA, recall that a state (N, C, S, B) of the complement automaton encodes that exactly the states in $N \cup C \cup S$ are reachable. For a DBA, $N \cup C \cup S$ thus contains exactly one state q of Q. Moreover, N is empty and thus B coincides with C since B becomes empty together with C. If $q \in \mu(\bullet)$, then it is in both B and C. If $q \in Q_D \setminus \mu(\bullet)$, then it is either only in S, or in both B and C, leading to a size $|P| \leq 2|Q_D| - |\mu(\bullet)|$.

8.3 RANKS AND CORRECTNESS

We open this section by recalling *run graphs* and introducing *ranks*. We then look at the NCSB construction through the ranking lense and use the insights this provides for proving correctness and unambiguity of the construction.

Run graphs. We have introduced run graphs in the previous section for automata with marks on transitions and with a unique initial state. Here we redefine it for automata with marks on states and with multiple initial states.

Let $\mathcal{A} = (Q, \Sigma, \delta, I, \{\bullet\}, \mu, \mathsf{Inf}\bullet)$ be an NSBA with multiple initial states and let $u = u_1 u_2 \ldots$ be a word. A *run graph* of $\mathcal{A}$ over u is a vertex-labelled directed acyclic graph $G_u^{\mathcal{A}} = (V, E, \overline{\mu})$ where $V \subseteq Q \times \omega$ is a set of vertices, $E \subseteq V \times V$ is a set of edges and $\overline{\mu}\colon \{\bullet\} \to 2^V$ is a vertex-labelling function. V, E, and $\overline{\mu}$ are defined as follows.

$$\begin{aligned} V &= \big\{(q,0) \mid q \in I\big\} \cup \big\{(q, i+1) \mid i \geq 0 \text{ and } q \in \theta_\delta(\{q\}, u_{..i}) \text{ and } q \in I\big\} \\ E &= \big\{\big((q_1, i), (q_2, i+1)\big) \in V \times V \mid i \geq 0 \text{ and } (q_1, u_i, q_2) \in \delta\big\} \\ \overline{\mu}(\bullet) &= \big\{(q_1, i) \in V \mid q_1 \in \mu(\bullet)\big\} \end{aligned}$$

The vertices on the ith level represent states reachable in $\mathcal{A}$ under the prefix of u of length i. Edges correspond to transitions and also the placement of marks is preserved.

As we will only use run graphs of $\mathcal{A}$ we will only write G_u instead of $G_u^{\mathcal{A}}$. Each infinite path in G_u that starts in $(q, 0)$ where q is some initial state represents a run of $\mathcal{A}$ over u and conversely, each run of $\mathcal{A}$ over u is represented by an unique infinite path in G_u.

The run graph G_u is *rejecting* if no path in G_u satisfies the Büchi condition. That is, G_u is rejecting iff $\mathcal{A}$ has no accepting run over u and thus iff u is not in the language of $\mathcal{A}$.

[17] Kurshan (1994), "The Complexity of Verification", [76].

Ranks. The property that a run graph G_u is rejecting can be expressed in terms of *ranks*.[18] We call a vertex $v \in V$ of a graph $G_u = (V, E, \overline{\mu})$ *safe*, if no vertex v' reachable from v is accepting (v' is accepting if $v' \in \overline{\mu}(\bullet)$), and *finite*, if the set of vertices reachable from v is finite. Based on these definitions, *ranks* can be assigned to the vertices of a rejecting run graph. We set $G^0_u = G_u$, and repeat the following procedure until a fixed point is reached, starting with $i = 1$:

1. Assign all safe vertices of G^{i-1}_u the rank i, and set G^i_u to G^{i-1}_u minus the vertices with rank i.

Remove safe vertices for G^i_u

2. Assign all finite vertices of G^i_u the rank $i + 1$, and set G^{i+1}_u to G^i_u minus the vertices with rank $i + 1$.

Remove finite vertices for G^{i+1}_u

3. Increase i by 2.

A fixed point is reached in $|Q| + 2$ steps,[19] and the ranks can be used to characterise the complement language of a nondeterministic Büchi automaton:

Proposition 1. *A nondeterministic Büchi automaton $\mathcal{A}$ with n states rejects a word w iff G^{2n+2}_u is empty.*[20] □

Ranks and complementation of sDBA. Let us now consider the situation when $\mathcal{A}$ is an sDBA. Then we only need to consider three ranks: 1, 2, and 3. Moreover, the vertices $Q_D \times \omega$ reachable from accepting vertices can only have rank 1 or rank 2 in a rejecting run graph.

Proposition 2. *A semi-deterministic Büchi automaton $\mathcal{A}$ rejects a word w iff G^3_u is empty. This is the case iff G^2_u contains no vertex in $Q_D \times \omega$.*

Proof. Let u be a word rejected by $\mathcal{A}$. By construction, G^1_u contains no safe vertices as removing safe vertices does not introduce new safe vertices.

G^1_u is built from G^0_u by removing safe vertices.

Let us assume for contradiction that G^1_u contains a vertex $(q_i, i) \in Q_D \times \omega$ that is not finite. As (q_i, i) is not finite, there is a run $\sigma = q_0 q_1 \dots q_i q_{i+1} \dots$ of $\mathcal{A}$ over u such that (q_j, j) is a vertex in G^1_u for all $j \geq i$. This is because $q_i \in Q_D$, the deterministic part of $\mathcal{A}$, and $\{(q_j, j) \mid j \geq i\}$ is therefore (1) determined by u and (q_i, i), and (2) fully in G^1_u because otherwise (q_i, i) would be finite.

G^2_u is built from G^1_u by removing finite vertices.

But if all vertices in $\{(q_j, j) \mid j \geq i\}$ are in G^1_u, then none of them is safe in G_u. Using again that the tail $q_i q_{i+1} q_{i+2} \dots$ is unique and well defined (as $q_i \in Q_D$, the deterministic part of $\mathcal{A}$), it follows that, for all $j \geq i$, there is an index $k \geq j$ such that q_k is marked by $\bullet$. Consequently, σ is accepting (contradiction).

We have thus shown that if $\mathcal{A}$ rejects a word u, then G^2_u contains no state in $Q_D \times \omega$. This also implies that G^2_u contains no accepting vertices. Consequently, all vertices in G^2_u are safe and thus G^3_u is empty. □

G^3_u is built from G^2_u by removing safe vertices.

[18] Kupferman and Vardi (2001), [70].

[19] It is common to use 0 as the minimal rank and start with the finite vertices, but the correctness of the complementation does not rely on this. The proof in [70] refers to this case, and requires $|Q| + 1$ steps. For our purpose, the minimal rank needs to be odd, i.e. we need to start with safe vertices.

[20] Kupferman and Vardi (2001), [70].

Rational runs. We now consider the NCSB construction from the perspective of ranks. We start with an intuition for *rational* runs of the complement automaton. Let $\sigma = (N_0, C_0, S_0, B_0)(N_1, C_1, S_1, B_1)(N_2, C_2, S_2, B_2)\ldots$ be a run of $\mathcal{C}$ over a word u such that $u \notin L(\mathcal{A})$ and let $G_u = (V, E, \overline{\mu}(\bullet))$ be the run graph of $\mathcal{A}$ over u. The run σ is *rational* if it is the unique accepting run of $\mathcal{C}$ over u which guesses the ranks precisely, that is:

- $N_i = \{q \mid (q, i) \in V, q \in Q_N\}$,
- $C_i = \{q \mid (q, i) \in V, q \in Q_D$ and the rank of (q, i) is $2\}$,
- $S_i = \{q \mid (q, i) \in V, q \in Q_D$ and the rank of (q, i) is $1\}$,
- $B_i \subseteq C_i$.

We need to check that these vertices are finite in G_u^1.

These vertices are safe in G_u.

All runs of $\mathcal{C}$ that differ on some i from the rational run will either block or will keep the wrongly guessed vertices with rank 1 in C and B and thus will not be accepting.

Correctness. We now establish that the automaton $\mathcal{C}$ is an unambiguous automaton that recognises the complement language of $\mathcal{A}$ by showing

1. $\mathcal{C}$ does not accept a word that is accepted by $\mathcal{A}$,
2. for a word that is not accepted by $\mathcal{A}$, we have a unique rational run of $\mathcal{C}$ and this run is accepting, and
3. for a word u that is not accepted by $\mathcal{A}$, the rational run is the only accepting run of $\mathcal{C}$ over u.

$u \in L(\mathcal{A}) \implies u \notin L(\mathcal{C})$

$u \notin L(\mathcal{A}) \implies u \in L(\mathcal{C})$

unambiguity of $\mathcal{C}$

Lemma 8.1. *Let $\mathcal{A}$ be an sDBA, $\mathcal{C}$ be constructed by the NCSB complementation of $\mathcal{A}$, and $u \in L(\mathcal{A})$ be a word in the language of $\mathcal{A}$. Then $\mathcal{C}$ does not accept u.*

Proof. Let $\sigma = q_0 q_1 \ldots$ be an accepting run of $\mathcal{A}$ over u, and let $i \in \omega$ be an index such that $q_i \in \mu(\bullet)$. Let us assume for contradiction that we have an accepting run $\sigma' = (N_0, C_0, S_0, B_0)(N_1, C_1, S_1, B_1)\ldots$ of $\mathcal{C}$ over u. As q_i is marked it holds that $q_i \in C_i$ and thus $q_j \in C_j \cup S_j$ for all $j \geq i$. We look at the following case distinction.

1. For all $j \geq i$, $q_j \in C_j$. As σ' is accepting, there is a breakpoint ($B_j = \emptyset$) for some $l \geq i$. For such an index we have that $q_{l+1} \in B_{l+1}$ and, moreover, that $q_k \in B_k$ for all $k \geq l+1$. Thus, $B_k \neq \emptyset$ for all $k \geq l+1$ and σ' visits only finitely many marks (contradiction).
2. There is a $j \geq i$ such that $q_j \in S_j$. But then $q_k \in S_k$ holds for all $k \geq j$ by construction. However, as σ is accepting, there is an $l \geq j$ such that $q_l \in \mu(\bullet)$, which contradicts $q_l \in S_l$ (contradiction).

□

Lemma 8.2. *Let $\mathcal{A}$ be an sDBA, $\mathcal{C}$ be the automaton constructed by the NCSB complementation of $\mathcal{A}$, $u \notin L(\mathcal{A})$, and $(V, E, \overline{\mu}(\bullet)) = G_u$ be the run graph of $\mathcal{A}$ over u. Then there is exactly one* rational *run σ of $\mathcal{C}$ over u and it has the form $\sigma = (N_0, C_0, S_0, B_0)(N_1, C_1, S_1, B_1)\ldots$. The run σ is accepting.*

Proof. It is easy to check that the rational run is unique: the updates of the N, C, and S components follow the rules for transitions from the definition of $\mathcal{C}$ and the ranks of vertices from G_u, and the update of the B component is fully determined by the update of C and the previous value of B.

What remains is to show that the run σ is accepting. Let us assume for contradiction that there are only finitely many breakpoints reached, i.e. there is an index $i \in \omega$, for which there is no $j \geq i$, such that $B_j = \emptyset$.

Now we have that $\emptyset \neq B_i \subseteq C_i$ where

$$C_i = \{q \mid (q,i) \in V \text{ s.t. } q \in Q_D \text{ and the rank of } (q,i) \text{ is } 2\}$$

The construction provides that, if there is no breakpoint on or after position i, then B_j is the set of states that correspond to vertices from $Q \times \{j\}$ reachable in G_u^1 from the vertices $B_i \times \{i\}$ for all $j \geq i$. As there is no future breakpoint, there are infinitely many such vertices, and König's lemma implies that there is an infinite path in G_u^1 from at least one of the vertices in $B_i \times \{i\}$. This provides a contradiction to the assumption that the rank of these vertices is 2, i.e. that they are finite in G_u^1. □

Lemma 8.3. *Let $\mathcal{A}$ be an sDBA, $\mathcal{C}$ be the automaton constructed by the NCSB complementation of $\mathcal{A}$, $u \notin L(\mathcal{A})$, and $(V, E, \overline{\mu}(\bullet)) = G_u$ be the run graph of $\mathcal{A}$ over u. Let $\sigma = (N_0, C_0, S_0, B_0)(N_0, C_1, S_1, B_1)\dots$ be a non-rational run of $\mathcal{C}$ over u; that is, σ does not satisfy*

- $N_i = \{q \mid (q,i) \in V$ *s.t.* $q \in Q_N\}$,
- $C_i = \{q \mid (q,i) \in V$ *s.t.* $q \in Q_D$ *and the rank of* (q,i) *is* $2\}$,
- $S_i = \{q \mid (q,i) \in V$ *s.t.* $q \in Q_D$ *and the rank of* (q,i) *is* $1\}$,

for some i. Then σ is rejecting.

Proof. As the N component always tracks the reachable states in Q_N correctly by construction, and the $C \cup S$ part always tracks the reachable states in Q_D correctly by construction, we have one of the following two cases according to Proposition 2.

Safe vertices have rank 1.

1. There is a safe vertex $(q,i) \in V$ such that $q \in C_i$. By construction, a unique maximal path $(q_i,i)(q_{i+1},i+1)(q_{i+2},i+2)\dots$ for $q_i = q$ exists in G_u and this path does not contain any state marked by ●. By an inductive argument, for all vertices (q_j,j) on this path, $q_j \in C_j$. If the path is finite, σ blocks at the end (due to the definition of the transition function of $\mathcal{C}$), which contradicts the assumption that σ is a run (which is infinite by definition). Similarly, if the path is infinite we have $q_k \in B_k$ for some $k \geq i$. Then $q_j \in B_j$ for all $j \geq k$ with (q_j,j) on this path. Therefore, σ cannot be accepting.
2. There is a non-safe vertex in $(q,i) \in V$ such that $q \in S_i$, which implies that q is not marked by ●. By construction, we get a unique maximal path $(q_i,i)(q_{i+1},i+1)(q_{i+2},i+2)\dots$ in G_u such that $q_i = q$ and this path contains a marked state q_k. By an inductive argument, all vertices (q_j,j) on this path are in S_j. But this includes the marked state q_k is also in S_k which is not permitted by the construction of $\mathcal{C}$ (contradiction). □

The first two lemmata provide the correctness of our complementation algorithm and the third lemma establishes that $\mathcal{C}$ is unambiguous. All together they prove the following theorem.

Theorem 8.4. *Let $\mathcal{A}$ be an sDBA and $\mathcal{C}$ be the automaton constructed by the NCSB complementation of $\mathcal{A}$. Then $\mathcal{C}$ is an unambiguous Büchi automaton that recognises the complement of the language of $\mathcal{A}$.*

8.4 ON-THE-FLY APPROACH

Some algorithms do not need to construct the whole complement automaton. For example, in order to verify that $u \notin L(\mathcal{A})$ one only needs to build the accepting lasso in $\mathcal{C}$ for u. Or when building a product with some other automaton (or Markov chain), it is unnecessary to build the part of $\mathcal{C}$ which is not used in the product. Further, some tools work with implicitly encoded automata and/or query an SMT solver to check the presence of a transition in the automaton, which is expensive. ULTIMATE BÜCHI AUTOMIZER has both properties: it stores automata in an implicit form and builds a product of the complement with a program flow-graph. Such tools can greatly benefit from an *on-the-fly* complementation that does not rely on the knowledge of the whole input automaton.

The NCSB complementation can be easily adapted for an on-the-fly implementation. Because we have no knowledge about Q_N, Q_D, and δ_c, the runs are held in N until they reach a state with a mark (which has to be in Q_D), only then they are moved to C.

Technically, the "$N' = \tau_{\delta_N}(N, a)$" from the definition of $\delta_{\mathcal{C}}$ would be replaced by "$N' = \tau_\delta(N, a) \setminus \mu(\bullet)$" and for C' now holds:

$$C' \subseteq \tau_\delta(C, a) \cup (\tau_\delta(N, a) \cap (\bullet))$$

As non-marked states of Q_D can also appear in N the complexity of the on-the-fly variant increases slightly.

$$|P| \leq 2^{|Q_N|} \cdot 3^{|\mu(\bullet)|} \cdot 5^{|Q_D \setminus \mu(\bullet)|}$$

Note that the on-the-fly construction does not add any further nondeterminism to the construction. Furthermore, there is an injection of runs from the original NCSB construction to this on-the-fly variant. Therefore, the correctness argument and the uniqueness argument for the accepting run which are given in Section 8.3 require only minor adjustments.

8.5 IMPLEMENTATION

We implemented the NCSB complementation in two tools. One implementation is available in the GOAL tool.[21] GOAL is an interactive graphical tool for ω-automata, temporal logics, and games. It provides several Büchi complementation algorithms and has been used in an extensive evaluation of these algorithms.[22] In the command-line version, the parameters that run NCSB construction are `complement -m sdbw -a`. The partition of the set Q into Q_N and Q_D is not a parameter, instead the implementation uses the set of all states that are reachable from some accepting state as Q_D.

[21] Tsai, Tsay, and Hwang (2013), "GOAL for Games, Omega-Automata, and Logics", [78], available at http://goal.im.ntu.edu.tw/.

[22] Tsai et al. (2014), "State of Büchi Complementation", [79].

Our second implementation is available in the ULTIMATE AUTOMATA LIBRARY. This library is used by the termination analyser ULTIMATE BÜCHI AUTOMIZER and other tools of the ULTIMATE program analysis framework.[23] This implementation uses the on-the-fly variant of the construction. The library provides a language that allows users to define automata and a sequence of commands that should be executed by the library. This language is called *automata script* and an interpreter for this language is available via a web interface on the tool's website. The operation that implements the NCSB construction has the name `buchiComplementNCSB`.

8.6 EXPERIMENTAL EVALUATION

This section evaluates how the NCSB complementation performs in practice in comparison to other methods for complementation of nondeterministic Büchi automata. All automata, tools, scripts, and commands used in the evaluation with some further comparisons can be found at https://github.com/xblahoud/NCSB-Complementation.

Termination analysis. Program termination analysis is a model checking problem, where the aim is to prove that a given program terminates on all inputs. In other words, it tries to establish (or disprove) that all infinite execution paths in the program flowgraph are infeasible. ULTIMATE BÜCHI AUTOMIZER uses an sDBA to represent infinite paths that are already known to be infeasible. It needs to complement the sDBA and make the product with the program flowgraph to identify the set of infinite execution paths whose infeasibility still needs to be proven.

Benchmark automata. For the evaluation, we took automata whose complementation was needed while the tool ULTIMATE BÜCHI AUTOMIZER was analysing the programs from the Termination category of the software verification competition SV-COMP 2015.[24] We wrote each Büchi automaton that was semi-deterministic but not deterministic to a file in the HOA format.[25] We obtained 106 semi-deterministic Büchi automata. Among these automata, we have identified 97 automata that were pairwise non-isomorphic.

By construction, all these automata behave deterministically only after the first visit of an accepting state. Hence the partition of the states Q into Q_N and Q_D is unique and the results of the original construction and the results of the on-the-fly modification presented in Section 8.4 coincide.

Other complementation constructions. For each category of methods for complementing NBA described in Section 8.1, GOAL provides implementations that can be adjusted by various parameters. We included one construction from each category. For the latter three categories, we took the arguments that were most successful in the extensive evaluation mentioned earlier.[26] For the Ramsey-based category we used additionally an optimization that minimizes the finite automata used during the complementation.[27] The commands that we used are listed in Table 8.1.

[23] http://ultimate.informatik.uni-freiburg.de/

[24] Beyer (2015), "Software Verification and Verifiable Witnesses - (Report on SV-COMP 2015)", [80].

[25] Babiak et al. (2015), "The Hanoi Omega-Automata Format", [17].

[26] Tsai et al. (2014), [79].

[27] Breuers, Löding, and Olschewski (2012), "Improved Ramsey-Based Büchi Complementation", [81].

construction	GOAL command
Ramsey-based	`complement -m ramsey -macc -min`
Determinization-based	`complement -m piterman -macc -sim -eq`
Rank-based	`complement -m rank -macc -tr -ro -cp`
Slice-based	`complement -m slice -macc -eg -madj -ro`
NCSB	`complement -m sdbw -a`

Table 8.1: Complementation constructions and their GOAL commands.

Hardware. All complementations were run on a laptop with an Intel Core i5 2.70GHz CPU. We restricted the maximal heap space of the JVM to 8GB (all complementations are implemented in Java) and used a timeout of 300s.

8.6.1 Results and observations

All algorithms of Table 8.1 were applied to the 97 pairwise non-isomorphic sDBA. We present the results in Table 8.2 and in Figure 8.2. For 91 out of the 97 sDBA, all implementations were able to compute a result. We refer to these 91 sDBA as easy sDBA, while the remaining six are referenced as difficult in the Table 8.2. For each complementation, we provide the cumulative numbers of states and transitions of all 91 easy complements. For each of the easy sDBA, NCSB construction produces the complement with the smallest number of states. In Figure 8.2, a size of the complement produced by the NCSB construction is compared to the size of the smallest complement produced by the other constructions for each of the easy sDBA.

For the difficult sDBA, at least one construction was not able to provide the result within the given time and memory limits. We provide the number of states of the computed complements for each of them. While there are two cases where the determinization-based construction produced an automaton with fewer states than the NCSB construction, the number of transition was always smaller for the NCSB construction.

Simplifications. A common approach to mitigate the problem of large results of complementation is to apply generic size reduction algorithms. Does our NCSB construction also outperform the other constructions if we apply size reduction techniques afterwards? In order to address this question, we applied the simplification routines of the Spot library (in version 1.99.4a) to the complements. We ran the command `autfilt --small --high -B -H` with a timeout of 300s and obtained the results that are presented in Table 8.3.

construction	91 easy sDBA		6 difficult sDBA					
	states	transitions	1	2	3	4	5	6
Ramsey-based	16909	848969	–	–	–	–	–	–
Rank-based	2703	21095	–	–	1022	7460	8245	–
Det.-based	1841	24964	–	–	172	346	385	3527
Slice-based	1392	14783	66368	–	184	421	475	9596
NCSB	950	8003	20711	84567	108	343	401	5449

Table 8.2: Performance of complementation algorithms without posteriori simplifications. For every algorithm we first show the cummulative numbers of states and edges summed for the 91 easy sDBA. The last 6 columns give the number of states for the difficult sDBA.

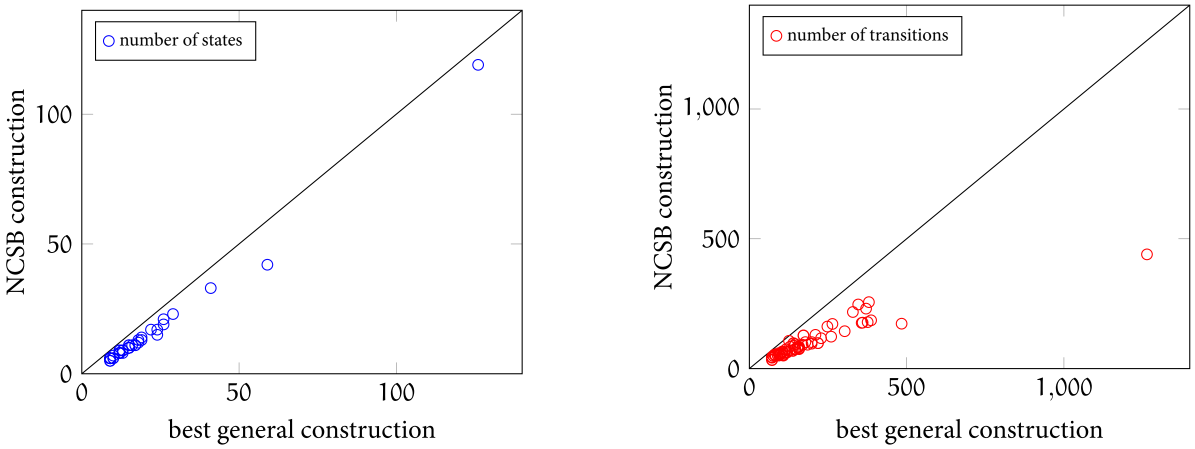


Figure 8.2: Comparison of the NCSB construction and other complementations.

For 75 sDBA, all complements could be simplified within the timeout. For these, we again provide the cumulative numbers of states and transitions before and after the simplifications. The column *min* shows how often each construction followed by simplification produced a complement with the minimal number of states. The column *failure* shows how often a timeout prevented successful complementation or simplification. It is interesting to see that the simplifications were not able to reduce the number of transitions much for the NCSB construction, while they were able to reduce it by more than 20% in case of the other complementations.

From the presented results we see that NCSB construction brings significant improvement to complementation of sDBA both in theory and practice. The results indicate that additional simplifications applied to the complement automata do not help the other tools to outperform the NCSB construction on sDBA. Further, the construction was successfully used in the termination analysis of the tool ULTIMATE BÜCHI AUTOMIZER.

construction	no simplifications		with simplifications			failure	
	states	transitions	states	transitions	min	compl.	simp.
Ramsey-based	6386	172351	5223	90548	0	6	22
Rank-based	1437	11677	899	7657	4	3	14
Det.-based	1300	15491	1083	9589	0	2	11
Slice-based	892	8921	785	6789	4	1	13
NCSB	598	4922	514	4460	73	0	10

Table 8.3: Performance of complementation algorithms without and with posteriori simplifications. For every algorithm we show the cumulative number of states and transitions before and after simplifications summed for the 74 cases when all algorithms succeeded in given timeout, and the number of cases where the result of the algorithm was simplified to the smallest automaton for the same input. The last two columns show how often complementations and simplifications ran out of time or memory.

Bibliography

[1] J. Richard Büchi (1962).
On a Decision Method in Restricted Second Order Arithmetic. In *1960 International Congress for Logic, Methodology and Philosophy of Science*. (Cited on pages 15, 125).

[2] Moshe Y. Vardi and Pierre Wolper (1986).
An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report). In *Proceedings of the 1st IEEE Symposium on Logic in Computer Science (LICS'86)*. IEEE Computer Society, pages 332–344. (Cited on pages 15, 105).

[3] Michael O. Rabin and Dana S. Scott (1959).
Finite Automata and Their Decision Problems. *IBM Journal of Research and Development* 3, pages 114–125. (Cited on page 16).

[4] Sven Schewe (2009).
Tighter Bounds for the Determinisation of Büchi Automata. In *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures (FOSSACS'09)*. Lecture Notes in Computer Science (vol. 5504). Springer, pages 167–181. (Cited on pages 16, 77).

[5] Thomas Colcombet and Konrad Zdanowski (2009).
A Tight Lower Bound for Determinization of Transition Labeled Büchi Automata. In *Proceedings of the 36th Internatilonal Colloquium on Automata, Languages and Programming, (ICALP'09)*. Lecture Notes in Computer Science (vol. 5556), part II. Springer, pages 151–162. (Cited on page 16).

[6] Sven Schewe (2009).
Büchi Complementation Made Tight. In *Proceedings of the 26th International Symposium on Theoretical Aspects of Computer Science (STACS'09)*. LIPIcs (vol. 3). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pages 661–672. (Cited on pages 17, 125).

[7] Qiqi Yan (2008).
Lower Bounds for Complementation of Omega-Automata Via the Full Automata Technique. *Logical Methods in Computer Science* 4. (Cited on pages 17, 126).

[8] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski (2014).
Termination Analysis by Learning Terminating Programs. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV'14)*. Lecture Notes in Computer Science (vol. 8559). Springer, pages 797–813. (Cited on pages 17, 125).

[9] Costas Courcoubetis and Mihalis Yannakakis (1988).
Verifying Temporal Properties of Finite-State Probabilistic Programs. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science (FOCS'88)*. IEEE Computer Society, pages 338–345. (Cited on pages 19, 107).

[10] Ernst Moritz Hahn, Guangyuan Li, Sven Schewe, Andrea Turrini, and Lijun Zhang (2015).
Lazy Probabilistic Model Checking without Determinisation. In *Proceedings of the 26th International Conference on Concurrency Theory (CONCUR'15)*. LIPIcs (vol. 42). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pages 354–367. (Cited on pages 19, 105, 114).

[11] František Blahoudek, Alexandre Duret-Lutz, Mojmír Křetínský, and Jan Strejček (2014).
Is there a Best Büchi Automaton for Explicit Model Checking? In *Proceedings of 21st International SPIN Symposium on Model Checking of Software (SPIN'14)*. ACM, pages 68–76. (Cited on page 19).

[12] František Blahoudek, Alexandre Duret-Lutz, Vojtěch Rujbr, and Jan Strejček (2015).
On Refinement of Büchi Automata for Explicit Model Checking. In *Proceedings of 22nd International SPIN Symposium on Model Checking of Software (SPIN'15)*. Lecture Notes in Computer Science (vol. 9232). Springer, pages 66–83. (Cited on pages 19, 20).

[13] Tomáš Babiak, František Blahoudek, Mojmír Křetínský, and Jan Strejček (2013).
Effective Translation of LTL to Deterministic Rabin Automata: Beyond the (F, G)-Fragment. In *Proceedings of the 11th International Symposium on Automated Technology for Verification and Analysis (ATVA'13)*. Lecture Notes in Computer Science (vol. 8172). Springer, pages 24–39. (Cited on pages 19, 20, 77).

[14] František Blahoudek, Mojmír Křetínský, and Jan Strejček (2013).
Comparison of LTL to Deterministic Rabin Automata Translators. In *Proceedings of the 19th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-19)*. Lecture Notes in Computer Science (vol. 8312). Springer, pages 164–172. (Cited on pages 19, 20, 77, 80).

[15] František Blahoudek, Alexandre Duret-Lutz, Mikuláš Klokočka, Mojmír Křetínský, and Jan Strejček (2017).
Seminator: A Tool for Semi-Determinization of Omega-Automata. In *Proceedings of the 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR-21)*. EPiC Series in Computing (vol. 46). EasyChair, pages 356–367. (Cited on page 20).

[16] František Blahoudek, Matthias Heizmann, Sven Schewe, Jan Strejček, and Ming-Hsien Tsai (2016).
Complementing Semi-deterministic Büchi Automata. In *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'16)*. Lecture Notes in Computer Science (vol. 9636). Springer, pages 770–787. (Cited on pages 20, 105).

[17] Tomáš Babiak, František Blahoudek, Alexandre Duret-Lutz, Joachim Klein, Jan Křetínský, David Müller, David Parker, and Jan Strejček (2015).
The Hanoi Omega-Automata Format. In *Proceedings of the 27th International Conference on Computer Aided Verification (CAV'15)*. Lecture Notes in Computer Science (vol. 9206), part I. Springer, pages 479–486. (Cited on pages 21, 23, 116, 134).

[18] Christel Baier and Joost-Pieter Katoen (2008).
Principles of Model Checking. MIT Press. (Cited on page 29).

[19] Moshe Y. Vardi (1995).
An Automata-Theoretic Approach to Linear Temporal Logic. In *8th Banff Higher Order Workshop (Banff'95)*. Lecture Notes in Computer Science (vol. 1043). Springer, pages 238–266. (Cited on page 29).

[20] Gerard J. Holzmann (1997).
The Model Checker SPIN. *IEEE Transaction on Software Engineering* 23, pages 279–295. (Cited on page 30).

[21] Gerard J. Holzmann (2003).
The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley. (Cited on pages 30, 32, 47).

[22] Radek Pelánek (2008).
Fighting State Space Explosion: Review and Evaluation. In *Proceedings of the 13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'08)*. Lecture Notes in Computer Science (vol. 5596). Springer, pages 37–52. (Cited on page 30).

[23] Gerard J. Holzmann, Doron A. Peled, and Mihalis Yannakakis (1996).
On Nested Depth First Search. In *Proceedings of the 2nd Spin Workshop (SPIN'96)*. DIMACS: Series in Discrete Mathematics and Theoretical Computer Science (vol. 32). American Mathematical Society, pages 23–32. (Cited on page 30).

[24] Paul Gastin, Pierre Moro, and Marc Zeitoun (2004).
Minimization of Counterexamples in SPIN. In *Proceedings of the 11th International SPIN Workshop on Model Checking of Software (SPIN'04)*. Lecture Notes in Computer Science (vol. 2989). Springer, pages 92–108. (Cited on page 31).

[25] Stefan Schwoon and Javier Esparza (2005).
A Note on On-the-Fly Verification Algorithms. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*. Lecture Notes in Computer Science (vol. 3440). Springer, pages 174–190. (Cited on page 31).

[26] Kousha Etessami and Gerard J. Holzmann (2000).
Optimizing Büchi Automata. In *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR'00)*. Lecture Notes in Computer Science (vol. 1877). Springer, pages 153–167. (Cited on pages 31, 35, 47, 82, 83, 119).

[27] Christian Dax, Jochen Eisinger, and Felix Klaedtke (2007).
Mechanizing the Powerset Construction for Restricted Classes of ω-Automata. In *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis (ATVA'07)*. Lecture Notes in Computer Science (vol. 4762). Springer, pages 223–236. (Cited on pages 31, 35, 46).

[28] Jaco Geldenhuys and Antti Valmari (2005).
More Efficient On-the-Fly LTL Verification with Tarjan's Algorithm. *Theoretical Computer Science* 345, pages 60–82. (Cited on page 31).

[29] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce (2011).
Swarm Verification Techniques. *IEEE Transaction on Software Engineering* 37, pages 845–857. (Cited on page 31).

[30] Radek Pelánek (2007).
BEEM: Benchmarks for Explicit Model Checkers. In *Proceedings of the 14th international SPIN conference on Model checking software (SPIN'07)*. Lecture Notes in Computer Science (vol. 4595). Springer, pages 263–267. (Cited on pages 32, 47, 82, 83, 119).

[31] Paul Gastin and Denis Oddoux (2001).
Fast LTL to Büchi Automata Translation. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*. Lecture Notes in Computer Science (vol. 2102). Springer, pages 53–65. (Cited on pages 32, 34, 38, 47, 63, 66, 84).

[32] Roberto Sebastiani and Stefano Tonetta (2003).
More Deterministic vs. *Smaller* Büchi Automata for Efficient LTL Model Checking. In *Proceedings of the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'03)*. Lecture Notes in Computer Science (vol. 2860). Springer, pages 126–140. (Cited on pages 32, 34).

[33] Tomáš Babiak, Mojmír Křetínský, Vojtěch Řehák, and Jan Strejček (2012).
LTL to Büchi Automata Translation: Fast and More Deterministic. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12)*. Lecture Notes in Computer Science (vol. 7214). Springer, pages 95–109. (Cited on pages 32, 47, 74).

[34] Alexandre Duret-Lutz (2014).
LTL Translation Improvements in Spot 1.0. *International Journal on Critical Computer-Based Systems* 5, pages 31–54. (Cited on pages 32, 46, 47).

[35] Jean-Michel Couvreur (1999).
On-the-Fly Verification of Linear Temporal Logic. In *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM'99)*. Lecture Notes in Computer Science (vol. 1708). Springer, pages 253–271. (Cited on page 34).

[36] Fabio Somenzi and Roderick Bloem (2000).
Efficient Büchi Automata from LTL Formulae. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*. Lecture Notes in Computer Science (vol. 1855). Springer, pages 248–263. (Cited on pages 34, 35, 82, 83, 119).

[37] Dimitra Giannakopoulou and Flavio Lerda (2002).
From States to Transitions: Improving Translation of LTL Formulae to Büchi Automata. In *Proceedings of the 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'02)*. Lecture Notes in Computer Science (vol. 2529). Springer, pages 308–326. (Cited on pages 34, 38).

[38] Xavier Thirioux (2002).
Simple and Efficient Translation from LTL Formulas to Büchi Automata. In *Proceedings of the 7th International ERCIM Workshop in Formal Methods for Industrial Critical Systems (FMICS'02)*. Electronic Notes in Theoretical Computer Science (vol. 66). Elsevier, pages 145–159. (Cited on page 34).

[39] Heikki Tauriainen and Keijo Heljanko (2002).
Testing LTL Formula Translation into Büchi Automata. *International Journal on Software Tools for Technology Transfer* 4, pages 57–70. (Cited on page 34).

[40] Alexandre Duret-Lutz and Denis Poitrenaud (2004).
SPOT: An Extensible Model Checking Library Using Transition-Based Generalized Büchi Automata. In *Proceedings of the 12th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'04)*. IEEE Computer Society, pages 76–83. (Cited on page 34).

[41] Alexandre Duret-Lutz (2013).
Manipulating LTL Formulas Using Spot 1.0. In *Proceedings of the 11th International Symposium on Automated Technology for Verification and Analysis (ATVA'13)*. Lecture Notes in Computer Science (vol. 8172). Springer, pages 442–445. (Cited on pages 34, 83).

[42] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett (1998).
Property Specification Patterns for Finite-State Verification. In *Proceedings of the 2nd Workshop on Formal Methods in Software Practice (FMSP'98)*. ACM, pages 7–15. (Cited on pages 35, 82, 83, 119).

[43] Bernard Boigelot, Sébastien Jodogne, and Pierre Wolper (2001).
On the Use of Weak Automata for Deciding Linear Arithmetic with Integer and Real Variables. In *Proceedings of the First International Joint Conference on Automated Reasoning (IJCAR'01)*. Lecture Notes in Computer Science (vol. 2083). Springer, pages 611–625. (Cited on page 36).

[44] Tomáš Babiak, Thomas Badie, Alexandre Duret-Lutz, Mojmír Křetínský, and Jan Strejček (2013).
Compositional Approach to Suspension and Other Improvements to LTL Translation. In *Proceedings of the 20th International SPIN Symposium on Model Checking of Software (SPIN'13)*. Lecture Notes in Computer Science (vol. 7976). Springer, pages 81–98. (Cited on pages 46, 117).

[45] Shin-ichi Minato (1993).
Fast Generation of Prime-Irredundant Covers from Binary Decision Diagrams. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 76, pages 967–973. (Cited on page 54).

[46] Stefan Edelkamp, Alberto Lluch-Lafuente, and Stefan Leue (2001).
Directed Explicit Model Checking with HSF-SPIN. In *Proceedings of the 8th International Spin Workshop on Model Checking of Software (SPIN'01)*. Lecture Notes in Computer Science (vol. 2057). Springer, pages 57–79. (Cited on page 55).

[47] Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente (2004).
Directed Explicit-State Model Checking in the Validation of Communication Protocols. *STTT* 5, pages 247–267. (Cited on page 55).

[48] Krishnendu Chatterjee, Andreas Gaiser, and Jan Křetínský (2013).
Automata with Generalized Rabin Pairs for Probabilistic Model Checking and LTL Synthesis. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV'13)*. Lecture Notes in Computer Science (vol. 8044). Springer, pages 559–575. (Cited on page 63).

[49] Denis Oddoux (2003).
Utilisation des Automates Alternants pour un Model-Checking Efficace des Logiques Temporelles Linéaires. French. PhD thesis. Université Paris 7 - Denis Diderot UFR d'Informatique, 2003. (Cited on page 67).

[50] Shmuel Safra (1988).
On the Complexity of Omega-Automata. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science (FOCS'88)*. IEEE Computer Society, pages 319–327. (Cited on pages 77, 125).

[51] Nir Piterman (2007).
From Nondeterministic Büchi and Streett Automata to Deterministic Parity Automata. *Logical Methods in Computer Science* 3. (Cited on pages 77, 125).

[52] Roman R. Redziejowski (2012).
An Improved Construction of Deterministic Omega-Automaton Using Derivatives. *Fundamenta Informaticae* 119, pages 393–406. (Cited on page 77).

[53] Joachim Klein (2005).
Linear Time Logic and Deterministic ω-Automata. MA thesis. University of Bonn, 2005. (Cited on page 77).

[54] Joachim Klein and Christel Baier (2006).
Experiments with Deterministic Omega-Automata for Formulas of Linear Temporal Logic. *Theoretical Computer Science* 363, pages 182–195. (Cited on page 77).

[55] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu (2016).
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)*. Lecture Notes in Computer Science (vol. 9938), pages 122–129. (Cited on pages 77, 116).

[56] Jan Křetínský and Javier Esparza (2012).
Deterministic Automata for the (F, G)-Fragment of LTL. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV'12)*. Lecture Notes in Computer Science (vol. 7358). Springer, pages 7–22. (Cited on page 77).

[57] Andreas Gaiser, Jan Křetínský, and Javier Esparza (2012).
Rabinizer: Small Deterministic Automata for LTL(F, G). In *Proceedings of the 10th International Symposium on Automated Technology for Verification and Analysis (ATVA'12)*. Lecture Notes in Computer Science (vol. 7561). Springer, pages 72–76. (Cited on page 77).

[58] Jan Křetínský and Ruslán Ledesma-Garza (2013).
Rabinizer 2: Small Deterministic Automata for LTL$\setminus$GU. In *Proceedings of the 11th International Symposium on Automated Technology for Verification and Analysis (ATVA'13)*. Lecture Notes in Computer Science (vol. 8172). Springer, pages 446–450. (Cited on page 77).

[59] Javier Esparza and Jan Křetínský (2014).
From LTL to Deterministic Automata: A Safraless Compositional Approach. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV'14)*. Lecture Notes in Computer Science (vol. 8559). Springer, pages 192–208. (Cited on page 77).

[60] Zuzana Komárková and Jan Křetínský (2014).
Rabinizer 3: Safraless Translation of LTL to Small Deterministic Automata. In *Proceedings of the 12th International Symposium on Automated Technology for Verification and Analysis (ATVA'14)*. Lecture Notes in Computer Science (vol. 8837). Springer, pages 235–241. (Cited on page 77).

[61] Javier Esparza, Jan Křetínský, Jean-François Raskin, and Salomon Sickert (2017).
From LTL and Limit-Deterministic Büchi Automata to Deterministic Parity Automata. In *Proceedings of the 23st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'17)*. Lecture Notes in Computer Science (vol. 10205), part I, pages 426–442. (Cited on page 78).

[62] Salomon Sickert, Javier Esparza, Stefan Jaax, and Jan Křetínský (2016).
Limit-Deterministic Büchi Automata for Linear Temporal Logic. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV'16)*. Lecture Notes in Computer Science (vol. 9780), part II. Springer, pages 312–332. (Cited on pages 78, 80, 118).

[63] Joachim Klein and Christel Baier (2007).
On-the-Fly Stuttering in the Construction of Deterministic Omega-Automata. In *Proceedings of the 12th International Conference on Implementation and Application of Automata (CIAA'07)*. Lecture Notes in Computer Science (vol. 4783). Springer, pages 51–61. (Cited on page 80).

[64] Jan Křetínský, Tobias Meggendorfer, Clara Waldmann, and Maximilian Weininger (2017).
Index Appearance Record for Transforming Rabin Automata into Parity Automata. In *Proceedings of the 23st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'17)*. Lecture Notes in Computer Science (vol. 10205), part I, pages 443–460. (Cited on page 80).

[65] Jan Holeček, Tomáš Kratochvíla, Vojtěch Řehák, David Šafránek, and Pavel Šimeček. Verification Results in Liberouter Project. Tech. rep. 03, 32pp. CESNET, Sept. 2004 (cited on pages 82, 83, 119).

[66] Jaco Geldenhuys and Henri Hansen (2006).
Larger Automata and Less Work for LTL Model Checking. In *Proceedings of the 13th International SPIN Symposium on Model Checking of Software (SPIN'06)*. Lecture Notes in Computer Science (vol. 3925). Springer, pages 53–70. (Cited on page 84).

[67] David Müller and Salomon Sickert (2017).
LTL to Deterministic Emerson-Lei Automata. In *Proceedings of the 8th International Symposium on Games, Automata, Logics and Formal Verification (GandALF'17)*. EPTCS (vol. 256), pages 180–194. (Cited on page 84).

[68] Orna Kupferman and Adin Rosenberg (2010).
The Blowup in Translating LTL to Deterministic Automata. In *Revised Selected and Invited Papers from the 6th International Workshop on Model Checking and Artificial Intelligence (MoChArt'10)*. Lecture Notes in Computer Science (vol. 6572). Springer, pages 85–94. (Cited on page 84).

[69] Dileep Kini and Mahesh Viswanathan (2017).
Optimal Translation of LTL to Limit Deterministic Automata. In *Proceedings of the 23st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'17)*. Lecture Notes in Computer Science (vol. 10206), part II. Springer, pages 113–129. (Cited on page 118).

[70] Orna Kupferman and Moshe Y. Vardi (2001).
Weak Alternating Automata are not that Weak. *ACM Trans. Comput. Log.* 2, pages 408–429. (Cited on pages 125, 126, 130).

[71] Moshe Y. Vardi (2007).
The Büchi Complementation Saga. In *Proceedings of the 24th Annual Symposium on Theoretical Aspects of Computer Science (STACS'07)*. Lecture Notes in Computer Science (vol. 4393). Springer, pages 12–22. (Cited on page 125).

[72] A. Prasad Sistla, Moshe Y. Vardi, and Pierre Wolper (1987).
The Complementation Problem for Büchi Automata with Appplications to Temporal Logic. *Theoretical Computer Science* 49, pages 217–237. (Cited on page 125).

[73] Sankar Gurumurthy, Orna Kupferman, Fabio Somenzi, and Moshe Y. Vardi (2003).
On Complementing Nondeterministic Büchi Automata. In *Proceedings of the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'03)*. Lecture Notes in Computer Science (vol. 2860). Springer, pages 96–110. (Cited on page 125).

[74] Ehud Friedgut, Orna Kupferman, and Moshe Y. Vardi (2006).
Büchi Complementation Made Tighter. *International Journal of Foundations of Computer Science* 17, pages 851–868. (Cited on page 125).

[75] Detlef Kähler and Thomas Wilke (2008).
Complementation, Disambiguation, and Determinization of Büchi Automata Unified. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming (ICALP'08)*. Lecture Notes in Computer Science (vol. 5125). Springer, pages 724–735. (Cited on page 125).

[76] Robert P. Kurshan (1994).
The Complexity of Verification. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing (STOC'94)*. ACM, pages 365–371. (Cited on pages 126, 129).

[77] Satoru Miyano and Takeshi Hayashi (1984).
Alternating Finite Automata on Omega-Words. *Theoretical Computer Science* 32, pages 321–330. (Cited on page 126).

[78] Ming-Hsien Tsai, Yih-Kuen Tsay, and Yu-Shiang Hwang (2013).
GOAL for Games, Omega-Automata, and Logics. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV'13)*. Lecture Notes in Computer Science (vol. 8044). Springer, pages 883–889. (Cited on page 133).

[79] Ming-Hsien Tsai, Seth Fogarty, Moshe Y. Vardi, and Yih-Kuen Tsay (2014).
State of Büchi Complementation. *Logical Methods in Computer Science* 10. (Cited on pages 133, 134).

[80] Dirk Beyer (2015).
Software Verification and Verifiable Witnesses - (Report on SV-COMP 2015). In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15)*. Lecture Notes in Computer Science (vol. 9035). Springer, pages 401–416. (Cited on page 134).

[81] Stefan Breuers, Christof Löding, and Jörg Olschewski (2012).
Improved Ramsey-Based Büchi Complementation. In *Proceedings of the 15th International Conference on Foundations of Software Science and Computational Structures (FOSSACS'12)*. Lecture Notes in Computer Science (vol. 7213). Springer, pages 150–164. (Cited on page 134).