Linear Time Logic
and
Deterministic omega-Automata

Joachim Klein

January 2005

Diplomarbeit

Rheinische Friedrich-Wilhelms
Universität Bonn
Institut für Informatik Abt.I
Prof. Dr. Christel Baier

Contents

1 Introduction

Automata on infinite words play a crucial role in logic, for verification purposes and in other areas (e.g. [Var94], [Tho97], [GTW02], [FOS03]), in particular ω-automata and the related ω-regular languages.

In the context of program verification using model checking, to check if a model (program) satisfies a given specification, both model and specification can be regarded as automata on infinite words (ω-automata), allowing to perform operations like union and intersection or checking for language emptiness with graph algorithms on the automata. As it is often easier for the users of a model checker to specify the properties that they want to verify using a formula in a suitable logic (e.g. linear time logic, LTL), an algorithm for translation of formulas to corresponding ω-automata is needed. For LTL formulas, traditionally a conversion to nondeterministic Büchi automata (NBA) is used. Despite a worst case exponential blowup in the size of the formula, in practice the formulas tend to be small and the resulting Büchi automata are of a manageable size for many interesting formulas. The state space explosion of the model tends to be a much bigger problem in explicit state model checkers.

For standard model checking, the nondeterminism of the Büchi automaton does not pose a big problem, for other algorithms, such as the verification of Markov decision processes ([CY95], [VW86], [dA97]), the ω-automata have to be deterministic. As deterministic Büchi automata are not as expressive as the nondeterministic Büchi automata, it is necessary to use deterministic automata with more complex acceptance types, for example Rabin and Streett automata.

Safra [Saf89] proposed an algorithm for the determinization of nondeterministic Büchi automata to deterministic Rabin automata. In the worst case, it has a $2^{\Theta(n \cdot \log n)}$ blowup in the size of the automaton, which was shown to be optimal up to a constant factor in the exponent. Using this result to translate LTL formulas into nondeterministic Büchi automata and then using Safra's construction to translate these into deterministic Rabin automata leads to a worst case double exponential blowup from the size of the formula to the size of the resulting automaton.

One of the main goals of this thesis is to answer the question whether it is feasible to use Safra's construction and deterministic ω-automata for LTL formulas in practice, relying on the same argument used to defend the practical usability of nondeterministic Büchi automata for LTL formulas despite the possible exponential blowup: That the formulas used in practice tend to be of limited length and do not necessarily have the properties that lead to the worst-case performance.

Due to its deterring complexity theoretic properties, very few implementations of Safra's construction exist. This naturally leads to few experimental data of the performance of Safra's construction in practice.

The main contributions of this thesis are:

1. Several improvements to Safra's construction that result in smaller automata in practice.
2. Additional approaches to the translation from LTL to deterministic ω-automata that perform better than the standard construction in some cases.
3. The implementation of Safra's construction in a tool, `ltl2dstar`[1] (**LTL** to **d**eterministic **St**reett **a**nd **R**abin automata), extended to allow translation from LTL to deterministic ω-automata.
4. Experimental results, showing the feasibility of using `ltl2dstar` and deterministic ω-automata for LTL formulas in practical applications.

As the translation of LTL formulas to nondeterministic Büchi automata has been extensively researched and many implementations exist, we can use these external tools to have a state-of-the-art building block.

Overview

In Chapter 2 we will define the notations we will use, define LTL and the different automata types and explore their relationship and basic properties. Chapter 3 will describe Safra's determinization algorithm in detail. Chapter 4 presents several optimizations that can be used to improve Safra's construction and reduce the size of the resulting automaton. Chapter 5 presents the tool `ltl2dstar`, provides a survey of LTL→NBA translators and gives experimental results on the performance of the `ltl2dstar` translator and the different optimizations presented in Chapter 4.

[1] `http://www.ltl2dstar.de`

2 Definitions and Notations

2.1 Notation

For a set S, S^* denotes the set of finite sequences $\sigma = \sigma_0, \sigma_1, \ldots, \sigma_n$, with $\sigma_i \in S$.
The set of infinite sequences $\sigma = \sigma_0, \sigma_1, \ldots$, with $\sigma_i \in S$ is denoted by S^ω.
If S is an alphabet Σ, the sequences in Σ^* and Σ^ω are called *words* over Σ, the empty word of length 0 in Σ^* is denoted by ε.
For two words $\alpha \in \Sigma^*$ and $\beta \in (\Sigma^* \cup \Sigma^\omega)$, the concatenation of α and β is denoted by $\alpha{\cdot}\beta$.
For a sequence σ, we define $\sigma|_i$ as the suffix $\sigma_i, \sigma_{i+1}, \ldots$ of σ starting at index i. $\sigma[i, j)$ denotes the sequence from position i to position $j-1$: $\sigma[i, j) = \sigma_i, \sigma_{i+1}, \ldots, \sigma_{j-1}$.
A language $\mathcal{L}$ over Σ^ω is a subset of Σ^ω: $\mathcal{L} \subseteq \Sigma^\omega$. The complement language, denoted by $\overline{\mathcal{L}}$, is defined as the words from Σ^ω that are not in $\mathcal{L}$:

$$\overline{\mathcal{L}} = \Sigma^\omega \setminus \mathcal{L}$$

The definitions for languages over finite words are similar, only with Σ^*.
x^i denotes the i-times repetition of x, x^0 is empty.
For a set S, 2^S denotes the power set of S (the set of all subsets of S).

2.2 Linear Time Logic (LTL)

Definition 2.2.1 (LTL syntax). The set of LTL formulas over a set of atomic propositions AP is defined by the grammar

$$\varphi ::= \texttt{true} \mid p \mid \neg\,\varphi \mid \varphi \vee \varphi \mid \texttt{X}\ \varphi \mid \varphi\ \texttt{U}\ \varphi,$$

where $p \in \mathrm{AP}$.

Definition 2.2.2 (LTL semantics). Let $\sigma = \sigma_0, \sigma_1, \sigma_2, \ldots$ be an infinite word in $\Sigma = 2^{\mathrm{AP}}$. Let φ be an LTL formula over AP. $\sigma \models \varphi$ is defined as follows:

- $\sigma \models \texttt{true}$.
- $\sigma \models p \in \mathrm{AP}$ iff $p \in \sigma_0$.
- $\sigma \models \neg\varphi$ iff $\sigma \not\models \varphi$.
- $\sigma \models \varphi_1 \vee \varphi_2$ iff $\sigma \models \varphi_1$ or $\sigma \models \varphi_2$.
- $\sigma \models \texttt{X}\,\varphi_1$ iff $\sigma|_1 \models \varphi_1$.
- $\sigma \models \varphi_1\,\texttt{U}\,\varphi_2$ iff $\exists k \geq 0 : \sigma|_k \models \varphi_2$ and $\forall\, 0 \leq i < k : \sigma|_i \models \varphi_1$.

Definition 2.2.3 (Language of an LTL formula). For an LTL formula ψ over a set of atomic propositions AP and with $\Sigma = 2^{\mathrm{AP}}$, define the *language* of ψ as

$$\mathcal{L}(\psi) = \{\sigma \in \Sigma : \sigma \models \psi\}.$$

From the basic operators defined above, we can derive other operators:

Duals			
False	$\texttt{false}$	:=	$\neg\,\texttt{true}$
And	$\varphi_1 \wedge \varphi_2$	:=	$\neg\,(\neg\varphi_1 \vee \neg\varphi_2)$
Release	$\varphi_1\ \texttt{V}\ \varphi_2$	:=	$\neg\,(\neg\varphi_1\ \texttt{U}\ \neg\varphi_2)$
Abbreviations			
Implication	$\varphi_1 \rightarrow \varphi_2$	:=	$(\neg\varphi_1) \vee \varphi_2$
Equivalence	$\varphi_1 \leftrightarrow \varphi_2$	:=	$(\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$
Finally	$\Diamond\ \varphi$	:=	$\texttt{true}\ \texttt{U}\ \varphi$
Globally	$\Box\ \varphi$	:=	$\texttt{false}\ \texttt{V}\ \varphi$

Definition 2.2.4 (Size of a formula). For an LTL formula ψ, the *size* of ψ is denoted by $|\psi|$ and defined as the number of operators in ψ.

Definition 2.2.5 (Positive normal form). A LTL formula φ is in *positive normal form* (PNF)[1], if the formula contains only the operators $\vee, \wedge, \texttt{U}, \texttt{V}$ and $\texttt{X}$ and negation only occurs in front of atomic propositions.
In other words, all formulas that can be generated by the grammar

$$\varphi ::= \texttt{true} \mid \texttt{false} \mid p \mid \neg p \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \texttt{X}\ \varphi \mid \varphi\ \texttt{U}\ \varphi \mid \varphi\ \texttt{V}\ \varphi$$

are in positive normal form, where $p \in \mathrm{AP}$.

It is clear that every LTL formula φ can be converted into an equivalent LTL formula φ' in PNF by using the duals to push negation inward and eliminating double negation.

Definition 2.2.6 (Closure). The closure of an LTL formula ψ, denoted by $cl(\psi)$ is the set of subformulas of ψ, including ψ itself.

[1] PNF is sometimes also called *negation normal form*.

2.3 Automata and Languages

While classical finite automata operate on finite words, ω-automata operate on infinite words. They are classified by the *mode of their transition function* (e.g. deterministic, nondeterministic, ...) and by the type of their *acceptance conditions* (e.g. Büchi, Rabin, Streett, ...).

This section describes the common concepts regarding ω-automata.

ω-automata

An ω-automaton is a five-tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, \Omega)$ where:

- Q is a (finite) set of states,
- Σ is the alphabet,
- $\delta : Q \times \Sigma \to 2^Q$ is the transition function,
- $q_0 \in Q$ is the initial state,
- Ω is an acceptance condition (the different types of acceptance conditions used in this thesis will be introduced later).

ω-automata can be graphically represented most naturally as a graph with the states Q as the vertices and edges labeled with $a \in \Sigma$ corresponding to the transition function δ.

For our purposes, normally $\Sigma = 2^{\mathrm{AP}}$, for a set of atomic propositions AP.

Deterministic automata

If $|\delta(q, a)| = 1$ for all $q \in Q$ and $a \in \Sigma$, the automaton is called *deterministic*. For each element of Σ, all states have exactly one successor and δ can be regarded as $\delta : Q \times \Sigma \to Q$.

For deterministic automata, we can easily extend the transition function δ to have finite words as input: $\delta : Q \times \Sigma^* \to Q$. For $q \in Q$ and $\sigma = \sigma_0, \sigma_1, \ldots, \sigma_n$,

$$\delta(q, \sigma) = \delta(\delta(\ldots\delta(\delta(q, \sigma_0), \sigma_1)\ldots, \sigma_{n-1}), \sigma_n).$$

Run

A sequence of states $\pi = \pi_0, \pi_1, \pi_2, \ldots \in Q^\omega$ is called a *run* over an infinite word $\sigma \in \Sigma^\omega$, if $\pi_0 = q_0$ and for every i, π_{i+1} is a σ_i-successor of π_i: $\pi_{i+1} \in \delta(\pi_i, \sigma_i)$. For deterministic automata, every word $\sigma \in \Sigma^\omega$ corresponds to exactly one run over σ. For nondeterministic automata there can be words with no, one or multiple runs. A run π *visits* a state $q \in Q$ if there exists an i such that $\pi_i = q$.

Infinity set

The infinity set of a run π is the set of states that are visited infinitely often in the run, $Inf(\pi) = \{q \mid \exists^{\omega} i : \pi_i = q\}$[2].

Acceptance & language

A run π is *accepting* if it satisfies the acceptance condition Ω. For the different acceptance types, what "satisfies" means is defined differently and will be explained later.

The language accepted by a deterministic automaton $\mathcal{A}$ is defined as

$$\mathcal{L}(\mathcal{A}) = \{\sigma \in \Sigma^{\omega} \mid \text{The run of } \mathcal{A} \text{ on } \sigma \text{ is accepting}\}.$$

For a nondeterministic automaton $\mathcal{A}$ we can have multiple or no runs, therefore the language accepted by $\mathcal{A}$ is defined as

$$\mathcal{L}(\mathcal{A}) = \{\sigma \in \Sigma^{\omega} \mid \text{There exists an accepting run of } \mathcal{A} \text{ on } \sigma\}.$$

Successor

Let $q \in Q$ be a state in an ω-automaton. Then we define the function $succ : Q \to 2^Q$ as

$$succ(q) = \{q' \in Q \mid \exists a \in \Sigma \ : \ q' \in \delta(q, a)\},$$

and a state in $succ(q)$ is called an *immediate successor* of q.

We now define the function $succ^*(q) : Q \to 2^Q$, giving us the set of states that are reachable from q, as

$$succ^*(q) = \{q' \in Q \mid \exists \sigma \in (\Sigma^* \setminus \{\varepsilon\}) \ : \ q' \in \delta(q, \sigma)\}.$$

The states in $succ^*(q)$ are called *successors* of q.

Propositional formulas on the edges

Sometimes, instead of using symbols $a \in 2^{\mathrm{AP}}$ in the transition function, it is more convenient to use propositional formulas over AP as the labels of the edges, representing possibly multiple elements of 2^{AP}. For example, for $\mathrm{AP} = \{\mathsf{a}, \mathsf{b}\}$ and $\Sigma = 2^{\mathrm{AP}}$, `true` would correspond to all the elements $\emptyset, \{\mathsf{a}\}, \{\mathsf{b}\}, \{\mathsf{a}, \mathsf{b}\}$ of 2^{AP}, the formula a would correspond to $\{\mathsf{a}\}$ and $\{\mathsf{a}, \mathsf{b}\}$ and the formula $\neg\mathsf{a} \wedge \mathsf{b}$ to the element $\{\mathsf{b}\}$.

These propositional formulas are used in this thesis for the edges of the graphical representations of automata. The conjunction $\vee$ is denoted by &, the negation $\neg$ by !.

[2] We use the abbreviation $\exists^{\omega}$ for "there are infinitely many". Formally, $\exists^{\omega} j : \phi(j) \equiv \forall i \exists j : j > i \wedge \phi(j)$.

2.4 Büchi Automata

There are many different types of acceptance conditions for ω-automata. We will first introduce Büchi acceptance, leading to nondeterministic and deterministic Büchi automata. Later, in Sections 2.5 and 2.6, we will introduce Rabin and Streett acceptance and the corresponding automata. For this thesis, we only need the deterministic variants of Rabin and Streett automata.

2.4.1 Nondeterministic Büchi Automata

Definition 2.4.1. A nondeterministic Büchi automaton (NBA) is a five-tuple $\mathcal{A}^{\text{NBA}} = (Q, \Sigma, \delta, q_0, F)$, where:

- Q is the set of states,
- Σ is the alphabet, in our case $\Sigma = 2^{\text{AP}}$,
- $\delta : Q \times \Sigma \to 2^Q$ is the transition function,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is the set of accepting states[3] (Büchi condition)

Definition 2.4.2 (Büchi acceptance). A run π of a nondeterministic Büchi automaton $\mathcal{A}^{\text{NBA}}$ with acceptance condition F is accepting iff $Inf(\pi) \cap F \neq \emptyset$, i.e. some states from F are visited infinitely often.

Therefore, a run in a nondeterministic Büchi automaton is accepting iff there is at least one run π that visits F infinitely often.

Büchi acceptance can be represented by the following LTL formula, where $\hat{F}$ is an atomic proposition that is true for a state q iff $q \in F$:

$$\square\lozenge\,\hat{F}$$

Sometimes NBA are defined to allow a set of initial states instead of a single start state. This can be easily converted to our definition.

2.4.2 Deterministic Büchi Automata

Deterministic Büchi automata (DBA) are defined like their nondeterministic counterparts, except that the transition function is deterministic:

- $\delta : Q \times \Sigma \to Q$.

[3] The accepting states are sometimes also called *final* or *fair* states.

Graphical Representation

The accepting states of the NBA are rectangular, the ”normal” states are circles. The start state q_0 is shaded grey.

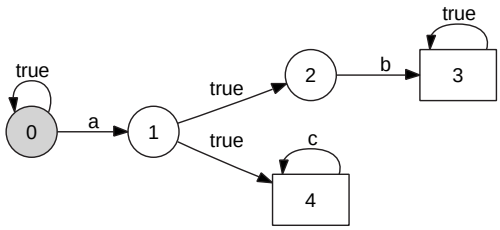


Figure 2.1: Graphical representation of a nondeterministic Büchi automaton

2.5 Deterministic Rabin Automata

Definition 2.5.1. A deterministic Rabin automaton (DRA) is a five-tuple $\mathcal{A}^{\mathrm{DRA}} = (Q, \Sigma, \delta, q_0, \Omega)$, where:

- Q is the set of states,
- Σ is the alphabet, in our case $\Sigma = 2^{\mathrm{AP}}$,
- $\delta : Q \times \Sigma \to Q$ is the transition function,
- $q_0 \in Q$ is the initial state,
- $\Omega = \{(L_1, U_1), \ldots, (L_r, U_r) \mid L_i, U_i \subseteq Q\}$ is the Rabin acceptance condition[4]

Definition 2.5.2 (Rabin acceptance). A run π of a Rabin automaton $\mathcal{A}$ with acceptance condition $\Omega = \{(L_1, U_1), \ldots, (L_r, U_r)\}$ is accepting iff there exists an $i \in \{1, \ldots, r\}$ such that $Inf(\pi) \cap L_i \neq \emptyset$ and $Inf(\pi) \cap U_i = \emptyset$.

In other words, an acceptance pair (L_i, U_i) is accepting when there is at least one state in L_i that is visited infinitely often in π and after some point no state in U_i is visited anymore in π. If at least one of the r pairs (L_i, U_i) is accepting, then the whole automata is accepting.

Rabin acceptance can be expressed as an LTL formula, with $\hat{L}_i, \hat{U}_i$ as atomic propositions that are `true` for a state q in the run iff $q \in L_i$ ($q \in U_i$):

$$\bigvee_{1 \leq i \leq r} (\Box \Diamond \hat{L}_i \wedge \neg \Box \Diamond \hat{U}_i) \quad = \quad \bigvee_{1 \leq i \leq r} (\Box \Diamond \hat{L}_i \wedge \Diamond \Box \neg \hat{U}_i)$$

[4] Another slightly different definition of the acceptance condition that is often used is defined as $\Omega = \{(E_1, F_1), \ldots, (E_r, F_r)\}$, with the role of U_i taken by E_i and that of L_i taken by F_i, i.e. the order of the two sets in the acceptance pairs is switched.

While the acceptance pairs contain sets of states, we are sometimes interested to know the acceptance pairs a specific state is a member of:

Definition 2.5.3 (Acceptance signature).
Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, \Omega = \{(L_1, U_1), \ldots, (L_r, U_r)\})$ be a deterministic Rabin automaton. Then, with $I = \{1, \ldots, r\}$ we define a function $acc : Q \to 2^I \times 2^I$ with

$$acc(q) = (\ \{i \in I \mid q \in L_i\}\ ,\ \{i \in I \mid q \in U_i\}\ ).$$

Informally, $acc(q)$ returns a pair of sets, the first set containing the indices i of the acceptance pairs (L_i, U_i) where q is in L_i, the second set containing the indices where q is in U_i.

Graphical Representation

Each state $q \in Q$ is represented by a box, containing an integer as an identifier for the state and a textual representation of the acceptance signature $acc(q)$, with '+i' used to signify $q \in L_i$ and '-i' for $q \in U_i$. Optionally, the states may contain additional information to show the internal representation used by the algorithm to create them. The graphical representation for them is explained with the description of the respective algorithm.

The start state q_0 is shaded grey.

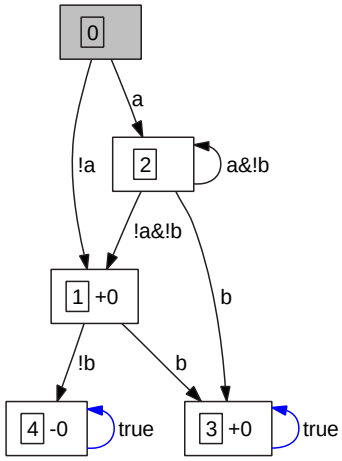


Figure 2.2: Deterministic Rabin Automaton for the LTL formula $\varphi = \mathtt{a\ U\ X\ b}$

2.6 Deterministic Streett Automata

Deterministic Streett Automata (DSA) are defined like deterministic Rabin Automata, they differ only in the semantics of the acceptance condition:

Definition 2.6.1 (Streett acceptance). A run π of a Streett automaton $\mathcal{A}$ with acceptance condition $\Omega = \{(L_1, U_1), \ldots, (L_r, U_r)\}$ is accepting iff for all $i \in \{1, \ldots, r\}$ $Inf(\pi) \cap L_i = \emptyset$ or $Inf(\pi) \cap U_i \neq \emptyset$.

In other words, a pair (L_i, U_i) is accepting if there is no state in L_i that occurs infinitely often in π or if there is a state in U_i that occurs infinitely often in π. If all of the r pairs (L_i, U_i) are accepting, then the whole automata is accepting.

As an LTL formula, Streett acceptance can be expressed as follows[5], with $\hat{L}_i, \hat{U}_i$ as atomic propositions that are `true` for a state q in the run iff $q \in L_i$ ($q \in U_i$):

$$\bigwedge_{1 \leq i \leq r} (\neg \Box \Diamond \hat{L}_i \vee \Box \Diamond \hat{U}_i) \quad = \quad \bigwedge_{1 \leq i \leq r} (\Box \Diamond \hat{L}_i \rightarrow \Box \Diamond \hat{U}_i)$$

The acceptance signature for states of Streett automata is defined as for Rabin automata.

2.6.1 Duality of DRA and DSA

Looking at the formulas for Rabin and Streett acceptance, we can see that they are dual, i.e. Rabin acceptance is the negation of Streett acceptance and vice versa. Therefore, if we have a deterministic Streett automaton $\mathcal{A}^{DSA}$ and a deterministic Rabin automaton $\mathcal{A}^{DRA}$ with exactly the same transition structure and the same set of acceptance pairs (L_i, U_i), then $\mathcal{A}^{DSA}$ accepts the complement language of $\mathcal{A}^{DRA}$ (e.g. [Löd98]):

$$\mathcal{L}(\mathcal{A}^{DSA}) = \overline{\mathcal{L}\left(\mathcal{A}^{DRA}\right)}$$

This provides a very simple method for complementing Rabin and Streett automata. One possible approach to complement nondeterministic Büchi automata using Safra's construction is as follows[6]:

$$\text{NBA} \xrightarrow{\text{Safra's construction}} \text{DRA} \xrightarrow[\text{=complementation}]{\text{regard as Streett}} \text{DSA} \xrightarrow[\text{(Section 2.8.3)}]{\text{convert to Büchi}} \overline{\text{NBA}}$$

Size of Streett and Rabin automata for the same language Complementation of NBA can require an exponential blowup, which is also true for the conversion from deterministic Streett to nondeterministic Büchi automata:

[5] Streett acceptance is essentially a strong fairness condition.

[6] More efficient complementation procedures for NBA that do not require determinization were developed since the introduction of Safra's construction, for example [FKV04].

Proposition 2.6.1 ([Saf89] Lemma 2.3). For every $n > 0$ there exists a deterministic Streett automaton with $3n$ states and $2n$ accepting pairs, such that any equivalent nondeterministic Büchi automaton has at least 2^n states. □

It follows from this proposition and the fact that there exists a polynomial conversion from deterministic Rabin automata to nondeterministic Büchi automata (see Section 2.8.2) that for some languages deterministic Streett automata are exponentially more compact than deterministic Rabin automata. Because of the duality of Streett and Rabin automata, it follows that for some languages the reverse is true and for them deterministic Rabin automata are exponentially more compact than deterministic Streett automata.

2.7 Expressiveness

It is well known that nondeterministic Büchi, deterministic Rabin and deterministic Streett automata all have the same expressive power. They can all express the ω-regular languages (e.g. [Tho90]), defined by an extension of regular expressions on finite words to infinite words, having the operators $\cdot$ (concatenation), $+$ (alternation), * (Kleene star) and adding the $^\omega$ operator:

ω-regular languages An *ω-regular expression* is an expression of the form

$$\alpha_1 \cdot \beta_1^\omega + \ldots + \alpha_n \cdot \beta_n^\omega,$$

where, for all i, α_i and β_i are regular expressions on finite words (such that the empty word $\varepsilon \notin \mathcal{L}(\beta_i)$). The languages recognizable by ω-regular expressions are called *ω-regular*.

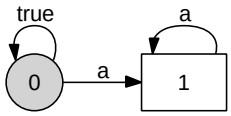
Interestingly, for the Rabin and Streett acceptance condition, the nondeterministic automata are not more expressive than their deterministic counterparts. But deterministic Büchi automata are less expressive than nondeterministic Büchi automata.

2.7.1 Expressiveness of deterministic Büchi automata

Proposition 2.7.1 ([Var96]). Deterministic Büchi automata can not express all ω-regular languages.

Proof. We show this by presenting a language $\mathcal{L}$ that can be expressed by a nondeterministic Büchi automaton $\mathcal{A}$ but not by a deterministic Büchi automaton. The language corresponds to the LTL formula[7] $\Diamond\Box \mathsf{a}$, which can be represented by a nondeterministic Büchi automaton $\mathcal{A}$ like this:

[7] This type of formula is called *persistence formula*.



As $\mathrm{AP} = \{\mathsf{a}\}$ and $2^{\mathrm{AP}} = \{\emptyset, \{\mathsf{a}\}\}$, we will use the shorthand $\mathsf{1}$ for $\{\mathsf{a}\}$ and $\mathsf{0}$ for $\emptyset$.

Assume that there is a deterministic Büchi automaton $\mathcal{A}' = (Q, \Sigma, \delta, q_0, F)$ that recognizes the same language as the NBA $\mathcal{A}$. Consider the infinite word $\sigma_0 = \mathsf{1}^\omega$, which is clearly accepted by $\mathcal{A}'$ and therefore has a finite prefix u_0 such that $\delta(q_0, u_0) \in F$. Then, the infinite word $\sigma_1 = u_0 \cdot \mathsf{0} \cdot \mathsf{1}^\omega$ is also accepted by $\mathcal{A}'$ and has a finite prefix $u_0 \cdot \mathsf{0} \cdot u_1$ such that $\delta(q_0, u_0 \cdot \mathsf{0} \cdot u_1) \in F$. We can continue to get finite words u_i with $\delta(q_0, u_0 \cdot \mathsf{0} \cdot u_1 \cdot \mathsf{0} \cdot \ldots \cdot \mathsf{0} \cdot u_i) \in F$. Since $\mathcal{A}'$ has only a finite number of states Q, there are $0 \leq i \leq j$ such that $\delta(q_0, u_0 \cdot \mathsf{0} \cdot u_1 \cdot \mathsf{0} \cdot \ldots \cdot \mathsf{0} \cdot u_i) = \delta(q_0, u_0 \cdot \mathsf{0} \cdot u_1 \cdot \mathsf{0} \cdot \ldots \cdot \mathsf{0} \cdot u_i \cdot \mathsf{0} \cdot \ldots \cdot \mathsf{0} \cdot u_j)$ and therefore $\mathcal{A}'$ has an accepting run on the word

$$\sigma = u_0 \cdot \mathsf{0} \cdot u_1 \cdot \mathsf{0} \cdot \ldots \cdot \mathsf{0} \cdot u_i \cdot (\mathsf{0} \cdot \ldots \cdot \mathsf{0} \cdot u_j)^\omega.$$

But σ has infinitely many $\mathsf{0}$ and therefore $\sigma \notin \mathcal{L}(\mathcal{A})$, which is a contradiction to the initial assumption. □

2.7.2 (Co-)Safety languages

An important subset of the languages that can be expressed with deterministic Büchi automata are the safety and co-safety languages:

Definition 2.7.1 (Safety languages[8]). Let $\mathcal{L} \in \Sigma^\omega$ be a language on infinite words over an alphabet Σ. Then a finite word $x \in \Sigma^*$ is called a *finite bad prefix* for $\mathcal{L}$, if for every $y \in \Sigma^\omega$ the concatenation $x \cdot y \notin \mathcal{L}$. If all words $w \in \Sigma^\omega \setminus \mathcal{L}$ (all words not in $\mathcal{L}$) have a finite bad prefix, $\mathcal{L}$ is called a *safety language*. We denote the set of bad prefixes of a language $\mathcal{L}$ by $\mathit{badpref}(\mathcal{L})$.
If for a language $\mathcal{L}$, the complement language $\overline{\mathcal{L}} = \Sigma^\omega \setminus \mathcal{L}$ is a safety language, then $\mathcal{L}$ is called a *co-safety language*[9].
If for an LTL formula φ, $\mathcal{L}(\varphi)$ is a safety language, then φ is called a *safety formula*. If $\mathcal{L}(\varphi)$ is a co-safety language, then φ is called a *co-safety formula*.

As $\mathit{badpref}(\mathcal{L}) \in \Sigma^*$, it can be recognized by nondeterministic and deterministic finite automata.

[8] In the topological Borel hierarchy, safety properties correspond to the class F, co-safety to the class G and deterministic Büchi automata to the class G_δ (e.g. [Tho90]).

[9] Co-safety properties are often also called *guarantee properties*, because they informally "guarantee that something good will happen" (e.g. [MP90]).

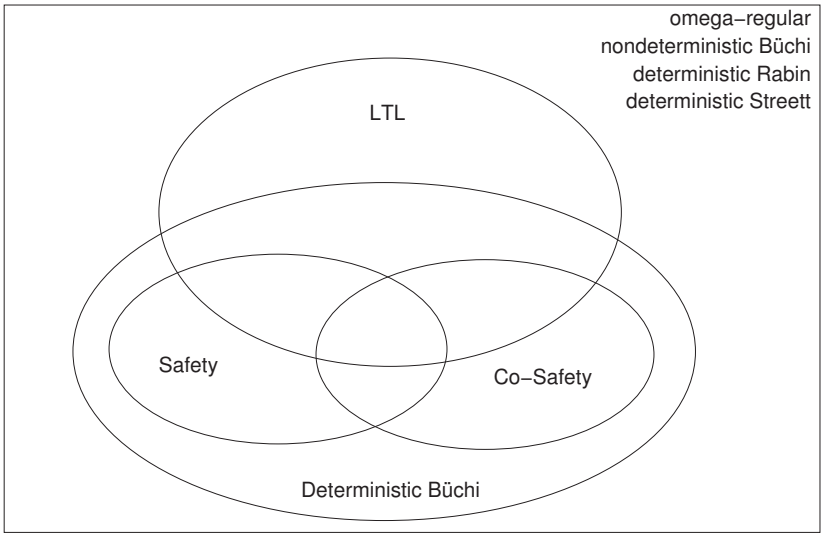
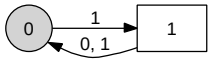


Figure 2.3: Language inclusion diagram for the presented automata and LTL

2.7.3 Expressiveness of LTL

The languages expressible by LTL formulas form another subset of the ω-regular languages, they correspond to the *star-free ω-regular languages* (e.g. [Tho90]). For example, the language "1 at every other index" recognized by the ω-regular expression $(1{\cdot}(0{+}1))^\omega$ can not be expressed by an LTL formula [Wol82], while the following NBA recognizes this language:



2.8 Simple transformations between ω-automata

This section presents conversions from DBA to DRA and to DSA, and from DRA and DSA back to nondeterministic Büchi automata.

2.8.1 Deterministic Büchi to Rabin and Streett

If we have a deterministic Büchi automaton $\mathcal{A}^{Büchi} = (Q, \Sigma, \delta, q_0, F)$, we can easily create an equivalent deterministic Rabin or Streett automaton on the same transition structure with one acceptance pair:

- Rabin: $\mathcal{A}^{Rabin} = (Q, \Sigma, \delta, q_0, \Omega = \{L, U\})$, with $L = F$ and $U = \emptyset$.
- Streett: $\mathcal{A}^{Streett} = (Q, \Sigma, \delta, q_0, \Omega = \{L, U\})$, with $L = Q$ and $U = F$.

2.8.2 Deterministic Rabin to nondeterministic Büchi

The basic idea (e.g. [Löd98]) to generate an equivalent nondeterministic Büchi automaton from a deterministic Rabin automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, \Omega)$, with acceptance condition $\Omega = \{(L_1, U_1), \ldots, (L_r, U_r)\}$, is to take the original transition structure and generate r additional copies of the automaton, one for each Rabin acceptance pair. At each state in the original automaton, the NBA can nondeterministically choose the acceptance pair i which will ultimately be accepting and switch to its copy and stay there. In this copy, the states in U_i have no outgoing transitions. This ensures that no state in U_i can be visited infinitely often. The states in L_i are accepting states in the copy for i. This ensures that at least one of them is visited infinitely often in an accepting run.

Define the nondeterministic Büchi automaton $\mathcal{A}' = (Q', \Sigma, \delta', q_0, F')$ as follows:

- $Q' = Q \cup (Q \times \{1, \ldots, r\})$
- For $a \in \Sigma$, $q \in Q$ and $k \in \{1, \ldots, r\}$ define

$$\begin{aligned} \delta'(q, a) &= \delta(q, a) \cup \{(p, j) \mid p \in \delta(q, a) \wedge 1 \leq j \leq r\} \\ \delta'((q, k), a) &= \begin{cases} \emptyset & \text{if } q \in U_k \\ \{(p, k) \mid p \in \delta(q, a)\} & \text{otherwise} \end{cases} \end{aligned}$$

- $F' = \bigcup_{i=1}^{r} L_i \times \{i\}$

For a Rabin automaton with n states and r acceptance pairs, the resulting Büchi automaton has $\mathcal{O}(n \cdot r)$ states.

Note To remove some unnecessary nondeterminism in the NBA, we can change the transition function $\delta'(q, a)$ to only go to a copy j if the node $q \in L_j$ instead of for all nodes q. If we do this, we have to verify that the destination node p in copy j is not in U_j:

$$\delta'(q, a) = \delta(q, a) \cup \{(p, j) \mid p \in \delta(q, a) \wedge (1 \leq j \leq r) \wedge (q \in L_j \wedge p \notin U_j)\}$$

2.8.3 Deterministic Streett to nondeterministic Büchi

The transformation (e.g. [Löd98]) from deterministic Streett automata to nondeterministic Büchi automata is more complex; in fact, because of Proposition 2.6.1, we cannot expect a polynomial transformation from DSA to NBA.

Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, \Omega)$ be a deterministic Streett automaton, with acceptance condition $\Omega = \{(L_1, U_1), \ldots, (L_r, U_r)\}$. For every accepting run $\pi = \pi_0, \pi_1, \ldots$ of $\mathcal{A}$ there exists a k such that for every $l_1 > k$ with $\pi_{l_1} \in L_i$ there exists a $l_2 \geq l_1$ with $\pi_{l_2} \in U_i$.

The Büchi automaton operates in two modes: In the first (starting) mode, it just mimics the transitions in the deterministic Streett automaton. At any point it can nondeterministically guess that k has been reached and switch to the second mode. Here, besides mimicking the original Streett automaton, it keeps track of the indices i, j of the visited L_i and U_j in two sets I and J. Every time $I \subseteq J$, i.e. for every visited L_i, the corresponding U_i was also visited, we reset $I = J = \emptyset$. If I and J are reset infinitely often to $\emptyset$, then the run is accepting.

Define the nondeterministic Büchi automaton $\mathcal{A}' = (Q', \Sigma, \delta', q_0, F')$ as follows:

- $Q' = Q \cup (Q \times 2^{\{1,\ldots,r\}} \times 2^{\{1,\ldots,r\}})$
- For $a \in \Sigma$, $q \in Q$, $I, J \subseteq \{1, \ldots, r\}$ and $I' = I \cup \{i : q \in L_i\}, J' = J \cup \{j : q \in U_j\}$ define

$$\begin{aligned} \delta'(q,a) &= \delta(q,a) \cup \{(p, \emptyset, \emptyset) \mid p \in \delta(q,a)\} \\ \delta'((q,I,J),a) &= \begin{cases} \{(p, I', J') : p \in \delta(q,a)\} & \text{if } I' \not\subseteq J' \\ \{(p, \emptyset, \emptyset) : p \in \delta(q,a)\} & \text{if } I' \subseteq J' \end{cases} \end{aligned}$$

- $F' = Q \times \{\emptyset\} \times \{\emptyset\}$

For a Streett automaton with n states and r acceptance pairs, the resulting Büchi automaton has $n \cdot 2^{\mathcal{O}(r)}$ states.

3 Safra's Construction

Safra [Saf89] proposed a construction to convert a nondeterministic Büchi automaton into an equivalent deterministic Rabin automaton. We will first explore the motivation behind this construction, then describe the construction formally and investigate its complexity.

3.1 Motivation

Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be a nondeterministic Büchi automaton we want to determinize. The natural starting point in the search for a determinization construction for automata on infinite words is surely to consider the determinization of automata on *finite* words, the classical powerset construction. There, the states of the deterministic automaton are sets of states $\hat{Q} \in 2^Q$ of the nondeterministic automaton, and the new transition function δ' is defined as the union of the immediate successors of the states in $\hat{Q}$:

$$\delta'(\hat{Q}, a) = \bigcup_{q \in \hat{Q}} \delta(q, a)$$

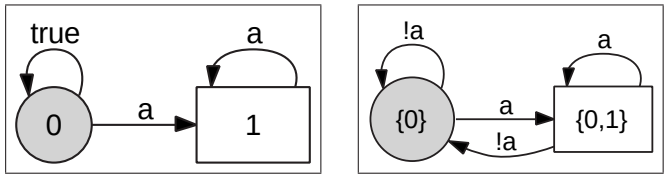
The accepting states of the deterministic automaton are the states $\hat{Q}$, where at least one state $q \in \hat{Q}$ is an accepting state in the nondeterministic automaton:

$$F' = \{\hat{Q} \mid \exists q \in \hat{Q} : q \in F\}$$

This construction for automata on finite words can also be used for nondeterministic Büchi automata, which would result in deterministic Büchi automata. As we have seen in Section 2.7.1, deterministic Büchi automata are not as expressive as nondeterministic Büchi automata and therefore it is guaranteed that the classical powerset construction will not work correctly for some NBA. We will consider an example where the powerset construction has to fail, the same used to show that DBA are less expressive, and we will examine why it fails.

Consider the nondeterministic Büchi automaton $\mathcal{A}$ for $\mathcal{L}(\Diamond\Box\,\mathsf{a})$ in Figure 3.1 a). Applying the powerset construction results in the deterministic Büchi automaton $\mathcal{A}'$ shown in Figure 3.1 b), having two states, $\{0\}$ and $\{0,1\}$, where the state $\{0,1\}$ is accepting because state 1 in the original automaton is accepting.

It is clear that $\mathcal{L}(\mathcal{A}') \neq \mathcal{L}(\mathcal{A})$, for example the infinite word $\sigma = (\{\mathsf{a}\}\cdot\emptyset)^\omega$ (a and $\neg\mathsf{a}$ alternating infinitely often) is accepted by the deterministic automaton $\mathcal{A}'$ but rejected



(a) Nondeterministic Büchi (b) Powerset construction

Figure 3.1: NBA for $\mathcal{L}(\Diamond\Box \mathsf{a})$ and result of the classical powerset construction

by the nondeterministic automaton $\mathcal{A}$. The problem is easily identified: For σ, there are infinitely many runs in the nondeterministic Büchi automaton $\mathcal{A}$ that visit the accepting state 1, *but then abort*, i.e. are of finite length and can therefore not be accepting. In the deterministic automaton, these finite runs are not detected and thus not rejected.

Safra's idea was to use multiple powerset constructions in parallel to track the runs originating in accepting states in addition to the classical powerset construction, which allows to detect when these runs are finite and need to be rejected. These different powersets are organized in a tree-structure called *Safra trees* and the states of the deterministic ω-automaton are not simply sets of states as in the classical powerset construction but Safra trees.

These trees consist of nodes that have a *name*, which allows us to refer them and keep track of their existence over multiple trees, and a *label*, a set of states from the original NBA associated with this node. The transition function will transform a Safra tree to its successor by separately applying the powerset construction to the labels of every node of the tree. The initial tree will have only a root node with $\{q_0\}$ as its powerset, therefore the label of the root node in all trees will correspond to the classical powerset construction.

As we want to keep track of runs originating from accepting states, we create a new child for every node that contains an accepting state in its label. The label of the newly branched child consists of all the accepting states from the parent's label. If at a future point this node has an empty label, meaning that the runs it tracked were finite, we can remove the node and record in the acceptance condition that these runs should be rejected (Figure 3.2).

We now face the problem that, because there is no limit on the branching of new nodes, the trees can grow infinitely large. To get finite trees, we therefore have to bound both height and width of the trees. The width can be limited by the realization that it is not necessary that a state appears in the labels of multiple siblings, it is sufficient that only one of the siblings tracks the corresponding run. To have a well defined rule which sibling is chosen to keep such a state, Safra proposes an ordering on the siblings in the tree by considering their "age", i.e. the relative order of their creation. If two siblings

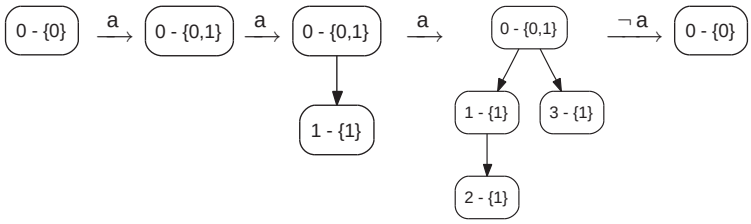


Figure 3.2: New nodes are created to keep track of runs originating from accepting states, nodes with empty labels are deleted.

share a state in their labels, the older sibling keeps the state and the state is removed from the younger sibling. By applying this operation, we can ensure that the labels of the sibling nodes are disjunct.

To bound the height, we notice that the union of the labels of the children of a node in a Safra tree is always a subset of the label of the parent node, as they track a subset of runs that the parent tracks. When a parent and its child have exactly the same labels, they both track the same runs, which is redundant and we can remove the child node. To keep track when this happens, every node has a flag (graphically represented by a '!') that is set when the children of a node are removed like this.
As the states of the parent's label can be distributed over multiple children, we extend this operation to handle the more general case: If the label of the parent equals the union of the labels of its children, then remove all children (and their children) and mark the parent node.
After this operation, the union of the labels of the children are a proper subset of the label of the parent node, which bounds the height of the tree, as the states that can appear in the labels are finite and every additional level of the tree has to have at least one state less in the labels of the children.

Now that siblings have disjunct labels and a parents label is a proper superset of the labels of its children, the Safra trees are bounded both in height and width and are therefore finite. We will later prove that for an NBA with n states all Safra trees will have at most n nodes and that it is sufficient to have $2n$ different node names, so we take the node names from $\{1, \ldots, 2n\}$.

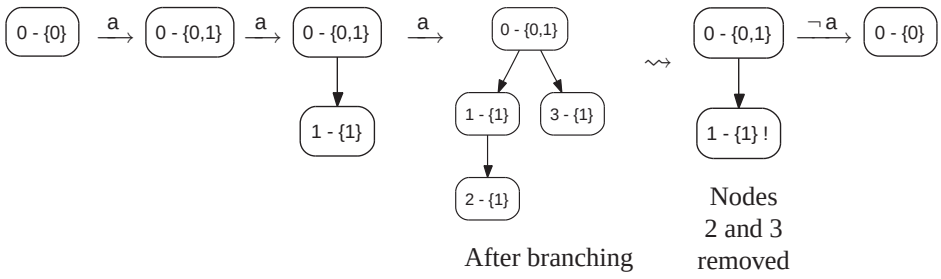


Figure 3.3: Example of Safra's construction. The fourth and fifth tree show the effect of the two operations that limit the height and width of the trees.

Now we have to consider the acceptance condition. We have seen that Büchi acceptance is not enough for deterministic ω-automata, so we consider Rabin acceptance, by connecting one Rabin acceptance pair (L_i, U_i) with each node of the Safra trees. If a node named i is deleted because its label becomes empty, the Safra tree is put into U_i, as the runs tracked by node i were finite and we have to reject them. On the other hand, if a node i is marked, the Safra tree is placed in L_i.

We will now formally define Safra trees and Safra's construction and prove its correctness.

3.2 Safra trees[1]

Let Q be a nonempty, finite set, with $|Q| = n$. Let T be a finite, ordered tree with nodes N and the following structure:

- Every node of the tree has a unique name from the set $\{1, \ldots, 2n\}$
  $name : N \to \{1, \ldots, 2n\}$
- Every node is labeled with a member of 2^Q
  $label : N \to 2^Q$.
- Every node is either marked or unmarked
  $marked : N \to \{\texttt{true}, \texttt{false}\}$

We denote by $root(T)$ the root node of tree T, by $parent(n)$ the parent of a node n (with $parent(root(T)) = \perp$) and by $child_i(n)$ the i-th child of a node n.

To be a valid Safra tree, T has to satisfy the following constraints:

1. No node has an empty label: $label : N \to 2^Q \setminus \{\emptyset\}$
2. The label of a parent node is a proper superset of the union of the labels of the children.
3. The labels of sibling nodes are disjoint.

Lemma 3.2.1. A Safra tree over a set Q with $|Q| = n$ has at most n nodes.

Proof. We proof the lemma by induction over the depth of the tree.

If the depth is 1, the tree consists only of the root node.

Let $h + 1$ be the depth of the tree. Let $i_1, \ldots, i_k$ be the children of the root node with labels $Q_1, ..., Q_k$. For $j = 1, \ldots, k$, the subtree with i_j as root is a Safra tree over Q_j with a depth of at most h. By the induction hypothesis this subtree has at most $n_j = |Q_j|$ nodes. The Q_j are disjoint and their union is a proper subset of Q. Therefore $\Sigma_{j=1}^{k} n_j < n$. This means that the number of nodes of the whole Safra tree is at most $1 + \Sigma_{j=1}^{k} n_j < 1 + n \leq n$. □

[1] This and the next two sections roughly follow the presentation of Safra's construction in [Löd98].

3.3 Constructing the deterministic Rabin automaton

Given a nondeterministic Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, \Omega)$ with n states, we construct a deterministic Rabin automaton $\mathcal{A}' = (Q', \Sigma, \delta', q_0', \Omega')$ as follows:

- $Q' =$ The set of all Safra trees over Q.
- $q_0' =$ The Safra tree with only one node, named 1, labeled with $\{q_0\}$ and unmarked.
- The transition function δ' transforms a Safra tree into its successor given input $a \in \Sigma$ by the following procedure:
  1. **Unmark**
     Set $marked(n) =$ `false` for all nodes n.
  2. **Branch accepting**
     For every node n with $label(n) \cap F \neq \emptyset$, create a new node as the youngest child of n labeled with $label(n) \cap F$. The new node is named with an unused name from $\{1, \ldots, 2n\}$ (see explanation below).
  3. **Powerset**
     For every node n, replace $label(n)$ with $\bigcup_{q \in label(n)} \delta(q, a)$.
  4. **Normalize siblings**
     For every two sibling nodes such that they share a $q \in Q$ in their labels, remove q from the label of the youngest node and all its children.
  5. **Remove empty**
     Remove all nodes with empty labels.
  6. **Mark**
     For every node whose label equals the union of the labels of its children, remove all descendants of this node and mark it.
- The acceptance condition $\Omega' = \{(L_1, U_1), \ldots, (L_{2n}, U_{2n})\}$ consists of $2n$ acceptance pairs, defined as
  - $L_i =$ set of all Safra trees with node i marked
  - $U_i =$ set of all Safra trees without node i

During the calculation of the successor with the transition function δ' the intermediate tree may be not a valid Safra tree. Steps 4, 5 and 6 reestablish the required structure.

It is clearly sufficient to just generate the Safra trees as states of the DRA that are actually reachable from the initial Safra tree q_0'.

There are enough unused names New nodes in the tree can only be added in step 2. To ensure that there are enough unused names for the new nodes, the names are from the set $\{1, \ldots, 2n\}$. As shown in Lemma 3.2.1, the original Safra tree can have at most n nodes and in step 2 for every node at most one child can be added. As a consequence, the tree after step 2 can have at most $2n$ nodes, so that there are enough names for the nodes.

Empty tree? The Safra tree used as the initial state of the resulting deterministic Rabin automaton has only the root node, with name 1, and labeled with the initial state of the Büchi automaton. By applying the transition function, the label of the root node always represents the power set construction. The root node can only be deleted when the states in the label do not have any successors for all possible inputs $a \in \Sigma$ in the Büchi automaton. In that case, the label after step 3 is the empty set and the root node is deleted in step 5, resulting in the empty tree. To ensure that the resulting deterministic Rabin automaton is complete, i.e. that for every input $a \in \Sigma$ there is a successor state, the transition function δ' outputs the empty tree as the successor for the empty tree and any possible input from Σ, resulting in a loop that is not accepting.

Graphical representation The Safra trees shown in graphics of DRA are represented using a table-like tree structure, as shown in Figure 3.4. For technical reasons the node names are from the set $\{0, \ldots, 2n-1\}$ instead of from $\{1, \ldots, 2n\}$.

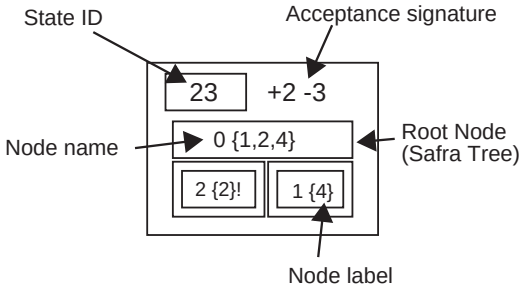


Figure 3.4: Graphical representation of Safra trees in the states of DRA.

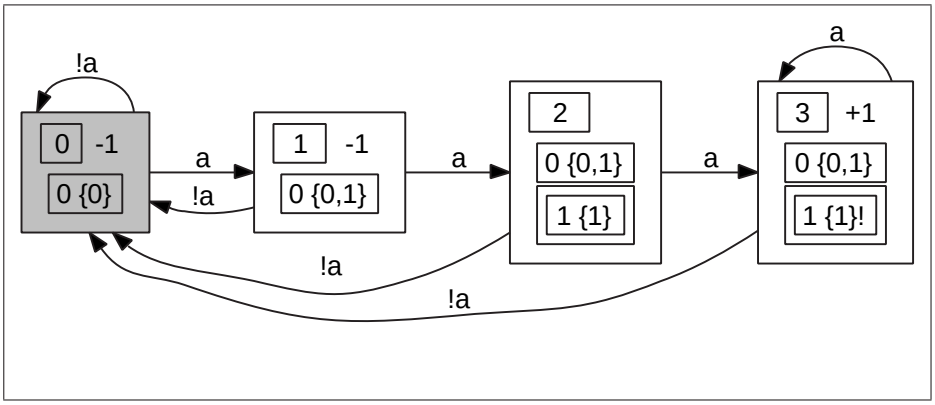


Figure 3.5: The deterministic Rabin automaton generated from the automaton from Figure 3.1 a) for LTL formula $\Diamond\Box$ a by Safra's construction.

Theorem 3.3.1. Correctness. Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be a nondeterministic Büchi automaton with n states and $\mathcal{A}' = (Q', \Sigma, \delta', q_0', \Omega)$ the deterministic Rabin automaton constructed from $\mathcal{A}$ by the procedure detailed above. Then $\mathcal{A}$ and $\mathcal{A}'$ are equivalent, i.e. $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$.

Proof. The proof consists of two steps, first showing $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}')$ and then showing $\mathcal{L}(\mathcal{A}') \subseteq \mathcal{L}(\mathcal{A})$.

$\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}')$:
Let $\sigma \in \mathcal{L}(\mathcal{A})$ be an infinite word and let $\pi \in Q^\omega$ be an accepting run for σ of the nondeterministic Büchi automaton $\mathcal{A}$. We have to show that the corresponding run $\pi' \in Q'^\omega$ in the deterministic Rabin automaton $\mathcal{A}'$ is accepting. For π' to be accepting, one of the acceptance pairs (L_i, U_i) has to be satisfied, i.e. from some point on trees from L_i are visited infinitely often and no tree from U_i is visited. This means we have to find a node i in the Safra trees that is marked infinitely often (so the trees are included in L_i) and is never removed (so the trees will not be in U_i).
Because the label of the root node always contains the state from π, the label of the root node is never empty and will never be removed. So if the root node is marked infinitely often, we are done.
Otherwise, we consider the next time π visits an accepting state in the NBA after the last time the root node was marked. In step 2 of the transition function, a new child for the root node is branched. From now on, the state of π is always included in one of the labels of the children of the root node. The state from π can only be moved to an older sibling, but as there are only finitely many siblings, at some point it will forever stay in one sibling, which will therefore never be empty and not be removed. If this node is marked infinitely many times, we are done. If not, we can apply the same argument and will find a node at the next level that contains the state of π and will never be removed. As the depth of the Safra tree is finite and bound by n, we will eventually find a node that is marked infinitely often and will never be removed.

$\mathcal{L}(\mathcal{A}') \subseteq \mathcal{L}(\mathcal{A})$:
Let $\sigma \in \mathcal{L}(\mathcal{A}')$ be an infinite word and let $\pi' = \pi_0', \pi_1', \pi_2', \cdots \in Q'^\omega$ be the accepting run for σ of the deterministic Rabin automaton $\mathcal{A}'$. For π' to be accepting, there has to be some node n that occurs from some point on in all Safra trees of π' and is marked infinitely often. Let x be the position such that the node n occurs in every tree π_y', with $y \geq x$. Now let $x \leq m_1, m_2, \ldots$ be the positions in π' where node n is marked and let $Q_1, Q_2, \ldots$ be the corresponding labels of node n at these positions. Additionally, set $m_0 = 0$ and $Q_0 = \{q_0\}$.
For a node to be marked, the construction guarantees that for each $q \in Q_j + 1$, $j \geq 0$, there exists at least one $q' \in Q_j$ such that there exists a run in $\mathcal{A}$ over $\sigma[m_j, m_{j+1})$ which leads from q' to q and visits at least one accepting state:

Index in σ :	m_j		i		m_{j+1}
$label(n)$:	Q_j	$\xrightarrow{\sigma[m_j,i)}$	Q'_i	$\xrightarrow{\sigma[i,m_{j+1})}$	Q_{j+1}
			$\cup\mid$		$\mid\mid$
$label(m)$:			$C'_i = Q_i \cap F$	$\xrightarrow{\sigma[i,m_{j+1})}$	$C'_{m_{j+1}} = Q_{j+1}$
			Branch child		Mark

Consider the simple case with only one child node: For a node n that was marked at index m_j in input σ with label Q_j to be marked again at some later point m_{j+1}, there has to be an index $m_j \leq i \leq m_{j+1}$ where a child node m with label C'_i was branched. For this to happen, the label Q'_i at this point had to contain at least one accepting state from the NBA. These runs originating in an accepting state are tracked in the child node m. For n to be marked at point m_{j+1}, the label Q_{j+1} and the label $C_{m_{j+1}}$ of the child node m have to be the same. This allows us to construct for every state in Q_{j+1} a path back to a state in Q_j which visited at least one accepting state in between.
If the tracked runs are spread over multiple child nodes and move between them, the situation is slightly more complex, but the same argument can be used.

Using these path segments, we now have to construct an infinite path in the NBA, which we do by applying König's Lemma: *Any tree with infinitely many nodes, in which every node has only finitely many offsprings, contains an infinite path.*
We do this by constructing a tree, whose nodes are all pairs of the form (q, j) for $q \in Q_j$. As the parent for a node $(q, j+1)$ with $q \in Q_{j+1}$, we pick a node (q', j) with $q' \in Q_j$, so that there exists a run from q' to q. It is clear that every level of the tree has at most n nodes and there exist only edges between adjacent levels. Therefore every node has only finitely many offsprings. It is also clear that every node (q, j) has a unique path from $(q_0, 0)$ to (q, j) and therefore the tree is well formed. By König's Lemma, there has to be an infinite path $(q_0, 0), (q_1, 1), (q_2, 2), \ldots$ in the tree. It follows from the design of the tree that with these $q_0, q_1, q_2, \ldots$ we can construct an infinite path in the NBA $\mathcal{A}$ that is accepting. □

3.4 Complexity

Acceptance conditions The number of pairs in the acceptance condition of the resulting deterministic Rabin automaton $\mathcal{A}'$ is bound by $2n$, the number of different node names.

States in the DRA The number of states of $\mathcal{A}'$ is bound by the number of different Safra trees over Q, which can be demonstrated to be bound by $2^{\mathcal{O}(n \cdot \log n)}$ by representing the Safra trees as sets of functions, and by counting all the different possible combinations to get an upper bound on the number of Safra trees:

1. A node n is called characteristic for a state $q \in Q$, iff q occurs in the label of n but not in the labels of any of its children. For any state q appearing in the label of the root node, there is exactly one node in the tree which is characteristic for q. The states not appearing in the root label have no characteristic node. This relation can be described by the mapping $Q \to \{1, \ldots, 2n, \bot\}$, which can have $(2n+1)^n$ different values and allows the reconstruction of the labels in a Safra tree, given its structure.

2. A mapping to determine if a node is marked or not: $\{1, \ldots, 2n, \} \to \{0, 1\}$, which can have 2^{2n} different values.

3. A mapping $\{1, \ldots, 2n\} \to \{1, \ldots, 2n, \bot\}$ for the parent relation between the nodes, which can have $(2n+1)^{2n}$ different values.

4. A mapping $\{1, \ldots, 2n\} \to \{1, \ldots, 2n, \bot\}$ to describe the ordering of the sibling nodes, which can also have $(2n+1)^{2n}$ different values.

With these results, we can bound the number of Safra trees and thus the number of states in the DRA as follows:

$$\begin{aligned} Q' &\leq (2n+1)^n \cdot 2^{2n} \cdot (2n+1)^{2n} \cdot (2n+1)^{2n} \\ &\leq (2n+1)^{7n} \\ &= 2^{\log((2n+1)^{7n})} \\ &= 2^{7n \cdot \log(2n+1)} \\ &\in 2^{\mathcal{O}(n \cdot \log n)} \end{aligned}$$

3.4.1 Optimality of Safra's construction – A lower bound for NBA→DRA

A lower bound – originally due to Max Michel in an unpublished manuscript – is presented in [Löd98].

Proposition 3.4.1 ([Löd98], Section 2.2). There exists a family of languages $\mathcal{L}_n$ which can be represented by nondeterministic Büchi automaton with $\mathcal{O}(n)$ states but which can not be represented by a deterministic Rabin automaton with less than $n!$ states. □

As $n! \in 2^{\Omega(n \cdot \log n)}$, it follows that Safra's construction with its $2^{\mathcal{O}(n \cdot \log n)}$ upper bound is optimal up to a constant factor in the exponent.

3.4.2 A lower bound for LTL→DRA

In [KV98] (Theorem 5.2), a lower bound for the translation from LTL to deterministic Büchi automata is provided.

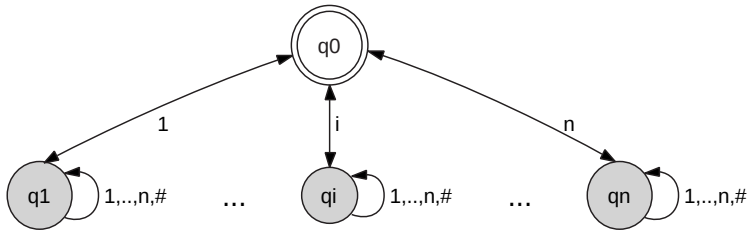


Figure 3.6: The nondeterministic Büchi Automaton $\mathcal{A}_n$ representing $\mathcal{L}_n$. A deterministic Rabin automaton for $\mathcal{L}_n$ has at least $n!$ states.

They define a family of finite languages $\mathcal{L}_n \subseteq \Sigma^*$, with alphabet $\Sigma = \{0, 1, \#, \$\}$ as follows:

$$\mathcal{L}_n = \{x \cdot \# \cdot w \cdot \# \cdot y \cdot \$ \cdot w \quad \mid x, y \in \{0,1,\#\}^*,\ w \in \{0,1\}^n\}$$

Informally, the words in $\mathcal{L}_n$ first consist of sequences from $\{0,1\}^*$, separated by #, until \$ is read. Then, there is a sequence $w \in \{0,1\}^n$ of length n that starts after \$ and this sequence w already occurred in front of the \$. By [CKS81], the smallest deterministic automaton on finite words that accepts $\mathcal{L}_n$ has at least 2^{2^n} states, as it has to "remember" the set of sequences $\{0,1\}^n$ that occurred before \$.

Ignoring the technical fact that LTL and deterministic Büchi automata operate on infinite languages, $\mathcal{L}_n$ can be expressed by an LTL formula φ_n:

$$\begin{aligned}\varphi_n \quad = \quad & (\neg\$ \ \mathtt{U}\ (\$ \wedge \mathtt{X}\,\Box\neg\$)) \wedge \\ & \Diamond[\# \wedge (\mathtt{X}^{n+1}\,\#) \wedge \bigwedge_{1\le i\le n} ((\mathtt{X}^i\,0 \wedge \Box(\$ \rightarrow \mathtt{X}^i\,0)) \vee (\mathtt{X}^i\,1 \wedge \Box(\$ \rightarrow \mathtt{X}^i\,1)))]\end{aligned}$$

The first part of the formula guarantees that there is exactly one occurrence of \$, while the second part guarantees the occurrence of #w# with $w \in \{0,1\}^n$ where w is repeated after \$.

The size of the formula $|\varphi_n| \in \mathcal{O}(n^2)$ is quadratic in n and the deterministic ω-automaton has, as noted before, at least 2^{2^n} states. Because the argumentation is independent of the acceptance type used by the deterministic ω-automaton, it is valid for deterministic Büchi, Rabin and Streett automata.

So, for an LTL formula ψ, we get a lower bound for the size of a deterministic Rabin, Streett or Büchi automaton for ψ of

$$2^{2^{\Omega\left(\sqrt{|\psi|}\right)}}.$$

4 Improving the translation

In this chapter, we will try to develop improvements for the translation from LTL and nondeterministic Büchi automata to deterministic ω-automata. First, we consider ways to improve Safra's construction by handling special cases more efficiently. Then we investigate ways to optimize deterministic Rabin and Streett automata. Finally, we consider additional approaches to the translation from LTL to deterministic ω-automata, using the relationship between $\vee$ in LTL and the union of DRA, the duality of Rabin and Streett automata, and the expressibility of (co-)safety properties by deterministic Büchi automata to potentially generate smaller automata.

In Chapter 5, the performance of these optimizations will be experimentally evaluated.

4.1 Optimizations during the conversion

This section presents optimizations that can be performed "on-the-fly" during Safra's construction. Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be the nondeterministic Büchi automaton used as input in Safra's construction.

4.1.1 NBA: True loops on accepting states

Definition 4.1.1 (Accepting states with true loops). We define the set $accTrueLoop$ to be the set of accepting states of NBA $\mathcal{A}$ that have a transition back to themselves for all $a \in \Sigma$:

$$accTrueLoop(\mathcal{A}) = \{q \in F \mid q \in \delta(q, a) \text{ for all } a \in \Sigma\}.$$

If a run of the NBA $\mathcal{A}$ reaches a state q_F from $accTrueLoop(\mathcal{A})$, it can clearly be continued to get an infinite, accepting run: It just has to stay in q_F.

From this follows that we can abort Safra's construction any time the label of the root node of a Safra tree T contains a state $q_F \in accTrueLoop(\mathcal{A})$, and make the tree T accepting and set $\delta'(T, a) = T$ for all $a \in \Sigma$, i.e. add a `true` self loop.

This can be implemented by inserting an additional step 3' after step 3 (Powerset) in Safra's construction as follows:

3'. **Accepting true loop detection**
If $label(root) \cap accTrueLoop(\mathcal{A}) \neq \emptyset$, pick a canonical state q_F from $accTrueLoop(\mathcal{A})$, remove all descendants of the root node, set $label(root) = q_F$ and mark the root node.

By changing the tree in the this way, the normal Safra's construction will automatically generate a self loop for every $a \in \Sigma$ and the tree will be accepting as the root node is marked infinitely often.

Choosing a canonical representative state q_F from $accTrueLoop(\mathcal{A})$ instead of, for example, setting $label(root) = label(root) \cap accTrueLoop(\mathcal{A})$ ensures that there will be only a single Safra tree generated by rule 3' instead of potentially many.

Calculating $accTrueLoop(\mathcal{A})$: The set $accTrueLoop(\mathcal{A})$ can be easily calculated by inspecting the transitions of the accepting states in the NBA $\mathcal{A}$. If the implementation provides no direct access to the accepting states, a simple depth-first search (DFS) can be used to calculate $accTrueLoop(\mathcal{A})$.

This optimization will be even more effective if the NBA was optimized by finding states that are equivalent to an accepting state with a true loop and replacing them with an accepting true loop. For example, nodes q with $\delta(q, a) \cap accTrueLoop(\mathcal{A}) \neq \emptyset$ for all $a \in \Sigma$ can be replaced by an accepting state with a `true` self loop. States in maximally strongly connected components (SCCs) containing only accepting states with a total transition function (i.e. for all $a \in \Sigma$, $\delta(q, a) \neq \emptyset$) and no edges leaving the SCC can also be replaced.

Example Consider the LTL formula $\Diamond(\mathsf{a} \vee \Box\,\mathsf{b})$, which corresponds to the NBA in Figure 4.1.

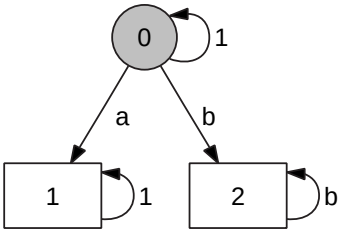


Figure 4.1: Nondeterministic Büchi Automaton for $\mathcal{L}(\Diamond(\mathsf{a} \vee \Box\,\mathsf{b}))$

Safra's construction without step 3' creates a DRA with 22 states, with 18 states containing the NBA state 1 in the labeling of the root node. This 18 DRA states can all be collapsed to 1 accepting state, resulting in a DRA with 5 states (Figure 4.2).

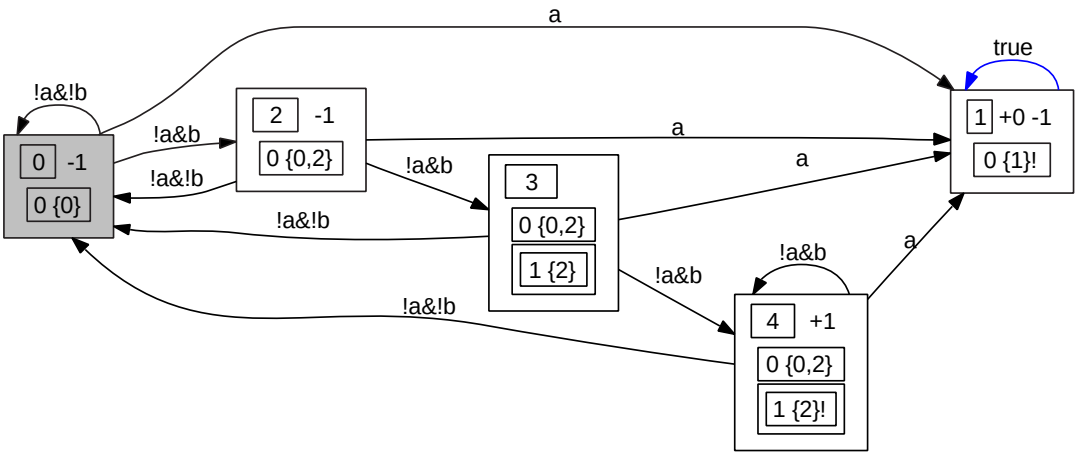


Figure 4.2: Deterministic Rabin Automaton for $\mathcal{L}(\Diamond(\mathsf{a} \vee \Box\,\mathsf{b}))$, created with Safra's construction and detection of accepting true loops in the NBA

4.1.2 NBA: All successors are accepting

A special case of Safra's construction arises when all the states in a label have only successors that are accepting in the NBA. Then a single powerset construction is sufficient as we only have to track if a run is finite; the infinite runs are all accepting as no non-accepting state in the NBA can be reached.

Safra's construction handles this special case well by default: If $\mathit{label}(n) \cap F = \mathit{label}(n)$, node n will be marked and has no children (a child with $\mathit{label}(n)$ is branched in step 2 and deleted in step 6, marking n). If all successors of $\mathit{label}(n)$ are also in F, node n will stay marked and have no children in subsequent trees or it will be deleted when the runs it tracks are finite.

A possibility for optimization remains, as it takes an additional step in the beginning for Safra's construction to fall into the pattern described above.

Consider the automaton in figure 4.2. As NBA state 2 is accepting and has only itself as successor, in DRA state 4 we see that node 1 stays marked or is deleted when the transition to DRA state 1 is taken. But DRA state 3 has node 1 with label 2 but is unmarked. This is unnecessary and marking node 1 would merge state 3 with state 4, resulting in a reduction of 1 state.

We now formalize the set of states that have only accepting states as successors:

$$\mathit{succAcc}(\mathcal{A}) = \{q_F \in F \mid \mathit{succ}^*(q_F) \subseteq F\}$$

If after the construction of a new tree with Safra's algorithm, the label of a node i of the Safra tree has only states that are members of $\mathit{succAcc}(\mathcal{A})$ and is not marked, it can be marked (and the tree will thus be placed into L_i of the acceptance condition). This

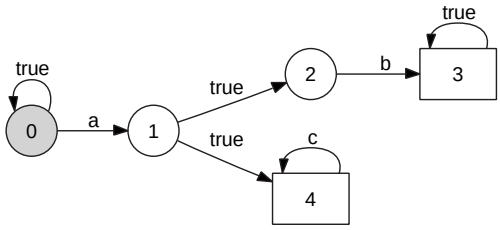


Figure 4.3: NBA for $\mathcal{L}(\Diamond(\mathtt{a} \wedge \mathtt{X}\,\mathtt{X}(\mathtt{b} \vee \Box\,\mathtt{c})))$
States 1 and 2 can be made accepting without changing the language.

can be done as an additional step after the normal Safra's construction:

6'. **Mark nodes with** $succAcc(\mathcal{A})$
If for a node n is not marked and $label(n) \cap succAcc(\mathcal{A}) = label(n)$, remove all children of n and mark n.

To show that this does not change the language, we have to consider the two possibilities for the immediate successors of a tree that was modified this way: If, in the immediate successor, node i is deleted, the tree is placed into the corresponding U_i and effectively cancels the modification that was made. If on the other hand the immediate successor still contains node i, it will be marked as $\mathit{label}(n_i) \cap F = \mathit{label}(n_i)$. Without the modification, this tree would have been placed into L_i. As these are the only two possible types of immediate successors, the marking of node i does not change the language.

Some states in the NBA can be added to $succAcc(\mathcal{A})$ even if they are not accepting states: A state q in $\mathcal{A}$ that is not in F, but with $\delta(q, a) \subseteq succAcc(\mathcal{A})$ for all $a \in \Sigma$, i.e. all outgoing edges go to states in $succAcc$ and there is no self loop back to q, can be "made accepting" by adding it to F, without changing the language of the automaton. This also adds q to $succAcc(\mathcal{A})$ and can be repeated until there are no more states to add to F.

For example, in figure 4.3, initially only the states 3 and 4 are in $succAcc$, but state 2 and then state 1 can be made accepting, too. Only state 1 has to stay non-accepting as it has a loop to itself.

This step does not need to be carried out separately but can be implicitly done during the calculation of $succAcc$:

Calculating the set $succAcc$

We can easily calculate the set $succAcc(\mathcal{A})$ for a nondeterministic Büchi automaton $\mathcal{A}$ with standard graph analysis algorithms:

1. Calculate the maximally strongly connected components (SCCs) of the automaton $\mathcal{A}$.
2. Calculate the directed acyclical graph (DAG) of the SCCs representing reachability.
3. Calculate a topological ordering on the DAG of the SCCs.
4. In backward topological ordering, visit the SCCs and check:
   a) If all states in the current SCC are accepting and all SCCs that are successors of the current SCC are marked, then mark the current SCC.
   b) If the current SCC contains only a single state that has no edge leading back to itself, is not accepting and all successor SCCs are marked, then mark the current SCC.
5. $succAcc(\mathcal{A}) = \{$All the states in marked SCCs.$\}$.

It is clear that either all or none of the states in an SCC are in $succAcc(\mathcal{A})$. Visiting the SCCs in backward topological order guarantees that it was already considered if an SCC scc is marked to be in $succAcc(\mathcal{A})$ or not before an SCC scc' is visited that has scc as successor. Step 4b) adds non-accepting states to $succAcc$ if they can be made accepting without changing the language of $\mathcal{A}$.

Calculating the set $succAcc(\mathcal{A})$ can be done in linear time, $\mathcal{O}(|Q| + |E|)$, where $|E|$ is the number of transitions (i.e. edges) in $\mathcal{A}$.

4.1.3 Reordering the Safra tree

Safra's construction assumes a strict ordering of the sibling nodes in Safra trees, used in step 4 (Normalize siblings) to reestablish the requirement on Safra trees that siblings have disjunct labels:

4. **Normalize siblings**
   For every two sibling nodes such that they share a $q \in Q$ in their labels, remove q from the label of the youngest node and all its children.

This is used in the proof of correctness to show that while a run may move to an older sibling, it will eventually stay in a single node, because the width of the tree is bounded and it therefore can only move finitely many times to an older sibling.

This strict ordering is not necessary in all cases and can sometimes be relaxed to transform Safra trees that differ only in the ordering of their nodes to a canonical representation. This increases the chance that such a Safra tree already exists and therefore does not need to be added as a new state to the DRA.

To deal with the labels of Safra tree nodes, we extend the definition of $succ^*$ to handle sets of states, $succ^* : 2^Q \to Q$:

$$succ^*(S) = \bigcup_{q \in S} succ^*(q)$$

Definition 4.1.2 (Independent nodes). We call two siblings n and n' in a Safra tree *independent*, if the nodes in their labels have no common successors:

$$succ^*(label(n)) \cap succ^*(label(n')) = \emptyset.$$

It is clear that step 4 of Safra's construction will never be applied to two independent siblings, as they will never share a common state in their labels. Therefore, the relative order of independent siblings is irrelevant for the proof of correctness and changing it will not change the language recognized by the DRA.

This can be used to define a canonical order on the independent siblings, which reduces the number of possible different Safra trees:

Definition 4.1.3 (Canonical order for independent siblings). Let $<$ be the total order on the siblings as defined in Safra's construction ("older-than") and $\prec_{\text{ind.}}$ an arbitrary, but fixed total order on Safra tree nodes, then we can define a new order $<'$ on sibling nodes n and n' as follows:

$$n <' n' \quad \Leftrightarrow \quad \begin{cases} n \prec_{\text{ind.}} n' & \text{if } n \text{ and } n' \text{ are independant,} \\ n < n' & \text{otherwise.} \end{cases}$$

The order $\prec_{\text{ind.}}$ for the independent nodes can be any total order on Safra tree nodes using the name, label and "marked" flag of the nodes as its input variables.

To bring the Safra's trees into the canonical form, all siblings in the tree are sorted using a standard sorting algorithm using the order $<'$ at the end of the normal construction. The set of reachable successors $succ^*$ can be calculated using standard graph algorithms similar to the process shown in Section 4.1.2 and independence of nodes can then be determined easily.

4.1.4 Naming the nodes in Safra trees

New nodes in Safra trees are only created in step 2 (Branch accepting) of Safra's construction:

2. **Branch accepting**
   For every node n with $label(n) \cap F \neq \emptyset$, create a new node as the youngest child of n labeled with $label(n) \cap F$. The new node is named with an unused name from $\{1, \ldots, 2n\}$.

As we can choose any unused name, we have significant freedom in choosing the name for the new node. As the set of Safra trees that are created during Safra's construction becomes the set of states in the deterministic Rabin automaton, we are interested in having the smallest number of different Safra trees. One way to keep the number of different Safra trees low is to try to name new nodes in a way that the resulting tree matches an already existing tree, thus adding no additional state to the DRA. It is clear that this does not change the language of the resulting automaton in any way.

One way to do this is to mark the new nodes and then search for a matching tree among the already existing trees. If no matching tree is found, the new nodes are named as normal and a new state in the DRA is created for the tree.

This can be implemented by calculating the Safra trees the normal way, naming new nodes temporary with a special symbol, e.g. '*'. We simultaneously have to keep track of the names of nodes deleted during steps 4, 5 and 6 of Safra's construction, as they are still in use in step 2 where the new nodes are named and can therefore not be reused. It is clear that nodes that are created and then directly deleted again do not have to be tracked, as we can pretend to have named them with a convenient name that is unused.

Let T_* be a Safra tree after the steps of Safra's construction, with new nodes marked with '*' and $deleted \subseteq \{1, \ldots, 2n\}$ the set of names of the deleted nodes.

Possible candidates for a match must have the same structure as T_*, which can be formalized as follows:

Definition 4.1.4 (Structural equality). We define *structural equality* as a binary relation on Safra trees $\equiv_{\text{struct}}$ with

$$T_1 \equiv_{\text{struct}} T_2 \quad \Leftrightarrow \quad \text{Ignoring the names of the nodes, } T_1 \text{ and } T_2 \text{ are equal.}$$

This means that there is a bijection f from the node set of T_1 to the node set of T_2, such that for every node n of T_1 the following holds:

1. The labels agree: $label(n) = label(f(n))$,
2. the marks agree: $marked(n) = marked(f(n))$,
3. the parents agree: $f(parent(n)) = parent(f(n))$,
4. for all children of n, the children agree: $f(child_i(n)) = child_i(f(n))$.

Note that $name(n) = name(f(n))$ is not required, the naming is ignored.

Definition 4.1.5 (Match). An already constructed Safra tree T and a newly constructed tree T_* *match* if the following three conditions are met:

1. $T \equiv_{\text{struct}} T_*$
2. For all nodes n named '*' in T_*, the corresponding node $f(n)$ in T is not named with a name from $deleted$
3. For all nodes n not named '*' in T_*, the corresponding node $f(n)$ in T has the same name as the node in T_*.

One way to keep track of the trees that are possible candidates for matching is to partition the already existing trees by structural equality. This can be implemented for example by a hash map having the structure of the trees as the key and a set (an array, a list,...) as the value, containing all trees that are structural equal. This allows efficient access to all trees that are structural equal to T_*, which can then be checked one after another to see if one of them matches with T_*.
If, from the existing trees, many are structurally equal, then there are many candidates for matching and this linear search for a match among the candidates will be inefficient.

But as the idea of this optimization is to name new trees in a way that keeps the number of structural equal trees low, this should hopefully not be a big problem in practice.

4.2 Optimizations on the complete automaton

The following optimizations operate on complete deterministic Rabin automata, trying to simplify the acceptance condition and to reduce the number of states. Due to the duality of Streett and Rabin automata they work unchanged on Streett automata.

4.2.1 Optimizing the acceptance condition

A Rabin acceptance pair (L_i, U_i) may be removed from the acceptance condition Ω of a deterministic Rabin automaton if it can never be satisfied, for example if one of the following conditions is met:

- $L_i = \emptyset$
- $U_i = Q$
- $L_i \subseteq U_i$

Sometimes a Rabin acceptance pair (L_i, U_i) is redundant and can be removed:

- There exists a Rabin acceptance pair $(L_j, U_j) \in \Omega$, such that $L_i \subseteq L_j$ and $U_i \subseteq U_j$

Additionally, we can remove states from L_i if they are also members of U_i.

Note As these operations are very easy to perform and simplify the acceptance conditions a great deal, they are used for all automata discussed in this thesis, without noting explicitly that this optimization has been performed.

4.2.2 Bisimulation

One of the standard algorithms for minimization of deterministic automata on finite words is to calculate a simulation relation on the automaton and then build the quotient automaton. There are several different notions of simulation, for deterministic ω-automata we consider *direct bisimulation*:

Definition 4.2.1 (Equivalence). Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, \Omega)$ be a deterministic Rabin automaton. Then we define an equivalence relation $\equiv \subseteq Q \times Q$ as

$$q \equiv p \quad \Leftrightarrow \quad \text{for all } z \in (\Sigma^* \cup \Sigma^\omega) : acc(\delta(q, z)) = acc(\delta(p, z)).$$

Let $[q]_\equiv = \{p \in Q \mid p \equiv q\}$ be the *equivalence class* of a state q and
let $S/_\equiv = \{[q]_\equiv \mid q \in S\}$ for a set $S \subseteq Q$ be the *set of equivalence classes* for S.

Definition 4.2.2 (Quotient automaton). Let $\equiv$ be an equivalence relation on a DRA $\mathcal{A}$ with acceptance condition $\Omega = \{(L_1, U_1), \ldots, (L_r, U_r)\}$ as defined above. Then we define the *quotient automaton* $\mathcal{A}/_{\equiv} = (Q', \Sigma, \delta', q_0', \Omega')$, also a DRA, with

$$\begin{aligned} Q' &= Q/_{\equiv} \\ q_0' &= [q_0]_{\equiv} \\ \delta'([q]_{\equiv}, a) &= [\delta(q,a)]_{\equiv} \\ \Omega' &= \{(L_1/_{\equiv}, U_1/_{\equiv}), \ldots, (L_r/_{\equiv}, U_r/_{\equiv})\} \end{aligned}$$

We have to show that δ' is well defined, i.e. $p \equiv q \Rightarrow \delta(p,a) \equiv \delta(q,a)$ for $p, q \in Q$ and $a \in \Sigma$:

$$\begin{aligned} p \equiv q \quad &\Rightarrow \quad acc(\delta(p,z)) = acc(\delta(q,z)), \text{for all } z \in (\Sigma^* \cup \Sigma^\omega) \\ &\Rightarrow \quad acc(\delta(p,z)) = acc(\delta(q,z)), \text{for all } z = a \cdot y, \text{ with } a \in \Sigma \text{ and } y \in (\Sigma^* \cup \Sigma^\omega) \\ &\Rightarrow \quad acc(\delta(p, a \cdot y)) = acc(\delta(q, a \cdot y)) \text{ for all } a \in \Sigma \text{ and } y \in (\Sigma^* \cup \Sigma^\omega) \\ &\Rightarrow \quad acc(\delta(\delta(p,a), y)) = acc(\delta(\delta(q,a), y)) \text{ for all } a \in \Sigma \text{ and } y \in (\Sigma^* \cup \Sigma^\omega) \\ &\Rightarrow \quad \delta(p,a) \equiv \delta(q,a) \text{ for all } a \in \Sigma. \end{aligned}$$

Proposition 4.2.1. The language of the quotient automaton $\mathcal{A}/_{\equiv}$ and the language of the original automaton $\mathcal{A}$ are the same, $\mathcal{L}(\mathcal{A}/_{\equiv}) = \mathcal{L}(\mathcal{A})$.

Proof. By using the empty word $z = \varepsilon \in \Sigma^*$ in Definition 4.2.1, it follows directly that if $q \equiv p$, then the acceptance signatures $acc(q) = acc(p)$ are the same.

The acceptance condition of the quotient automaton consists of pairs $(L_i/_{\equiv}, U_i/_{\equiv})$, therefore a state $[q]_{\equiv}$ of the quotient automaton is in $L_i/_{\equiv}$ iff $q \in L_i$ and $[q]_{\equiv} \in U_i/_{\equiv}$ iff $q \in U_i$. It is easy to see that $acc(q) = acc([q]_{\equiv})$.

Let $\sigma \in \Sigma^\omega$ be an infinite word and let $\pi = \pi_1, \pi_2, \ldots$ be the run of automaton $\mathcal{A}$ for σ. We defined the quotient automaton so that the run of $\mathcal{A}/_{\equiv}$ for σ is $\pi' = [\pi_1]_{\equiv}, [\pi_2]_{\equiv}, \ldots$. Since $acc(\pi_i) = acc([\pi_i]_{\equiv})$, the two runs π and π' visit exactly the same acceptance pairs at the same time and are therefore either both accepting or both rejecting, $\sigma \in \mathcal{L}(\mathcal{A}) \Leftrightarrow \sigma \in \mathcal{L}(\mathcal{A}/_{\equiv})$. □

To calculate $\equiv$, we start with a relation P_0 of Q by grouping all states that have the same acceptance signature:

$$P_0 = \{(q,p) \in Q \times Q \mid acc(q) = acc(p)\}$$

It is clear that $P_0 \supseteq \equiv$, i.e. there may be pairs $(q,p) \in P_0$ that are not equivalent ($q \not\equiv p$). We iteratively refine the relations P_i to get a relation P_{i+1} until there are no more pairs with $q \not\equiv p$:

$$P_0 \supseteq P_1 \supseteq P_2 \supseteq \cdots \supseteq P_n = \equiv$$

To do this, we need to identify the states that are not equivalent. The following lemma provides a criterion we can work with:

Lemma 4.2.2. $p \not\equiv q \Leftrightarrow \exists\, a \in (\Sigma \cup \{\varepsilon\}) \,:\, \delta(p,a) \not\equiv \delta(q,a)$

Proof. "⇐": Follows directly from the definition of $\equiv$ by using $z = a$.
"⇒": Let $x \in \Sigma^*$ be the shortest string with $acc(\delta(p,x)) \neq acc(\delta(q,x))$. If $x = \varepsilon$, we are done. Otherwise, let a be the first letter of x: $x = a{\cdot}y$, with $a \in \Sigma$ and $y \in \Sigma^*$. Let $p' = \delta(p,a)$ and $q' = \delta(q,a)$. Because $\delta(p, a{\cdot}y) = \delta(p', y)$ and $\delta(q, a{\cdot}y) = \delta(q', y)$, it is clear that $acc(\delta(p', y)) \neq acc(\delta(q', y))$ and therefore $p' \not\equiv q'$. □

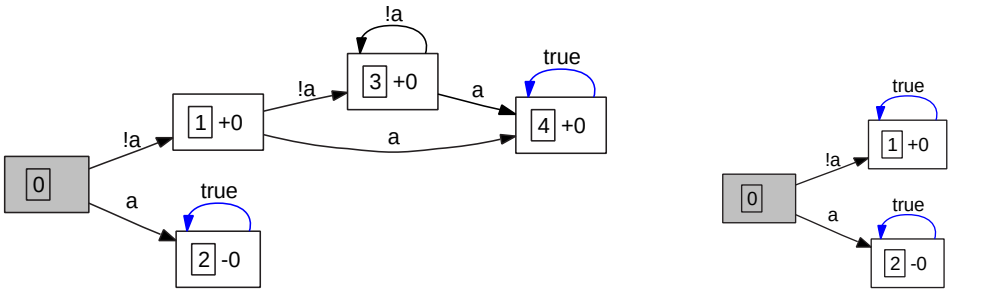
With this lemma, we can refine our relations until $P_{i+1} = P_i$, i.e. there are no more states $p \equiv_i q$ with an $a \in \Sigma$ so that $\delta(p,a) \not\equiv_i \delta(q,a)$, with $p \equiv_i q \Leftrightarrow (p,q) \in P_i$:

$$P_{i+1} = \{(p,q) \in Q \times Q \mid (p,q) \in P_i \text{ and } (\delta(p,a), \delta(q,a)) \in P_i \text{ for all } a \in \Sigma\}$$

If the automaton $\mathcal{A}$ has n states, it is guaranteed that $P_n = P_{n+1}$, i.e. we need at most n iterations: Every time $P_i \supseteq P_{i+1}$, there is at least one new equivalence class in P_{i+1} and therefore after n iterations where $P_i \neq P_{i+1}$, every equivalence class $[q]_{\equiv_n}$ has one single member, $[q]_{\equiv_n}$=$\{q\}$ and cannot be refined further.

Example

Figure 4.4 shows an example for bisimulation for the (randomly generated) LTL formula $\varphi = (\Diamond((\mathtt{X}\,\mathtt{X}\,\mathsf{a})\,\mathtt{U}\,\mathsf{a}))\,\mathtt{V}\,\neg\,\mathsf{a}$.



a) DRA before bisimulation b) DRA after bisimulation

Figure 4.4: Example of bisimulation for formula $\varphi = (\Diamond((\mathtt{X}\,\mathtt{X}\,\mathsf{a})\,\mathtt{U}\,\mathsf{a}))\,\mathtt{V}\,\neg\,\mathsf{a}$.
States 1, 3 and 4 in automaton a) bisimulate each other and can be merged to form state 1 in automaton b).

Analysis

For deterministic automata on finite words or for Büchi automata, the initial partition has only two equivalence classes, the states in F and the states not in F. For a DRA $\mathcal{A}$

the initial partition will be much finer, it has as many equivalence classes as there are different acceptance signatures for the states of $\mathcal{A}$. If $\mathcal{A}$ has k accepting pairs, there may be as much as 3^k different initial equivalence classes. As the number of equivalence classes in the initial partition is a lower bound on the number of states in the quotient automaton, it is clear that for DRA the possible minimization will be generally less successful than for the automata with "easier" acceptance condition.

Another problem is the need for strict equality of the acceptance signatures in the definition of equivalence, i.e. the requirement of *direct* bisimulation. Consider the DRA in Figure 3.5 on page 26, showing the DRA generated for $\varphi = \Diamond\Box \mathsf{a}$. The states 1, 2 and 3 are all equivalent and could be merged (replacing 1 and 2 by state 3), but our bisimulation reduction algorithm can do nothing as the acceptance signatures for the states are different.

For Büchi automata, several other, more powerful notions like *fair* or *delayed* simulation have been proposed (e.g. [EWS01]), but it has to be ensured that the simulation quotient preserves the language of the automaton to be useful in reducing the automaton size. If similar approaches can work for deterministic Rabin or Streett automata remains to be seen.

4.3 Union of DRA

If the LTL formula φ that needs to be translated has the form $\varphi = (\varphi_1) \vee (\varphi_2)$, i.e. the operator on the top of the syntax tree is the logical disjunction, we can construct two deterministic Rabin automata DRA_{φ_1} and DRA_{φ_2} for the two subformulas φ_1 and φ_2 and construct a DRA for φ by creating the union $DRA_{\varphi \vee \varphi_2} = DRA_{\varphi_1} \cup DRA_{\varphi_2}$ of the two automata.

The generated union DRA might be smaller than a DRA generated for the whole formula as the subformulas are shorter and probably simpler, which can lead to smaller NBA and DRA for the subformulas.

Definition 4.3.1 (Union of two DRA).
Let $DRA_1 = (Q_1, \Sigma, \delta_1, q_0^1, \Omega_1)$ and $DRA_2 = (Q_2, \Sigma, \delta_2, q_0^2, \Omega_2)$ be two deterministic Rabin automata over the same alphabet Σ, with $\Omega_1 = \{(L_1^1, U_1^1), \ldots, (L_{r_1}^1, U_{r_1}^1)\}$ and $\Omega_2 = \{(L_1^2, U_1^2), \ldots, (L_{r_2}^2, U_{r_2}^2)\}$ as acceptance conditions.

Then we define the union $DRA_{DRA_1 \cup DRA_2} = (Q', \Sigma, \delta', q_0', \Omega')$ as follows:

- $Q' = Q_1 \times Q_2$
- $q_0' = (q_0^1, q_0^2)$
- For $a \in \Sigma$, $q' = (q_1, q_2) \in Q'$ define
  $\delta'(q', a) = (\delta_1(q_1, a), \delta_2(q_2, a))$
- $\Omega' = \{(L_1', U_1'), \ldots, (L_{r_1}', U_{r_1}')\} \cup \{(L_1'', U_1''), \ldots, (L_{r_2}'', U_{r_2}'')\}$, with
  - $L_i' = L_i^1 \times Q_2$,
    $U_i' = U_i^1 \times Q_2$.
  - $L_i'' = Q_1 \times L_i^2$,
    $U_i'' = Q_1 \times U_j^2$.

  be the acceptance condition.

The basic idea is to generate the product automaton of the two DRA, as this allows them to "run in parallel". The states of the union DRA are 2-tuples, the first component being the state from DRA_1, the second component the state from DRA_2. The transition function δ' applies δ_1 from the first DRA to the first component, and δ_2 from the second DRA to the second component. The start states from the two DRA become the components in the start tuple for the union DRA.

The interesting part is the acceptance condition. The acceptance pairs from the first DRA are extended with all states from the second DRA and therefore "care" only for the first component. So, an acceptance pair that came from the first automaton will be satisfied iff the run of the first component of the product automaton would be accepting in DRA_1. The same is done with the acceptance pairs of the second automaton. With the semantics of the Rabin acceptance condition that a run is accepting iff one of the acceptance pairs is satisfied, the union DRA will be accepting iff there exists an accepting run in DRA_1 or in DRA_2:

$$\mathcal{L}(DRA_{DRA_1 \cup DRA_2}) = \mathcal{L}(DRA_1) \cup \mathcal{L}(DRA_2).$$

It is clearly sufficient to only generate the states reachable by (q_{0_1}, q_{0_2}) instead of creating the full Cartesian product $Q_1 \times Q_2$ for the states of the union DRA.

Size of the union DRA. If DRA_1 has $|Q_1| = n_1$ states, DRA_2 has $|Q_2| = n_2$ states, then the union $DRA_{DRA_1 \cup DRA_2}$ has at most $n_1 * n_2$ states. The number of acceptance pairs of the union DRA is $r' = r_1 + r_2$, the sum of the number of acceptance pairs of DRA_1 and DRA_2. As a lower bound, the union DRA has at least $max(n_1, n_2)$ states.

Accepting true loop. We can adapt the technique from Section 4.1.1 to abort the construction of the product automaton once a state q with a `true` self loop is reached in either the first or the second DRA, that is *accepting*, i.e. is member of at least one L_i and not U_i. All runs in a DRA visiting this state are clearly accepting, therefore the corresponding run in the union DRA will also be accepting and we can short cut the product automaton generation by just generating an accepting state with a `true` self loop.

Intersection for Streett automata. Due to the duality of the Rabin and Streett acceptance conditions, if we apply the union construction for DRA on two DSA, we get the intersection of the two DSA.

Intersection for Rabin automata (Union for Streett). A similar approach for the logical conjunction $\wedge$ by generating the intersection of two DRA is more complicated, it would have $r_1 \cdot r_2$ acceptance pairs in the intersection automaton, one for every 2-tuple consisting of one acceptance pair from the first and one from the second automaton. Additionally, in the states of the intersection automaton, the status of each acceptance condition in the two automata would have to be tracked, as it is e.g. not guaranteed that both automata become accepting at the same time in the product automaton.

Example

Figure 4.5 shows the union construction for the LTL formula $(\Diamond\Box\,\mathsf{a}) \vee (\Box\,\mathsf{b})$. Without the union construction, the resulting DRA has 11 states and 2 acceptance pairs, the union construction reduces the number of states to 7.

On the other hand, it is clear that for other formulas the LTL→NBA translation step used on the complete formula instead of on the two subformulas can detect and remove redundancies, which can lead to a smaller NBA and DRA than using the union construction.

For example, consider the formula $\varphi = \underbrace{(\Diamond\Box\,\mathsf{a})}_{\varphi_1} \vee \underbrace{((\Diamond\,\mathsf{a}) \wedge \Box((\mathsf{a} \rightarrow \mathtt{X}\,\mathsf{b}) \wedge (\mathsf{b} \rightarrow \mathsf{a} \wedge \mathtt{X}\,\mathsf{a})))}_{\varphi_2}$.
As $\mathcal{L}(\varphi_2) \subset \mathcal{L}(\varphi_1)$, φ_2 is redundant and a smart LTL→NBA translator could detect this redundancy and return a (small) NBA just for φ_1, while the union construction has two

generate the product automaton for the two DRA for φ_1 and φ_2.

4.4 Generating Rabin or Streett automata

Some applications need a translation from LTL formulas to deterministic ω-automata, but do not particularly care if the automaton is a Rabin or a Streett automaton, only that the automaton has as few states as possible. As we have seen in Section 2.6.1, for some languages Streett automata can be exponentially more compact than Rabin automata, so this flexibility can have huge benefits. But even in the case where the difference is smaller, if we create both a Streett automaton and a Rabin automaton for a formula ψ and return the smaller automaton, only in the case where both automata have exactly the same size we will get no smaller automaton.

To actually construct Streett automata, we exploit the duality of Rabin and Streett automata (Section 2.6.1) and use a trick often used in traditional LTL model checking: There, if we want to verify that a property expressed as an LTL formula ψ is satisfied by a transition system $\mathcal{T}$, we need a nondeterministic Büchi automaton with language $\overline{\mathcal{L}(\psi)}$ and check that the intersection $\mathcal{T} \cap \overline{\mathcal{L}(\psi)} = \emptyset$, i.e. is empty. Instead of generating a nondeterministic Büchi automaton for $\mathcal{L}(\psi)$ and then complementing it to get $\overline{\mathcal{L}(\psi)}$, which is computationally difficult, it is much easier to perform the complementation on the syntactic level of LTL and generate the NBA for $\neg\psi$, as $\mathcal{L}(\neg\psi) = \overline{\mathcal{L}(\psi)}$.

So, to construct a deterministic Streett automaton for a formula ψ using Safra's construction, we first create a deterministic Rabin automaton for $\neg\psi$ and then regard it as a Streett automaton, thereby complementing the accepted language:

$$\neg\psi \xrightarrow{\text{Safra's construction}} \text{DRA}(\mathcal{L}(\neg\psi)) \xrightarrow[\text{=complementation}]{\text{regard as Streett}} \text{DSA}(\mathcal{L}(\psi)).$$

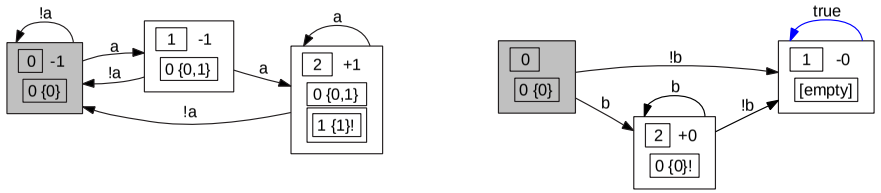
Example

Here are the sizes of DRA and DSA constructed for two strong fairness formulas:

Formula	DRA	DSA	DRA (opt.)	DSA (opt.)
$(\Box\Diamond \mathsf{a}) \rightarrow (\Box\Diamond \mathsf{b})$	61	7	12	7
$((\Box\Diamond \mathsf{a}) \rightarrow (\Box\Diamond \mathsf{b})) \wedge ((\Box\Diamond \mathsf{c}) \rightarrow (\Box\Diamond \mathsf{d}))$	67051	298	18526	49

The first two values are the number of states using the standard Safra's construction, the last two values are the number of states when the optimization techniques developed in this chapter were used. For example, the union construction is responsible for reducing the size of the DRA for the first formula (the implication $\varphi_1 \rightarrow \varphi_2$ can be regarded as $(\neg\varphi_1) \vee \varphi_2$).

a) DRA_{φ_1} for $\varphi_1 = \Diamond\Box\,\mathsf{a}$ b) DRA_{φ_2} for $\varphi_2 = \Box\,\mathsf{b}$



c) Union DRA for $\varphi = (\Diamond\Box\,\mathsf{a}) \vee (\Box\,\mathsf{b}), \quad \varphi = \varphi_1 \vee \varphi_2$

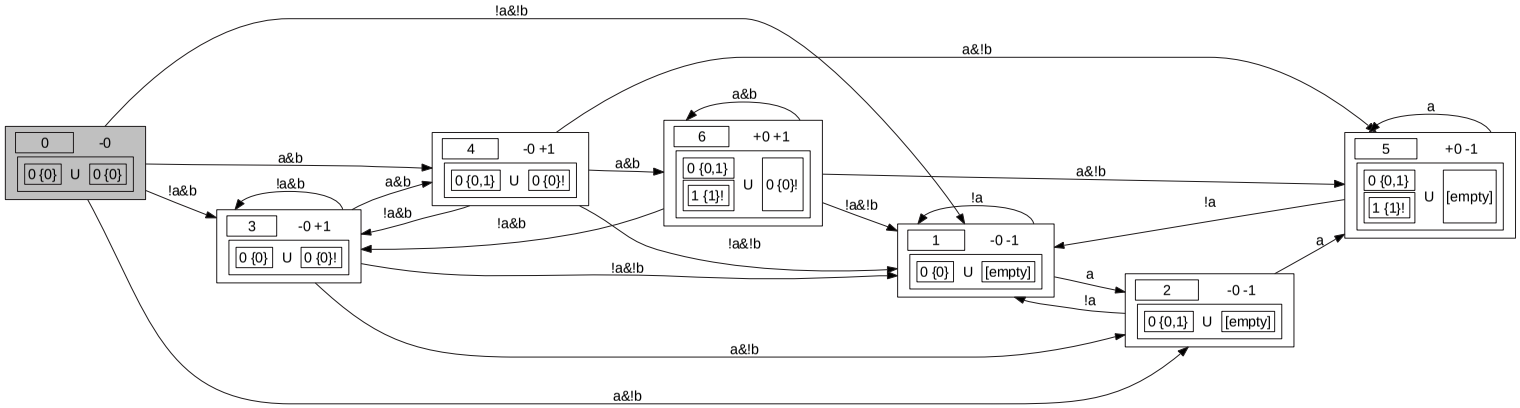


Figure 4.5: Union construction for LTL formula $\varphi = (\Diamond\Box\,\mathsf{a}) \vee (\Box\,\mathsf{b})$.
First, the automata DRA_{φ_1} and DRA_{φ_2} are constructed for the two subformulas φ_1 and φ_2. Then, the product automaton in c) is constructed with DRA_{φ_1} as the first (left) component and DRA_{φ_2} as the second (right) component. In the product automaton, acceptance condition 0 corresponds to the acceptance condition of DRA_{φ_1}, acceptance condition 1 to the acceptance condition of DRA_{φ_2}.

4.5 (Co-)Safety formulas and deterministic automata

As seen in Section 2.7.1, for safety and co-safety languages deterministic Büchi automata are expressive enough. For some LTL formulas representing safety and co-safety formulas, the corresponding deterministic Büchi automaton can be generated directly, i.e. without using Safra's construction.

Let φ_{safe} be a safety LTL formula. As $badpref(\mathcal{L}(\varphi_{safe}))$ is a subset of Σ^*, and therefore consists of finite words, it can be recognized by an automaton on finite words, either nondeterministic (NFA) or deterministic (DFA). In the finite case, the determinization of NFA to DFA is performed by a simple powerset construction, with a worst-case exponential blowup. This DFA can then easily be extended to a deterministic Büchi automaton $\mathcal{A}^{DBA}_{pref}(\varphi_{safe})$ which recognizes $\overline{\mathcal{L}(\varphi_{safe})} = \mathcal{L}(\neg\varphi_{safe})$.

[KV99] propose an algorithm that can construct NFA with $2^{\mathcal{O}(|\psi|)}$ states for the bad prefixes of a subset of safety formulas, resulting in DFA with $2^{2^{\mathcal{O}(|\psi|)}}$ states. This algorithm was improved by Timo Latvala [Lat02] to generate smaller-than-worst-case automata more often. He also provides an implementation, `scheck`[1].

As we have seen in Section 2.8.1, we can easily create an equivalent DRA or DSA for a deterministic Büchi automaton. This provides the efficient complementation we need to convert the automata recognizing the bad prefixes of $\mathcal{L}(\psi)$ to automata recognizing $\mathcal{L}(\psi)$. We get the following procedures for calculating a DRA for a safety LTL formula φ_{safe}:

$$\begin{array}{ccccccc}
\varphi_{\text{safe}} & \longrightarrow & \mathcal{A}^{DBA}_{pref}(\varphi_{\text{safe}}) & \xrightarrow[\Omega=\{(Q,F)\}]{\text{Büchi to Streett}} & \text{DSA} & \xrightarrow[\text{=complementation}]{\text{regard as Rabin}} & \text{DRA} \\
\| & & \| & & \| & & \| \\
\mathcal{L}(\varphi_{\text{safe}}) & & \overline{\mathcal{L}(\varphi_{\text{safe}})} & & \overline{\mathcal{L}(\varphi_{\text{safe}})} & & \mathcal{L}(\varphi_{\text{safe}})
\end{array}$$

And for a co-safety LTL formula $\varphi_{\text{co-safe}}$, because the negation $\neg\varphi_{\text{co-safe}}$ is a safety formula:

$$\begin{array}{ccccc}
\neg\varphi_{\text{co-safe}} & \longrightarrow & \mathcal{A}^{DBA}_{pref}(\neg\varphi_{\text{co-safe}}) & \xrightarrow[\Omega=\{(F,\emptyset)\}]{\text{Büchi to Rabin}} & \text{DRA}(\mathcal{L}(\varphi_{\text{co-safe}})) \\
\| & & \| & & \| \\
\overline{\mathcal{L}(\varphi_{\text{co-safe}})} & & \overline{\mathcal{L}(\neg\varphi_{\text{co-safe}})} & & \mathcal{L}(\varphi_{\text{co-safe}}) \\
 & & = \mathcal{L}(\varphi_{\text{co-safe}}) & &
\end{array}$$

[1] `http://www.tcs.hut.fi/~timo/scheck/`

Lower bound To establish a lower bound for the translation from safety formulas to DBA, the language used for establishing a lower bound for the conversion from LTL to deterministic ω-automata in Section 3.4.2 can be reused, as it is a safety language and therefore the same lower bound of $2^{2^{\Omega(\sqrt{|\psi|})}}$ applies.

Checking if an LTL formula is (co)-safe

The first challenge is to decide if an LTL formula ψ is actually a safety or co-safety formula. This is unfortunately a PSPACE-complete problem [Sis94][2]. As a partial solution, we can consider formulas that only use a certain fragment of the LTL syntax that can only express safety languages: An LTL formula φ is called *syntactically safe* iff in positive normal form (PNF) the only temporal operators are `V` (Release) and `X` (Nextstep). It follows that an LTL formula is *syntactically co-safe* iff in PNF the only temporal operators are `U` (Until) and `X` (Nextstep). Formulas that have only `X` (Nextstep) as temporal operator are syntactically safe and co-safe at the same time.

Informativeness

Generally, LTL safety formulas are difficult to handle, so the algorithms in [KV99] and [Lat02] generate finite automata only for a "well-behaved" subset of LTL safety formulas. For these formulas, it is easy to see why a given computation violates the formula. This concept is formalized by the concept of *informative prefixes*:

Definition 4.5.1 (Informativeness). [Lat02]
Let ψ be an LTL formula in PNF and $\sigma = \sigma_0, \sigma_1, \ldots, \sigma_n$ a finite word, then σ is called *informative* for ψ iff there exists a mapping $L : \{0, \ldots, n+1\} \to 2^{cl(\neg\psi)}$ such that the following conditions hold:

- $\neg\psi \in L(0)$
- $L(n+1)$ is empty
- for all $0 \leq i \leq n$ and $\varphi \in L(i)$ the following hold
  - If φ is a propositional assertion, it is satisfied by σ_i.
  - If $\varphi = \varphi_1 \vee \varphi_2$ then $\varphi_1 \in L(i)$ or $\varphi_2 \in L(i)$.
  - If $\varphi = \varphi_1 \wedge \varphi_2$ then $\varphi_1 \in L(i)$ and $\varphi_2 \in L(i)$.
  - If $\varphi = \mathtt{X}\,\varphi_1$, then $\varphi_1 \in L(i+1)$.
  - If $\varphi = \varphi_1\,\mathtt{U}\,\varphi_2$, then $\varphi_2 \in L(i)$ or $[\varphi_1 \in L(i)$ and $\varphi_1\,\mathtt{U}\,\varphi_2 \in L(i+1)]$.
  - If $\varphi = \varphi_1\,\mathtt{V}\,\varphi_2$, then $\varphi_2 \in L(i)$ and $[\varphi_1 \in L(i)$ or $\varphi_1\,\mathtt{V}\,\varphi_2 \in L(i+1)]$.

For example [KV99], the finite computation $\sigma = \{\mathsf{p}\}\cdot\emptyset$ is informative for $\Box\,\mathsf{p}$, because there exists the mapping L with $L(0) = \{\neg\Box\,\mathsf{p}\}$, $L(1) = \{\neg\Box\,\mathsf{p}, \neg\mathsf{p}\}$, $L(2) = \emptyset$.

[2] PSPACE-hardness can be shown by reducing LTL validity to safety checking: Any LTL formula φ is valid iff the formula $\psi = \varphi \vee \Diamond\,\mathsf{p}$ is a safety formula (with p an atomic proposition not occurring in φ), because $\psi \equiv$ `true` iff φ is valid, and `true` is a safety formula.

But for the equivalent formula $\psi = \Box(\mathsf{p} \vee (\mathtt{X}\,\mathsf{p} \wedge \mathtt{X}\,\neg\mathsf{p}))$, σ is not informative, but $\sigma' = \{\mathsf{p}\}\cdot\emptyset\cdot\emptyset$ would be.

Definition 4.5.2. With the notion of informativeness, we can group LTL formulas as follows:

- A safety formula ψ is *intentionally* safe iff all the bad prefixes for ψ are informative.
- A safety formula ψ is *accidentally* safe iff every computation that violates ψ has an informative bad prefix.
- A safety formula ψ is *pathologically* safe if there is a computation that violates ψ and has no informative bad prefix.

Proposition 4.5.1. ([KV99], Theorem 5.6) To decide whether a given formula ψ is pathologically safe is PSPACE-complete. □

Proposition 4.5.2. ([KV99], Theorem 5.3) A syntactically safe formula ψ is guaranteed to be intentionally or accidentally safe. □

Automata for informative prefixes

The general idea of [KV99] and [Lat02] is to generate a nondeterministic finite automaton $\mathcal{A}^{fin}_{inf}(\neg\psi)$ for the *informative bad prefixes* of ψ instead of for all the bad prefixes.

For intentionally safe formulas ψ, it follows directly from the definition that $\mathcal{L}(\mathcal{A}^{fin}_{inf}(\neg\psi)) = \mathit{badpref}(\psi)$, because all bad prefixes are informative.

For accidentally safe formulas ψ, $\mathcal{L}(\mathcal{A}^{fin}_{inf}(\neg\psi)) \subseteq \mathit{badpref}(\psi)$, because not every bad prefix is informative. But every computation violating ψ has at least one informative bad prefix, and therefore $\mathcal{L}(\mathcal{A}^{fin}_{inf}(\neg\psi)) = \mathcal{L}(\neg\psi)$, which is sufficient for our purposes.

It is clear that pathologically safe formulas can not be used with the automata for the informative bad prefixes.

We first define two constructions that help to formalize the algorithm.

Definition 4.5.3 (Restricted closure). [Lat02] The restricted closure $rcl(\psi)$ of a formula ψ is defined as the smallest set with the following properties:

- All temporal subformulas $\varphi \in cl(\psi)$ belong to $rcl(\psi)$.
- If a formula $\mathtt{X}\,\varphi$ belongs to $rcl(\psi)$ then $\varphi \in rcl(\psi)$.
- If $rcl(\psi)$ would be empty after application of the two rules above (no temporal operators in ψ), ψ is in $rcl(\psi)$.

The restricted closure $rcl(\psi)$ contains all subformulas that are not purely local and have to be tracked over multiple states. For the special case where ψ does not contain temporal operators, the third rule ensures that $rcl(\psi)$ is not empty by setting $rcl(\psi) = \psi$.

Definition 4.5.4. [Lat02] Let S be a subset of $cl(\psi)$. Then we define $sat(\varphi, S)$ as follows:

- $sat(\texttt{true}, S) = \texttt{true}$.
- $sat(\texttt{false}, S) = \texttt{false}$.
- $sat(\varphi, S) = \texttt{true}$ if $\varphi \in S$.
- $\varphi = \varphi_1 \vee \varphi_2 : sat(\varphi, S) = \texttt{true}$ if $sat(\varphi_1, S)$ or $sat(\varphi_2, S)$.
- $\varphi = \varphi_1 \wedge \varphi_2 : sat(\varphi, S) = \texttt{true}$ if $sat(\varphi_1, S)$ and $sat(\varphi_2, S)$.
- Otherwise $sat(\varphi, S) = \texttt{false}$.

$sat(\varphi, S)$ allows us to handle the propositional logic subformulas without adding them to $rcl(\varphi)$.

The basic idea for the algorithm is to use the elements of $2^{rcl(\psi)}$ as the states of the NFA and, starting with the empty set, generate the preceding states. For example, if a state S contains a subformula φ, then the preceding state S' will contain $\texttt{X}\,\varphi$. All states containing the formula ψ, for which the automaton is generated, will be start states.

Algorithm *Generate Informative Prefix Automaton*
Input: A formula ψ in positive normal form.
Output: A nondeterministic finite automaton $\mathcal{A}^{fin}_{inf}(\psi) = (Q, \Sigma, \delta, Q_0, F)$.

1. $\Sigma := 2^{AP}$;
2. $F := \{\emptyset\}$;
3. $Q := \{\emptyset\}$; ($*$ The set of states $*$)
4. $X := \{\emptyset\}$; ($*$ The set of unprocessed states $*$)
5. **while** $(X \neq \emptyset)$ **do**
6. $S :=$ "some set in X"; $X := X \setminus \{S\}$;
7. **for** each $\sigma \in 2^{AP}$ **do**
8. $S' := \sigma$; ($*$ The set of subformulas true in predecessor $*$)
9. **for** each $\varphi \in rcl(\psi)$ in increasing subformula order **do**
10. ($*$ The cases $\vee$ and $\wedge$ are used only when ψ contains no temporal operators. $*$)
11. **if** $\varphi = \psi_1 \vee \psi_2$ **then**
12. **if** $sat(\psi_1, S')$ or $sat(\psi_2, S')$ **then** $S' := S' \cup \{\varphi\}$
13. **if** $\varphi = \psi_1 \wedge \psi_2$ **then**
14. **if** $sat(\psi_1, S')$ and $sat(\psi_2, S')$ **then** $S' := S' \cup \{\varphi\}$
15. **if** $\varphi = \texttt{X}\,\psi_1$ **then**
16. **if** $sat(\psi_1, S)$ **then** $S' := S' \cup \{\varphi\}$
17. ($*$ ψ_1 true now $\rightarrow \texttt{X}\,\psi_1$ true in predecessor $*$)
18. **if** $\varphi = \psi_1\,\texttt{U}\,\psi_2$ **then**
19. **if** $sat(\psi_2, S')$ or ($sat(\psi_1, S')$ and $\varphi \in S$) **then** $S' := S' \cup \{\varphi\}$
20. **if** $\varphi = \psi_1\,\texttt{V}\,\psi_2$ **then**
21. **if** $sat(\psi_2, S')$ and ($sat(\psi_1, S')$ or $\varphi \in S$) **then** $S' := S' \cup \{\varphi\}$
22. **if** $\sigma \notin rcl(\psi)$ **then** $S' := S' \setminus \{\sigma\}$;
23. **od**

24. **if** $sat(\psi, S')$ **then** $Q_0 := Q_0 \cup \{S'\}$;
25. $\delta := \delta \cup \{(S', \sigma, S)\}$;
26. $X := X \cup \{S'\}$; $Q := Q \cup \{S'\}$;
27. **od**
28. **od**

Proposition 4.5.3. ([Lat02], Theorem 34) Let ψ be a safety formula. Given the negated formula $\neg\psi$ as input, the algorithm will generate a nondeterministic finite automaton $\mathcal{A}^{fin}_{inf}(\neg\psi)$ recognizing the informative bad prefixes of ψ. □

It is easy to see from the construction that the resulting NFA will have at most $2^{|rcl(\neg\psi)|}$ states, and an equivalent DFA or DBA with $\mathcal{O}(2^{2^{|rcl(\neg\psi)|}})$ states can be created as explained earlier.

The tool scheck

In scheck, the algorithm for the computation of the informative bad prefix automaton is implemented. As input, it expects a co-safety LTL formula ψ and generates a deterministic Büchi automaton $\mathcal{A}^{\texttt{scheck}}(\psi)$ with $\mathcal{L}(\mathcal{A}^{\texttt{scheck}}(\psi)) = \mathcal{L}(\psi)$. It can optionally perform a Hopcroft-style simulation reduction on the DFA.

Therefore, we have the following constructions available:

- For a safety LTL formula ψ:
  - for Rabin: $\mathcal{A}^{\texttt{scheck}}(\neg\psi) \xrightarrow{\text{Büchi to Streett}} \mathcal{A}^{Streett}(\neg\psi) \xrightarrow{\text{regard as Rabin}} \mathcal{A}^{Rabin}(\psi)$
  - for Streett: $\mathcal{A}^{\texttt{scheck}}(\neg\psi) \xrightarrow{\text{Büchi to Rabin}} \mathcal{A}^{Rabin}(\neg\psi) \xrightarrow{\text{regard as Streett}} \mathcal{A}^{Streett}(\psi)$
- For a co-safety LTL formula ψ:
  - for Rabin: $\mathcal{A}^{\texttt{scheck}}(\psi) \xrightarrow{\text{Büchi to Rabin}} \mathcal{A}^{Rabin}(\psi)$
  - for Streett: $\mathcal{A}^{\texttt{scheck}}(\psi) \xrightarrow{\text{Büchi to Streett}} \mathcal{A}^{Streett}(\psi)$

5 The tool `ltl2dstar`

Safra's construction and the optimizations detailed in Chapter 4 were implemented in the tool `ltl2dstar`[1] (**LTL** to **d**eterministic **St**reett **a**nd **R**abin automata), allowing a translation from LTL formulas to deterministic Rabin or Streett automata. It uses external LTL→NBA translators for the translation from LTL to NBA.

In this chapter, first the general structure of `ltl2dstar` and the testing done with the `lbtt` testbench is described. The next part presents the experimental results of the performance of `ltl2dstar` for a set of benchmark LTL formulas. Finally, a comparison of different LTL→NBA translators as used with `ltl2dstar` is presented.

5.1 Structure of `ltl2dstar`

The basic building blocks used by `ltl2dstar` for constructing the automata are:

- **Safra:**
  Generate a deterministic Rabin automaton for an LTL formula by creating an NBA with an external LTL→NBA translator and then applying Safra's construction on the NBA.
- **Scheck:**
  If the formula is syntactically (co-)safe, construct a deterministic Büchi automaton with the external tool `scheck` (Section 4.5) and convert it to a DRA.
- **Union:**
  If the formula has the form $(\varphi_1) \vee (\varphi_2)$, we construct DRA for φ_1 and φ_2 and create the union of the two automata (Section 4.3).

These basic blocks can be combined to get more complex blocks:

- **Rabin:**
  Generate a DRA with **Safra**, **Scheck** or **Union**.
- **Streett:**
  Generate a DRA with **Rabin** for the negated formula, and then consider it as a Streett automaton (Section 4.4).
- **Deterministic ω-automaton:**
  If the user accepts both DRA and DSA, generate both a DRA with **Rabin** and a DSA with **Streett** and return the smaller automaton. If the user wants only DRA or only DSA, generate this automaton.

[1] `http://www.ltl2dstar.de/`

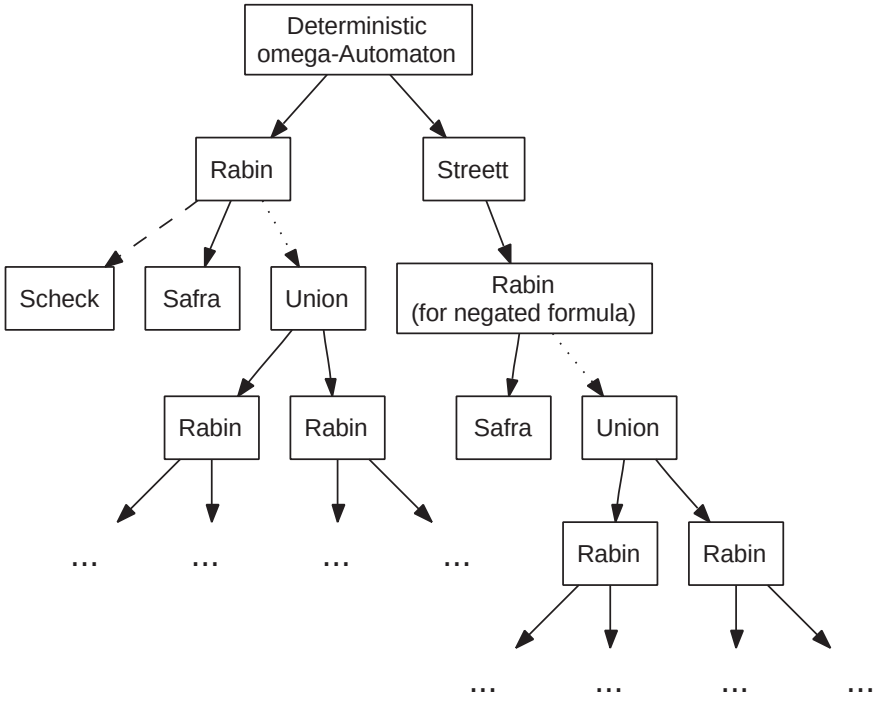


Figure 5.1: Tree for the different construction possibilities

Dashed line: Only if formula is syntactically (co-)safe.
Dotted line: Only if the LTL formula has $\vee$ as top-level operator.

These blocks form a tree of possible constructions, as shown in Figure 5.1. When we have the choice between multiple automata, for example between the automata generated by **Scheck**, **Union** and **Safra** in **Rabin**, we chose the smallest automaton of the generated automata. This way, we get the smallest possible (with these constructions) automaton as the final result.

Limiting If one of the constructions generates a small automaton, but another construction generates a significantly larger automaton, we have to wait for the second construction to finish if we want to generate all possible automata. If we only need the smallest, we can abort the second construction as soon as it is clear that its automaton will be larger than the already existing automaton. As long as we do not use optimizations which operate on the automaton after it is fully generated (e.g. bisimulation), we can abort the second construction as soon as the size of the generated automaton is superior to the already existing automaton. For the **Union** construction, when one of the two DRA is bigger than the already existing smallest DRA, we can abort the **Union** construction, as the union automaton will be at least as big as the smallest of the two DRA for the subformulas.

If we use optimizations on the complete automaton, we cannot abort directly, as the optimization might ultimately result in a smaller automaton than the existing one. If a

heuristic approach is acceptable, we can set a limit that is larger than the size n of the existing automaton ($\alpha \cdot n$, for example $2n$ or $10n$) to allow for the possibility that the optimization might shrink the automaton to $\frac{1}{\alpha}$ of the original size.

Limiting the construction of the automata like this is obviously sensitive to the order in which the different constructions are carried out. If the construction generating the largest automaton is used first, we still have to wait for it to finish. Unfortunately, there is no general optimal order for the constructions: For example, if for formula ψ the **Rabin** construction results in a smaller automaton than **Streett**, for the formula $\neg\psi$ the situation is exactly reversed.

As a heuristic for a good ordering, we can consider the size of the NBA used for the different blocks and start with the construction having the smaller NBA.

5.2 Testing with lbtt

lbtt[2] is a testbench for LTL→NBA translators by Heikki Tauriainen [Tau00]. It provides empirical testing of LTL→NBA translators by randomly generating LTL formulas with multiple translators and comparing the generated nondeterministic Büchi automata. Ideally, for two translators t_1, t_2 and a formula ψ, it would check that $\mathcal{L}(NBA_{t_1}(\psi)) = \mathcal{L}(NBA_{t_2}(\psi))$. The check for language equality is computationally expensive, so it verifies that $\mathcal{L}(NBA_{t_1}(\psi)) \cap \mathcal{L}(NBA_{t_2}(\neg\psi)) = \emptyset$ instead, which is simpler to check (generate intersection automaton and perform emptiness check) but will not detect every single error.

As the tool was designed to check nondeterministic Büchi automata and is not capable of handling DRA or DSA directly, the DRA and DSA have to be translated to nondeterministic Büchi automata, which can be done using the procedures in 2.8.2 and 2.8.3. This allows the ltl2dstar tool to be used as just another LTL→NBA translator, as shown in Figure 5.2.

During the implementation of ltl2dstar, testing with lbtt was very valuable and lead to the discovery of several bugs.

5.3 Benchmarks

To analyze the performance of ltl2dstar and the proposed optimizations, we have to choose a set of LTL formulas to use, preferably formulas that are representative of formulas often used in practice or that represent interesting difficult cases. We will use some formulas that were used in the evaluation of LTL→NBA translators, some formulas representing often used patterns and random formulas.

[2] http://www.tcs.hut.fi/Software/lbtt/

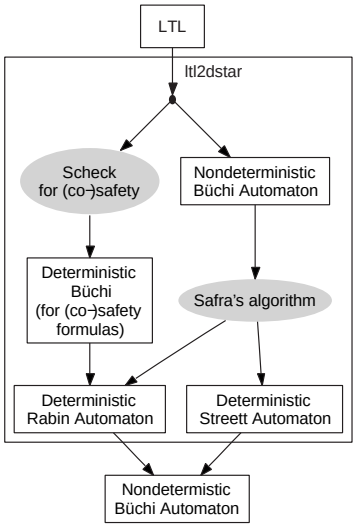


Figure 5.2: The use of ltl2dstar as an LTL→NBA translator, with automaton types.

5.3.1 The formulas from [SB00] and [EH00]

[EH00] presents 12 formulas, [SB00] presents 27 formulas to evaluate the performance of their LTL→NBA translators, shown in Table 5.1 and in Table 5.2 .

$p\,\mathtt{U}(q \wedge \Box r)$	$p\,\mathtt{U}(q \wedge \mathtt{X}(r\,\mathtt{U}\,s))$
$p\,\mathtt{U}(q \wedge \mathtt{X}(r \wedge (\Diamond(s \wedge \mathtt{X}(\Diamond(t \wedge \mathtt{X}(\Diamond u \wedge \mathtt{X}\Diamond v)))))))$	$\Diamond(p \wedge \mathtt{X}\Box q)$
$\Diamond(p \wedge \mathtt{X}(q \wedge \mathtt{X}(\Diamond r)))$	$\Diamond(p \wedge \mathtt{X}(p\,\mathtt{U}\,r))$
$(\Diamond\Box q) \vee (\Diamond\Box p)$	$\Box(p \rightarrow (q\,\mathtt{U}\,r))$
$\Diamond(p \wedge \mathtt{X}\Diamond(q \wedge \mathtt{X}\Diamond(r \wedge \mathtt{X}\Diamond s)))$	$\Box\Diamond p \wedge \Box\Diamond q \wedge \Box\Diamond r \wedge \Box\Diamond s \wedge \Box\Diamond t$
$(p\,\mathtt{U}\,q\,\mathtt{U}\,r) \vee (q\,\mathtt{U}\,r\,\mathtt{U}\,p) \vee (r\,\mathtt{U}\,p\,\mathtt{U}\,q)$	$\Box(p \rightarrow (q\,\mathtt{U}(\Box r \vee \Box s)))$

Table 5.1: The 12 formulas from [EH00]

5.3.2 Patterns for specifications

In [DAC99] Dwyer, Avrunin and Corbett propose a system of patterns for specifying properties[3] in LTL and other formalisms. Their research indicates that many of the specifications used in practice can be classified as instances of one of their patterns.

The patterns consist of *Occurrence Patterns* (Absence, Universality, Existence, Bounded Existence) that are concerned with the occurrence of a given state in the system and *Or-*

[3] http://patterns.projects.cis.ksu.edu/

$p \mathtt{U} q$	$p \mathtt{U} (q \mathtt{U} r)$
$\neg(p \mathtt{U} (q \mathtt{U} r))$	$(\Box\Diamond p) \rightarrow (\Box\Diamond q)$
$(\Diamond p) \mathtt{U} (\Box q)$	$(\Box p) \mathtt{U} q$
$\neg(\Diamond\Diamond p \leftrightarrow \Diamond p)$	$\neg(\Box\Diamond p \rightarrow \Box\Diamond q)$
$\neg(\Box\Diamond p \leftrightarrow \Box\Diamond q)$	$p \mathtt{V} (p \vee q)$
$((\mathtt{X} p) \mathtt{U} (\mathtt{X} q)) \vee \neg \mathtt{X}(p \mathtt{U} q)$	$((\mathtt{X} p) \mathtt{U} q) \vee \neg \mathtt{X}(p \mathtt{U} (p \wedge q))$
$\Box(p \rightarrow \Diamond q) \wedge (((\mathtt{X} p) \mathtt{U} q) \vee \neg \mathtt{X}(p \mathtt{U} (p \wedge q)))$	$\Box(p \rightarrow \Diamond q) \wedge (((\mathtt{X} p) \mathtt{U} (\mathtt{X} q)) \vee \neg \mathtt{X}(p \mathtt{U} q))$
$\Box(p \rightarrow \Diamond q)$	$\neg\Box(p \rightarrow \mathtt{X}(q \mathtt{V} r))$
$\neg((\Box\Diamond p) \vee \Diamond\Box q)$	$\Box(\Diamond p \wedge \Diamond q)$
$\Diamond p \wedge \Diamond \neg p$	$((\mathtt{X} q) \wedge r) \mathtt{V} \mathtt{X}(((s \mathtt{U} p) \mathtt{V} r) \mathtt{U} (s \mathtt{V} r))$
$(\Box(q \vee \Box\Diamond p) \wedge \Box(r \vee \Box\Diamond \neg p)) \vee \Box q \vee \Box r$	$(\Box(q \vee \Diamond\Box p) \wedge \Box(r \vee \Diamond\Box \neg p)) \vee \Box q \vee \Box r$
$\neg((\Box(q \vee \Box\Diamond p) \wedge \Box(r \vee \Box\Diamond \neg p)) \vee \Box q \vee \Box r)$	$\neg((\Box(q \vee \Diamond\Box p) \wedge \Box(r \vee \Diamond\Box \neg p)) \vee \Box q \vee \Box r)$
$\Box(q \vee \mathtt{X}\Box p) \wedge \Box(r \vee \mathtt{X}\Box \neg p)$	$\Box(q \vee (\mathtt{X}(p \wedge \mathtt{X} \neg p)))$
$(p \mathtt{U} p) \vee (q \mathtt{U} p)$	

Table 5.2: The 27 formulas from [SB00]

der Patterns (Precedence, Response, Precedence Chain, Response Chain) that are concerned with the relative order of states. Some of these patterns can be further modified by giving additional constraints.

The 11 instances of the patterns that were used are shown in Table 5.3.

Each of the patterns is modified by *Scope* (Global, Before R, After Q, Between Q and R, After Q until R), which specifies the duration of the pattern specification.

Each pattern combined with a scope leads to one LTL formula, for example the *Absence* pattern with *Global* scope can be expressed in LTL with the formula $\varphi = \Box\neg p$, and the *Universality* pattern with scope *After Q* with the formula $\varphi = \Box(q \rightarrow \Box p)$.

Therefore, these 11 patterns with the 5 different scopes lead to 55 LTL formulas[4] that can be used in benchmarking and should cover many interesting properties.

5.3.3 Random formulas

The `lbtt` testbench was used to generate a set of 100 and a set of 1000 random formulas.

The parameters for the random formula generator were as follows, producing LTL

[4]The LTL formulas were taken from
`http://patterns.projects.cis.ksu.edu/documentation/patterns/ltl.shtml`

Pattern	Description
Abscence	P is false
Universality	P is true
Existence	P becomes true
Bounded Existence (2)	P becomes true at most 2 times
Precedence	S precedes P
Response	S responds to P
Precedence Chain (2-1)	S, T precedes P (2 cause - 1 effect)
Precedence Chain (1-2)	P precedes (S, T) (1 cause - 2 effects)
Response Chain (2-1)	P responds to S,T (2 stimuli - 1 effect)
Response Chain (1-2)	S,T responds to P (1 stimulus - 2 effects)
Constrained Response (1-2)	S,T without Z responds to P (1 stimulus - 2 effects)

Table 5.3: The patterns

formulas with a size between 6 and 12 and at most 4 atomic propositions:

```
FormulaOptions {
Size = 6...12
Propositions = 4
PropositionPriority = 90
TruePriority = 5
FalsePriority = 5
DefaultOperatorPriority = 0
NotPriority = 15
NextPriority = 10
FinallyPriority = 5
GloballyPriority = 5
AndPriority = 10
OrPriority = 10
ImplicationPriority = 5
EquivalencePriority = 5
UntilPriority = 10
ReleasePriority = 10
...}
```

5.4 Experimental results

In this section, we will take a look at the practical performance of Safra's construction as implemented in `ltl2dstar` for our benchmark formulas and consider the effect of the different optimizations introduced in Chapter 4. At first, we will consider the optimizations separately and later consider the optimizations in combination and take a detailed look at the size of DRA and DRA for the individual formulas from [SB00], [EH00] and for the pattern formulas.

All experiments were conducted on a Pentium-M 1.5 GHz with 512 MB RAM, running Linux.

The LTL→NBA translator chosen for all the following experiments was `ltl2ba` (see Section 5.5), because using it, `ltl2dstar` was able to calculate DRA for all benchmark formulas, it is fast and the automata sizes are reasonable. In Section 5.5 we will compare `ltl2ba` to other LTL→NBA translators in the context of `ltl2dstar`.

The following notations will be often used in the tables showing the results:

$\Sigma(\|\mathcal{A}\|)$	The total number of states for all automata
$\Sigma(t)$	The total running time

It should be noted that the time keeping of the running time is not exact, so small differences in the running time should not be overestimated.

The following sets of formulas were used:

Name	**Number of formulas**	**Defined in**
[SB00], [EH00]	39	Section 5.3.1
Patterns	55	Section 5.3.2
100 random	100	Section 5.3.3
1000 random	1000	Section 5.3.3

5.4.1 On-the-fly optimizations

To allow a comparison of the relative effectiveness of the on-the-fly optimizations (Section 4.1), the four sets of LTL benchmark formulas were used as input to `ltl2dstar`. First, all on-the-fly optimizations were disabled, resulting in the standard Safra's construction. Then, a run was carried out with all on-the-fly optimizations enabled. To assess the individual impact of the different optimizations, for every optimization a run was carried out with just this optimization disabled. Then the difference with the size of the DRA with all on-the-fly optimizations enabled (increase in size of the DRA with the optimization disabled) was calculated. The sums of the differences for all the optimizations are shown in Table 5.4, a bigger value means a bigger impact of the particular optimization.

The optimizations that are not on-the-fly, like bisimulation and the union construction, were disabled.

	[SB00],[EH00]	Patterns	100 random	1000 random
$\Sigma(\|\mathcal{A}\|)$ with all opt.	926	246455	642	6743
No optimization	+394	+94666	+983	+36632
No 'Trueloop detection'	+195	+1467	+651	+26254
No 'All successors accepting'	+113	+95	+38	+400
No 'Node renaming'	+40	+92687	+8	+90
No 'Reordering'	+16	+0	+8	+31
$\Sigma(t)$ (no opt.)	0.48 s	358.50 s	0.70 s	12.89 s
$\Sigma(t)$ (all opt.)	0.39 s	270.14 s	0.56 s	5.57 s

Table 5.4: Results for the on-the-fly optimizations.

Analysis

The effectiveness of all the on-the-fly combinations combined was highest for the random formulas, where they resulted in a reduction by around 60% for the 100 and 84% for the 1000 random formulas. This is mostly due to the "Trueloop detection", followed by "All successors accepting". For the formulas from [SB00] and [EH00], the overall reduction is lower (around 30%) and "All successors accepting" plays a bigger role than for the random formulas. The pattern formulas, while also having an overall reduction of around 30%, exhibit a completely different behavior: Here, the "Node renaming" is almost exclusively responsible for the overall reduction. It seems that "Node renaming" works better for bigger automata, which can be explained by the fact that a single tree that can be matched early in the construction can result in a huge reduction of states, as an incompatible naming would result in the duplication (also with different names) of

many of the successor states. The bigger the automaton gets, the more states would be duplicated, so ”Node renaming” has a bigger effect.

The ”Reordering” does not seem to have a big effect.

Another interesting point is the running time: With all on-the-fly optimizations enabled, the running time was shorter than with the optimizations disabled, the benefit of handling fewer states far outweighs the additional effort needed to carry out the optimizations. The biggest reduction in running time could be observed for the 1000 random formulas with more than 50%; for the other formulas sets, the reduction was between 20% and 30%.

5.4.2 Union construction

Table 5.5 shows a comparison of the automata size when the union construction (Section 4.3) is used to handle formulas where the top-level operator is $\vee$ (or a similar operator like $\rightarrow$). The on-the-fly optmimizations were enabled.

The first three rows show the sum of the automata sizes only for the formulas that have this special form, first with the union construction disabled (”Without Union”), then the results when the union construction is always used if possible (”Only Union”) and at last the automata where the automata generated by the union construction were only chosen if they were smaller than the automata generated the ”normal way” (”With Union”).

The next two rows show the results for all formulas, once without the union construction and then with the union construction enabled but only used when the automata are smaller. The sixth row shows the difference between the two, being the reduction that is made possible by using the union construction.

		[SB00],[EH00]	Patterns	100 random	1000 random
$\Sigma(\lvert\mathcal{A}_{\varphi_1\vee\varphi_2}\rvert)$	Without union	199	228	135	1478
$\Sigma(\lvert\mathcal{A}_{\varphi_1\vee\varphi_2}\rvert)$	Only union	135	208	133	1445
$\Sigma(\lvert\mathcal{A}_{\varphi_1\vee\varphi_2}\rvert)$	With union	133	208	131	1436
$\Sigma(\lvert\mathcal{A}_{\text{all}}\rvert)$	Without union	926	246455	642	6743
$\Sigma(\lvert\mathcal{A}_{\text{all}}\rvert)$	With union	860	246435	638	6701
$\Sigma(\lvert\mathcal{A}_{\text{all}}\rvert)$	Difference	-66 (-7.7%)	-20 (-0.0%)	-4 (-0.6%)	-42 (-0.6%)
$\varphi = \varphi_1 \vee \varphi_2$		8 (20.5%)	17 (30.9%)	25 (25.0%)	270 (27.0%)
Union construction is smaller		5 (62.5%)	15 (88.2%)	2 (8.0%)	17 (6.3%)
Union construction is bigger		2 (25.0%)	0 (0.0%)	1 (4.0%)	9 (3.3%)

Table 5.5: Results for the union construction.

The last three rows show how many formulas had the special form needed for the union construction and for how many of these formulas the DRA generated using it were smaller or bigger than the ”normal” DRA.

Analysis

If we only consider the formulas with $\vee$ as their top operator, the union construction provides a modest reduction in the size of the automata, ranging from around 3% for the random formulas, around 9% for the pattern formulas to more than 30% for the formulas from [SB00] and [EH00]. But considering all formulas, the overall effect is smaller, ranging from a reduction by around 7% for the formulas from [SB00] and [EH00] to less than 1% for the random formulas, and to almost nothing for the pattern formulas, which is due to the big automata for formulas without $\vee$ as their top operator.

5.4.3 Bisimulation

	[SB00],[EH00]		Patterns		100 random		1000 random	
	$\Sigma(\|\mathcal{A}\|)$	$\Sigma(t)$	$\Sigma(\|\mathcal{A}\|)$	$\Sigma(t)$	$\Sigma(\|\mathcal{A}\|)$	$\Sigma(t)$	$\Sigma(\|\mathcal{A}\|)$	$\Sigma(t)$
No opt., no bisim.	1320	0.5 s	341121	362.5 s	1625	0.7 s	43375	12.9 s
No opt., with bisim.	-636	0.5 s	-217780	373.1 s	-631	0.7 s	-29990	12.9 s
No bisimulation	860	0.4 s	246435	272.8 s	638	0.7 s	6701	7.1 s
With bisimulation	-474	0.4 s	-142792	281.1 s	-132	0.7 s	-1383	7.2 s

Table 5.6: Results for the bisimulation optimization

To evaluate the performance of the bisimulation optimization (Section 4.2.2), we compare the difference in the size of the DRA with bisimulation disabled and enabled. The first comparison was conducted with no other optimizations enabled, i.e. plain Safra’s construction. The second comparison was conducted with the on-the-fly optimizations and the union construction enabled. Table 5.6 show the comparison and additionally the running time.

Analysis

Bisimulation provides a surprisingly big reduction in the size of the automata at a very moderate cost (less than 3% increase in running time). For the pattern formulas, the effect is highest, with reductions by around 60%. For the formulas from [SB00] and [EH00] the reductions are around 50%. For these two formula sets, bisimulation works roughly as well when the other optimizations are enabled, leading to a combined reduction of around 70%!

For the random formulas, bisimulation reduces the already optimized automata by an additional 20%, which improves the (already high) reduction from the on-the-fly optimizations for the 1000 formulas to an impressive 90%.

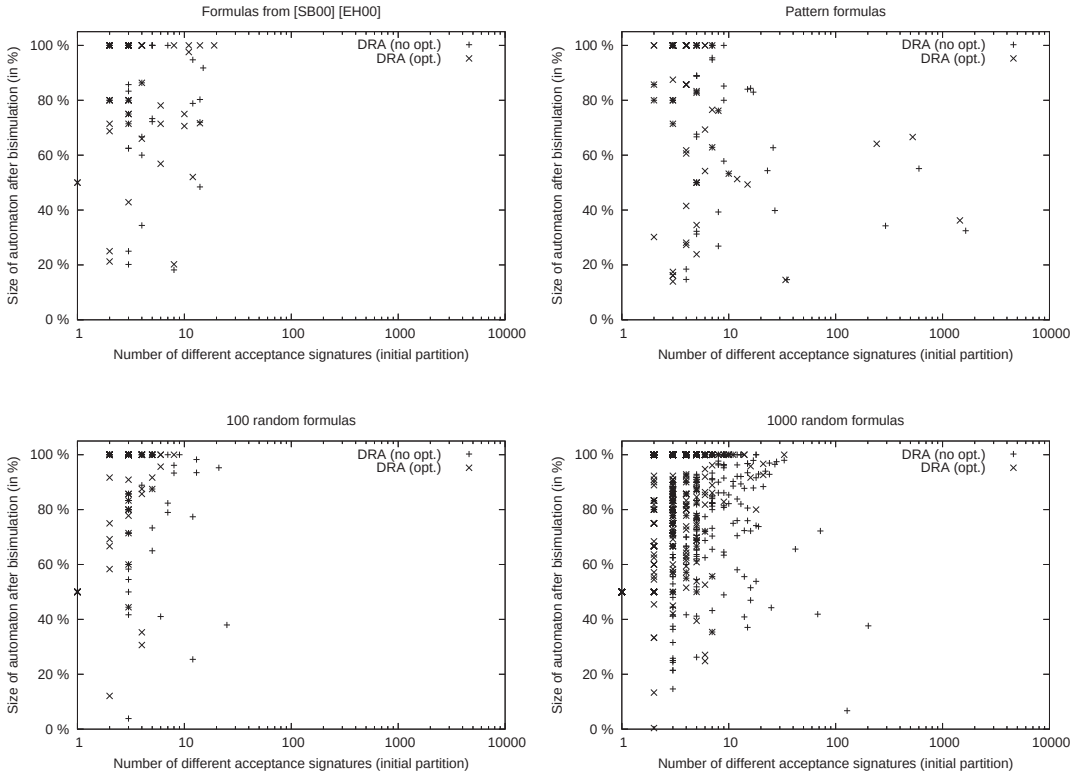


Figure 5.3: Relationship between the number of different acceptance signatures and the number of states in the quotient automaton (in percent of the number of states of the original automaton, 100% equals no reduction due to bisimulation).

As noted in the description of the bisimulation reduction, it is clear that the number of different acceptance signatures in the original automaton plays an important role, as it determines the number of elements in the initial partition. Figure 5.3 shows the relationship between the number of different acceptance signatures and reduction in the number of states of the quotient automaton. The data is from the runs with bisimulation enabled, so it contains automata generated with enabled and disabled on-the-fly optimizations.

These graphs are interesting, because they show that even for very big initial partitions, the reductions can be significant. This indicates that Safra's construction produces many redundant states in big DRA, which would be interesting to investigate further, perhaps leading to additional on-the-fly optimizations that avoid creating this redundant states in the first place.

	[SB00],[EH00]	Patterns	100 random	1000 random
$\Sigma(\lvert\mathcal{A}_{\text{(co)safe}}\rvert)$ No scheck	82	132	299	3049
$\Sigma(\lvert\mathcal{A}_{\text{(co)safe}}\rvert)$ Only scheck	67	66	255	2410
$\Sigma(\lvert\mathcal{A}_{\text{(co)safe}}\rvert)$ With scheck	66	66	251	2357
$\Sigma(\lvert\mathcal{A}_{\text{all}}\rvert)$ No scheck	860	246435	638	6701
$\Sigma(\lvert\mathcal{A}_{\text{all}}\rvert)$ With scheck	844	246365	590	6007
Difference	-16 (-1.9%)	-70 (-0.0%)	-48 (-8.1%)	-694 (-11.6%)
φ is syntactically (co)safe	17 (43.6%)	16 (29.1%)	61 (61.0%)	617 (61.7%)
DRA with scheck is smaller	5 (29.4%)	14 (87.5%)	15 (24.6%)	139 (22.5%)
DRA with scheck is bigger	1 (5.9%)	0 (0.0%)	4 (6.6%)	58 (9.4%)

Table 5.7: Results: Using the external scheck tool for handling syntactically safe and co-safe formulas.

5.4.4 Using scheck for (co-)safety formulas

Table 5.7 shows a comparison of the automata size when the scheck tool was used to handle syntactically safe and co-safe LTL formulas (Section 4.5). The first three rows show the sum of the automata sizes only for the formulas that are syntactically (co-)safe, first without using scheck (”Without scheck”), then the results of scheck (”Only scheck”) and at last the automata where the automata from scheck were only chosen if they were smaller than the automata generated the ”normal way” (”With scheck”).

The next two rows show the results for all formulas, once without scheck and then with scheck enabled, but only used when the automata are smaller. The sixth row shows the reduction is made possible by using scheck.

The last three rows show how many formulas were syntactically (co-)safe and for how many of these formulas the DRA generated with scheck were smaller or bigger than the ”normal” DRA.

Analysis

If we only consider the syntactically (co-)safe formulas, the use of scheck provides a reduction of around 50% for the pattern formulas and of around 20% for the other formulas. Considering all formulas, the overall reduction is reduced to around 10% for the random formulas, 2% for the formulas from [SB00] and [EH00] and for the same reasons as with the union construction to almost nothing for the pattern formulas.

	[SB00],[EH00]	Pattern formulas	100 random	1000 random
$\Sigma(\lvert\mathcal{A}_{\text{DRA}}\rvert)$	386	103643	506	5319
$\Sigma(\lvert\mathcal{A}_{\text{DSA}}\rvert)$	614	6763	660	5265
$\Sigma(\mathit{min}(\lvert\mathcal{A}_{\text{DRA}}\rvert, \lvert\mathcal{A}_{\text{DSA}}\rvert))$	268	6401	475	4487
DRA / DSA is min.	59.0% / 48.7%	12.7% / 90.9%	81.0% / 72.0%	71.6% / 72.4%
$\Sigma(\lvert\text{NBA}(\varphi)\rvert)$	205	435	389	3574
$\Sigma(\lvert\text{NBA}(\neg\varphi)\rvert)$	176	271	399	3609

Table 5.8: Comparison of DRA and DSA

5.4.5 Both DRA and DSA

What are the benefits of accepting both deterministic Rabin and Streett automata (Section 4.4)? To answer this question, for all benchmark formulas a DRA and a DSA (by generating the DRA for the negated formula) were generated. All on-the-fly optimizations, union construction and bisimulation were enabled, but `scheck` was disabled.

The results are shown in Table 5.8. The first two rows show the sum of the sizes of the DRA and DSA respectively, $\Sigma(\lvert\mathcal{A}_{\text{DRA}}\rvert)$ and $\Sigma(\lvert\mathcal{A}_{\text{DSA}}\rvert)$. As the smaller automaton of the DRA and DSA is returned when the user accepts both DRA and DSA, the third row is the sum of the size of this minimum automaton, $\Sigma(\mathit{min}(\lvert\mathcal{A}_{\text{DRA}}\rvert, \lvert\mathcal{A}_{\text{DSA}}\rvert))$. The fourth row is the percentage how often the DRA and DSA were the minimum automaton. As DRA and DSA can both be the minimum automaton at the same time when both have the same size, the two percentages add up to a number greater than 100%.

The last two rows show the sizes of the NBA for the positive formula (used for the DRA) and the negated formula (used for the DSA).

Analysis

The most interesting result here is the enormous difference for the pattern formulas, where in almost all cases the Streett automata are smaller than the Rabin automata. A detailed look at the DRA and DSA sizes for the pattern formulas can be found in Figure 5.4. It can clearly be seen that the *After Q until R* scope with the different *Response* patterns results in very big Rabin automata with smaller Streett automata. Yet, the same scope used with the *Precedence Chain (1 stimulus, 2 responses)* pattern produces a ten times larger Streett automaton.

For the formulas from [SB00] and [EH00], all the DSA together are bigger than the DRA. For the random formulas, accepting both DRA and DSA results in smaller automata in about 20% - 30% of the cases and DRA and DSA have roughly the same sizes, as is to be expected.

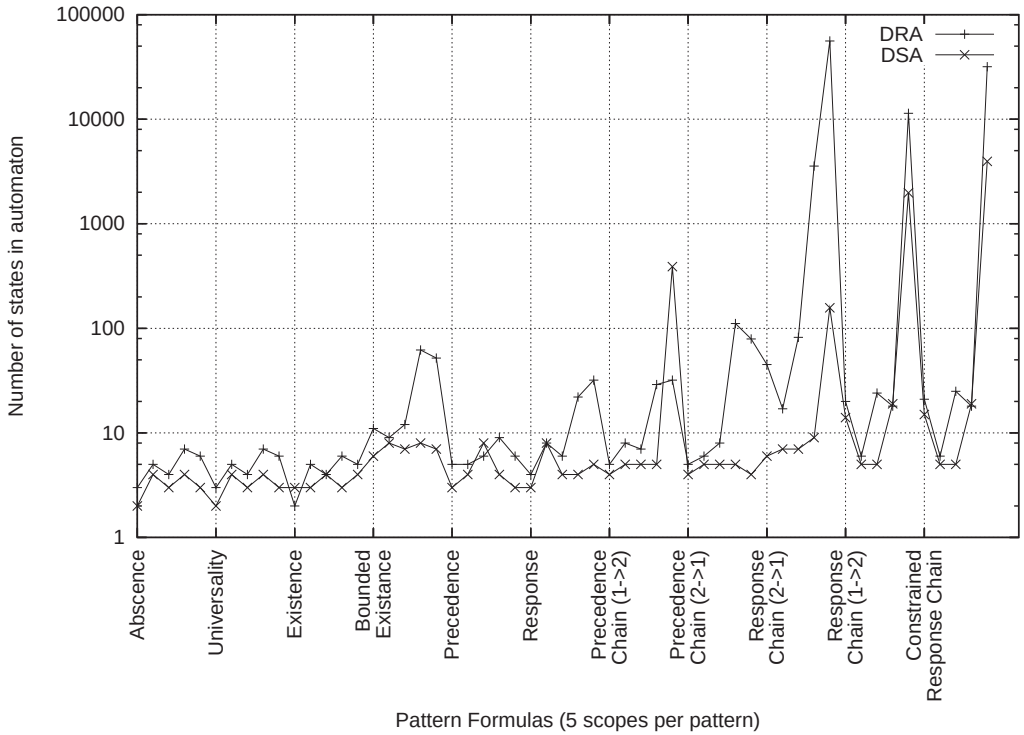


Figure 5.4: DRA and DSA sizes for the pattern formulas. The scopes for each pattern are ordered *Global*, *Before R*, *After Q*, *Between Q and R*, *After Q until R*.

It is interesting to note that for the formulas from [SB00] and [EH00], the NBA for the positive formulas are collectively bigger than the NBA for the negative formulas, yet for the deterministic Rabin and Streett automata this is reversed. For the pattern formulas, the NBA for the positive formulas are collectively only 60% bigger than the NBA for the negative formulas, the DRA however are 15 times as large as the DSA, showing that relatively small differences in the NBA size can lead to big differences in the deterministic automata.

5.4.6 All optimizations combined

After having looked at the different optimizations separately, we will now look at the overall performance of the `ltl2dstar` tool.

Limiting

First, we will consider the effect of aborting the creation of an automaton if a smaller automaton was already generated with an alternative method (like **Streett**, **Union** or **scheck**) as described in Section 5.1. Table 5.9 shows the results for runs of `ltl2dstar`

with limiting disabled (thus using all available approaches to generate an automaton and choosing the smallest afterward) and then with limiting enabled and for different values of α, the factor used to expand the limit to temporarily allow bigger automata which might later be made smaller using bisimulation. All optimizations were enabled, including bisimulation and generating DRA or DSA.

	[SB00],[EH00]		Patterns		100 random		1000 random	
	$\Sigma(\lvert\mathcal{A}\rvert)$	$\Sigma(t)$	$\Sigma(\lvert\mathcal{A}\rvert)$	$\Sigma(t)$	$\Sigma(\lvert\mathcal{A}\rvert)$	$\Sigma(t)$	$\Sigma(\lvert\mathcal{A}\rvert)$	$\Sigma(t)$
No Limit	268	133.35 s	6399	278.97 s	474	1.53 s	4480	15.32 s
Limit ($\alpha = 1$)	347	0.93 s	6406	13.62 s	476	1.51 s	4529	15.28 s
Limit ($\alpha = 2$)	269	0.94 s	6402	19.88 s	476	1.53 s	4503	15.47 s
Limit ($\alpha = 5$)	268	1.04 s	6401	40.42 s	474	1.54 s	4480	15.52 s
Limit ($\alpha = 10$)	268	1.08 s	6399	74.67 s	474	1.55 s	4480	15.44 s

Table 5.9: Sizes of the automata and running time using limiting and for different values of α. All optimizations enabled, including generation of DRA or DSA.

As can be seen, using limiting can dramatically improve the running time, at the cost of potentially missing the opportunity that an automaton may be larger during construction but is reduced significantly afterwards by bisimulation. With increasing α, the automata sizes approach the sizes of the automata generated without limiting and the running time increases, but is either roughly the same (as for the random formulas) or still lower ([SB00],[EH00] and pattern formulas) than without limiting.

The proposed heuristic of starting with the construction of the DRA or of the DSA depending on which of the two has the smaller NBA seems to work well.

Performance of `ltl2dstar`

Table 5.10 shows a comparison between the sizes of the automata generated using Safra's construction without optimization and the sizes of DRA and DSA generated with all optimizations enabled (including limiting, with $\alpha = 10$), and the reduction the optimizations provide compared to Safra's construction without optimization.

If only DRA are generated, the introduced optimizations had the effect of an overall reduction of around 70% and more. If the user has the flexibility of handling both DRA and DSA, this has a big impact on the automata size; for the pattern formulas the overall automata sizes are 50 times smaller than with Safra's construction and DRA.

The running time is acceptable, especially if DRA or DSA can be returned and limiting with a small α is enabled.

	DRA (no opt.)	DRA (opt.)	DSA (opt.)	DRA or DSA (opt.)
[SB00],[EH00] (39 formulas)				
$\Sigma(\|\mathcal{A}\|)$	1320	384	614	268
Reduction		-936 (-70.9 %)	-706 (-53.5 %)	-1052 (-79.7 %)
Time $\Sigma(\|t\|)$	1.02 s	0.96 s	0.88 s	1.04 s
Patterns (55 formulas)				
$\Sigma(\|\mathcal{A}\|)$	341121	103623	6763	6399
Reduction		-237498 (-69.6 %)	-334358 (-98.0 %)	-334722 (-98.1 %)
Time $\Sigma(\|t\|)$	358.98 s	276.78 s	11.37 s	73.83 s
100 random (100 formulas)				
$\Sigma(\|\mathcal{A}\|)$	1625	498	660	474
Reduction		-1127 (-69.4 %)	-965 (-59.4 %)	-1151 (-70.8 %)
Time $\Sigma(\|t\|)$	0.66 s	1.13 s	0.70 s	1.49 s
1000 random (1000 formulas)				
$\Sigma(\|\mathcal{A}\|)$	43375	5232	5262	4480
Reduction		-38143 (-87.9 %)	-38113 (-87.9 %)	-38895 (-89.7 %)
Time $\Sigma(\|t\|)$	12.58 s	11.59 s	6.63 s	14.91 s

Table 5.10: Automata sizes with all optimizations enabled and the reduction compared to the aumata generated with the standard Safra's construction ("DRA no opt.")

Automata sizes for selected formulas

To allow a detailed look at the automata sizes for individual formulas, Table 5.11 shows the sizes of the DRA and DSA for the formulas from [EH00], Table 5.12 for the formulas from [SB00] and Table 5.13 for the pattern formulas. For comparison, the sizes of the NBA for the positive formulas is presented, too.

It can be seen that for most formulas either the DRA or the DSA has less than 50 states. A big exception are the automata for the formulas for the *Response Chain (1-2)* and the *Constrained Response (1-2)* patterns with the *After Q until R* scope, which had about 2000 and 4000 states.

Comparing the deterministic automata sizes with the sizes of the NBA, it can be seen that for many formulas the sizes are comparable, sometimes the deterministic automata are even smaller.

	NBA	DRA	DSA
$p\,\mathtt{U}(q \wedge \Box r)$	2	**5 (2)**	7 (1)
$p\,\mathtt{U}(q \wedge \mathtt{X}(r\,\mathtt{U}\,s))$	3	**5 (1)**	6 (1)
$p\,\mathtt{U}(q \wedge \mathtt{X}(r \wedge (\Diamond(s \wedge \mathtt{X}(\Diamond(t \wedge \mathtt{X}(\Diamond u \wedge \mathtt{X}\Diamond v)))))))$	7	**9 (1)**	10 (1)
$\Diamond(p \wedge \mathtt{X}\Box q)$	2	**3 (1)**	8 (1)
$\Diamond(p \wedge \mathtt{X}(q \wedge \mathtt{X}(\Diamond r)))$	4	**4 (1)**	5 (1)
$\Diamond(p \wedge \mathtt{X}(p\,\mathtt{U}\,r))$	3	**3 (1)**	4 (1)
$(\Diamond\Box q) \vee (\Diamond\Box p)$	3	9 (2)	**5 (1)**
$\Box(p \rightarrow (q\,\mathtt{U}\,r))$	2	5 (1)	**4 (2)**
$\Diamond(p \wedge \mathtt{X}\Diamond(q \wedge \mathtt{X}\Diamond(r \wedge \mathtt{X}\Diamond s)))$	5	**5 (1)**	6 (1)
$\Box\Diamond p \wedge \Box\Diamond q \wedge \Box\Diamond r \wedge \Box\Diamond s \wedge \Box\Diamond t$	6	**11 (1)**	243 (5)
$(p\,\mathtt{U}\,q\,\mathtt{U}\,r) \vee (q\,\mathtt{U}\,r\,\mathtt{U}\,p) \vee (r\,\mathtt{U}\,p\,\mathtt{U}\,q)$	8	**3 (1)**	**3 (1)**
$\Box(p \rightarrow (q\,\mathtt{U}(\Box r \vee \Box s)))$	4	25 (3)	**24 (2)**

Table 5.11: Number of states of DRA and DSA for the formulas from [EH00] and number of states of the NBA for the positive formulas (with `ltl2ba`). In parentheses, the number of acceptance pairs in Ω. The smaller of the two automata is marked in bold.

	NBA	DRA	DSA
$p \mathtt{U} q$	2	**3 (1)**	4 (1)
$p \mathtt{U} (q \mathtt{U} r)$	3	**4 (1)**	5 (1)
$\neg(p \mathtt{U} (q \mathtt{U} r))$	3	**4 (1)**	**4 (1)**
$(\Box\Diamond p) \rightarrow (\Box\Diamond q)$	5	12 (2)	**7 (1)**
$(\Diamond p) \mathtt{U} (\Box q)$	6	**4 (1)**	9 (2)
$(\Box p) \mathtt{U} q$	4	6 (2)	**5 (1)**
$\neg(\Diamond\Diamond p \leftrightarrow \Diamond p)$	1	**1 (0)**	2 (1)
$\neg(\Box\Diamond p \rightarrow \Box\Diamond q)$	3	**7 (1)**	12 (2)
$\neg(\Box\Diamond p \leftrightarrow \Box\Diamond q)$	1	**1 (0)**	2 (1)
$p \mathtt{V} (p \vee q)$	2	**3 (1)**	**3 (1)**
$((\mathtt{X} p) \mathtt{U} (\mathtt{X} q)) \vee \neg \mathtt{X}(p \mathtt{U} q)$	1	2 (1)	**1 (0)**
$((\mathtt{X} p) \mathtt{U} q) \vee \neg \mathtt{X}(p \mathtt{U} (p \wedge q))$	4	3 (2)	**1 (0)**
$\Box(p \rightarrow \Diamond q) \wedge (((\mathtt{X} p) \mathtt{U} q) \vee \neg \mathtt{X}(p \mathtt{U} (p \wedge q)))$	6	19 (2)	**3 (1)**
$\Box(p \rightarrow \Diamond q) \wedge (((\mathtt{X} p) \mathtt{U} (\mathtt{X} q)) \vee \neg \mathtt{X}(p \mathtt{U} q))$	2	4 (1)	**3 (1)**
$\Box(p \rightarrow \Diamond q)$	2	4 (1)	**3 (1)**
$\neg\Box(p \rightarrow \mathtt{X}(q \mathtt{V} r))$	3	**3 (1)**	4 (1)
$\neg((\Box\Diamond p) \vee \Diamond\Box q)$	3	**7 (1)**	12 (2)
$\Box(\Diamond p \wedge \Diamond q)$	3	**5 (1)**	15 (2)
$\Diamond p \wedge \Diamond \neg p$	4	**4 (1)**	**4 (2)**
$((\mathtt{X} q) \wedge r) \mathtt{V} \mathtt{X}(((s \mathtt{U} p) \mathtt{V} r) \mathtt{U} (s \mathtt{V} r))$	21	38 (5)	**6 (1)**
$(\Box(q \vee \Box\Diamond p) \wedge \Box(r \vee \Box\Diamond \neg p)) \vee \Box q \vee \Box r$	11	33 (4)	**7 (1)**
$(\Box(q \vee \Diamond\Box p) \wedge \Box(r \vee \Diamond\Box \neg p)) \vee \Box q \vee \Box r$	12	**29 (5)**	53 (3)
$\neg((\Box(q \vee \Box\Diamond p) \wedge \Box(r \vee \Box\Diamond \neg p)) \vee \Box q \vee \Box r)$	27	**17 (1)**	64 (4)
$\neg((\Box(q \vee \Diamond\Box p) \wedge \Box(r \vee \Diamond\Box \neg p)) \vee \Box q \vee \Box r)$	21	73 (3)	**49 (5)**
$\Box(q \vee \mathtt{X}\Box p) \wedge \Box(r \vee \mathtt{X}\Box \neg p)$	3	**5 (1)**	**5 (1)**
$\Box(q \vee (\mathtt{X}(p \wedge \mathtt{X} \neg p)))$	1	3 (1)	**2 (1)**
$(p \mathtt{U} p) \vee (q \mathtt{U} p)$	2	**3 (1)**	4 (1)

Table 5.12: Number of states of DRA and DSA for the formulas from [SB00] and number of states of the NBA for the positive formulas (with `ltl2ba`). In parentheses, the number of acceptance pairs in Ω. The smaller of the two automata is marked in bold.

	Global			Before R			After Q			Between Q and R			After Q until R		
	NBA	DRA	DSA	NBA	DRA	DSA	NBA	DRA	DSA	NBA	DRA	DSA	NBA	DRA	DSA
Abscence	1	**2** (1)	**2** (1)	4	**4** (1)	**4** (1)	2	**3** (1)	**3** (1)	4	7 (2)	**4** (1)	3	6 (2)	**3** (1)
Universality	1	**2** (1)	**2** (1)	4	**4** (1)	**4** (1)	2	**3** (1)	**3** (1)	4	7 (2)	**4** (1)	3	6 (2)	**3** (1)
Existence	2	**2** (1)	3 (1)	3	5 (2)	**3** (1)	5	**3** (1)	4 (1)	3	6 (2)	**3** (1)	2	5 (1)	**4** (2)
Bounded Existence (2)	6	11 (2)	**6** (1)	8	**8** (1)	**8** (1)	9	11 (3)	**7** (1)	16	62 (3)	**8** (1)	12	52 (3)	**7** (1)
Precedence	3	5 (2)	**3** (1)	4	**4** (1)	**4** (1)	6	**5** (2)	8 (1)	4	9 (2)	**4** (1)	3	6 (2)	**3** (1)
Response	2	4 (1)	**3** (1)	5	**8** (1)	**8** (1)	3	6 (1)	**4** (1)	6	22 (3)	**4** (1)	6	32 (3)	**5** (2)
Precedence Chain (1-2)	5	**4** (1)	**4** (1)	6	7 (2)	**5** (1)	7	6 (3)	**5** (1)	8	29 (2)	**5** (1)	12	**32** (2)	389 (4)
Precedence Chain (2-1)	5	**4** (1)	**4** (1)	5	**5** (1)	**5** (1)	7	7 (2)	**5** (1)	10	111 (4)	**5** (1)	10	79 (4)	**4** (1)
Response Chain (2-1)	11	45 (3)	**6** (1)	20	16 (2)	**7** (1)	12	82 (4)	**7** (1)	35	3563 (7)	**9** (1)	30	56050 (11)	**157** (4)
Response Chain (1-2)	5	20 (2)	**14** (3)	10	**5** (1)	**5** (1)	4	24 (2)	**5** (2)	15	**18** (1)	19 (3)	24	11395 (8)	**1976** (11)
Constrained Response (1-2)	5	21 (2)	**15** (3)	10	**5** (1)	**5** (1)	4	25 (2)	**5** (2)	15	**18** (1)	19 (3)	24	31742 (8)	**3952** (11)

Table 5.13: Number of states of DRA and DSA for the pattern formulas and number of states of the NBA for the positive formulas (with `ltl2ba`). In parentheses, the number of acceptance pairs in Ω. The smaller of the two automata is marked in bold.

5.5 Evaluating LTL→NBA translators

ltl2dstar is designed to use an external LTL→NBA translator, which allows experimentation with different approaches to the LTL→NBA problem in the context of subsequent determinization.

We will evaluate four LTL→NBA translators to see which one provides the best basis for the conversion of LTL to deterministic ω-automata.

The area of LTL→NBA translators has been actively researched for several decades, with many different approaches and optimizations. Many implementations are freely available. They often differ in the format of the LTL formulas they expect and in the format for the nondeterministic Büchi automaton they output. Fortunately, there exist wrappers for most tools that provide a simple standard interface, specified by the lbtt testbench[5].

For most LTL→NBA translators, the translation process can be separated into three phases:

1. **LTL formula rewriting:** Try to simplify the formula by using known equivalences.
2. **LTL→NBA:** Convert the formula to a nondeterministic Büchi automaton.
3. **Minimization of the NBA:** Try to simplify the NBA. Common techniques are building the quotient automaton for a simulation relation or finding never accepting SCCs that can be deleted.

The following LTL→NBA translators were selected for a comparison:

- spin[6]: The model checker spin [Hol97] is probably the most widely used LTL model checker, but can also be used as a stand-alone LTL→NBA translator.

  Algorithm: The algorithm spin utilizes was proposed in [GPVW95] and uses a tableaux method to keep track of locally satisfied subformulas and the subformulas that need to be satisfied in the next step.

  Implementation: The NBA are output in the form of *never claims*, program fragments expressed in the language *PROMELA*, which is also used for the specification of the models to check. As *PROMELA* is a complete programming language, never claims can theoretically be very complex. In practice, the never claims generated by spin have a certain structure that can be parsed without implementing a complete *PROMELA* parser. spin is written in C.

- ltl2ba[7]:
  Algorithm: [GO01] ltl2ba was the first LTL→NBA translator that translates

[5] http://www.tcs.hut.fi/Software/lbtt/doc/html/Interfacing-with-lbtt.html
[6] http://www.spinroot.com/
[7] http://www.liafa.jussieu.fr/~oddoux/ltl2ba/

the LTL formulas into *very weak alternating Büchi automata* (e.g. [Var96]). The transitions of alternating automata can express nondeterminism (choose one successor) and universality (choose all successors) at the same time, which allows the construction of an alternating automaton with $\mathcal{O}(|\psi|)$ states for an LTL formula ψ. These automata have a special property (they are *very weak*), allowing to convert them to NBA with $\mathcal{O}(2^{|\psi|})$ states. `ltl2ba` uses simple, but effective reductions at every intermediate step.

Implementation: `ltl2ba` is meant as a replacement for the LTL→NBA algorithm for `spin`, so it outputs never claims in *PROMELA*, too. It is written in C.

- `Modella`[8]:
  Algorithm: [ST03] The algorithm used by `Modella` is a modification of [GPVW95] with the aim of generating more deterministic automata (i.e. avoiding nondeterministic branching where possible, even if this leads to more states).
  It can perform formula rewriting and simulation-based minimization of the NBA with *delayed simulation* [EWS01].

  Interface: `Modella` uses the `lbtt` interface, but requires the LTL formulas to be in PNF. It is written in C++.

- `LTL→NBA`[9]:
  Algorithm: [Fri03] `LTL→NBA` converts the LTL formula to a weak alternating Büchi automaton and calculates a *delayed simulation relation* [EWS01] on the alternating automaton (instead of on the NBA as other tools do) and converts it to a nondeterministic Büchi automaton.

  Implementation: `LTL→NBA` is written in Python and provides an `lbtt` compatible interface.

5.5.1 Benchmarking the LTL→NBA translators

To compare the LTL→NBA translators, the formula sets from Section 5.3 were used to generate nondeterministic Büchi automata with the four different programs. These NBA were then converted to DRA with Safra's construction, using the on-the-fly optimizations from 4.1 and the **Union** construction, but not bisimulation.

Failures to generate NBA or DRA To limit the running time and memory consumption, the generation of the NBA or DRA was aborted if it took longer than 15 minutes or used more than 300 MB of memory. Unfortunately, for some formulas and LTL→NBA translators, these limits were reached, so that the DRA or even the NBA could not be calculated.

[8] `http://www.science.unitn.it/~stonetta/modella.html`
[9] `http://www.ti.informatik.uni-kiel.de/~fritz/`

Using spin, the generation of the NBA was aborted in all 9 failing runs. The three failing runs using Modella were aborted during the construction of the DRA, with DRA sizes exceeding 200,000 states. The NBA generated for these formulas by Modella had 44 and 53 states.

Statistical analysis

To facilitate comparison, for each formula the smallest DRA generated using one of the four different LTL→NBA translators was selected. This allows a comparison of each LTL→NBA translator with this minimum DRA. The smaller the difference between the DRA size and the minimum DRA, the better.

Additionally, the following values were calculated and the results are shown in Table 5.14 to Table 5.17:

Failures	Number of failures to compute the NBA or DRA.
$\Sigma(t)$	Total running time for the LTL→NBA translators and ltl2dstar.
Min. DRA	How often was the generated DRA the smallest of the four programs?
$\Sigma(\lvert\mathcal{A}\rvert)$	The total number of states of the DRA.
Δ	For every DRA, Δ is the difference between the number of states and the number of states of the minimum DRA for the formula.
$\Sigma(\Delta)$	The sum of the differences to the minimum DRA.
$avg(\Delta)$	The average difference to the minimum DRA.

As the size of the DRA for the aborted runs are not known, the presented number for the LTL→NBA translators with failed runs are a lower bound.

Graphs

Figure 5.5 shows the results for the formulas from [SB00] and [EH00], Figure 5.6 for the pattern formulas and Figures 5.7 and 5.8 for the 100 and 1000 random formulas.

The graphs on the left show the size of the DRA and the size of the minimum DRA (of the four) for each formula. The formulas were sorted by the size of the minimum DRA. ”Peaks” indicate bigger DRA size than the minimum DRA, a gap in the graph occurs when generating the DRA failed.

The graphs on the right plotthe size of the NBA against the size of the resulting DRA. Due to the logarithmic scale of the DRA sizes, the exponential relationship between NBA and DRA size appears linear in the graphs.

Both graphs use a logarithmic scale for the number of states in the DRA, so that small and large automata can be shown in the same graph.

Program	Failures	$\Sigma(t)$	Min. DRA	$\Sigma(\|\mathcal{A}\|)$	Δ with min. DRA $\Sigma(\Delta)$	$avg(\Delta)$
`ltl2ba`	0	0.45 s	66%	926	535	13.72
`Modella`	0	46.70 s	41%	5952	5561	142.59
`Spin`	(2.6%) 1	340.00 s	36%	$>$ 3241	$>$ 2858	$>$ 75.21
`LTL→NBA`	0	703.12 s	64%	1315	924	23.69

Table 5.14: Statistical analysis for the 39 formulas from [SB00] and [EH00].

Program	Failures	$\Sigma(t)$	Min. DRA	$\Sigma(\|\mathcal{A}\|)$	Δ with min. DRA $\Sigma(\Delta)$	$avg(\Delta)$
`ltl2ba`	0	269.49 s	56%	246455	88484	1608.80
`Modella`	(5.5%) 3	816.17 s	38%	$>$ 20525	$>$ 19359	$>$ 372.29
`Spin`	(12.7%) 7	5453.19 s	31%	$>$ 45935	$>$ 45068	$>$ 938.92
`LTL→NBA`	0	300.30 s	58%	288745	130774	2377.71

Table 5.15: Statistical analysis for the 55 pattern formulas.

Program	Failures	$\Sigma(t)$	Min. DRA	$\Sigma(\|\mathcal{A}\|)$	Δ with min. DRA $\Sigma(\Delta)$	$avg(\Delta)$
`ltl2ba`	0	0.58 s	79%	642	129	1.29
`Modella`	0	1.56 s	80%	771	258	2.58
`Spin`	0	3.59 s	67%	830	317	3.17
`LTL→NBA`	0	8.00 s	75%	559	46	0.46

Table 5.16: Statistical analysis for the 100 random formulas.

Program	Failures	$\Sigma(t)$	Min. DRA	$\Sigma(\|\mathcal{A}\|)$	Δ with min. DRA $\Sigma(\Delta)$	$avg(\Delta)$
`ltl2ba`	0	5.86 s	80%	6743	1622	1.62
`Modella`	0	26.23 s	74%	69858	64737	64.74
`Spin`	(0.1%) 1	1043.76 s	63%	$>$ 11726	$>$ 6625	$>$ 6.63
`LTL→NBA`	0	81.03 s	77%	6167	1046	1.05

Table 5.17: Statistical analysis for the 1000 random formulas.

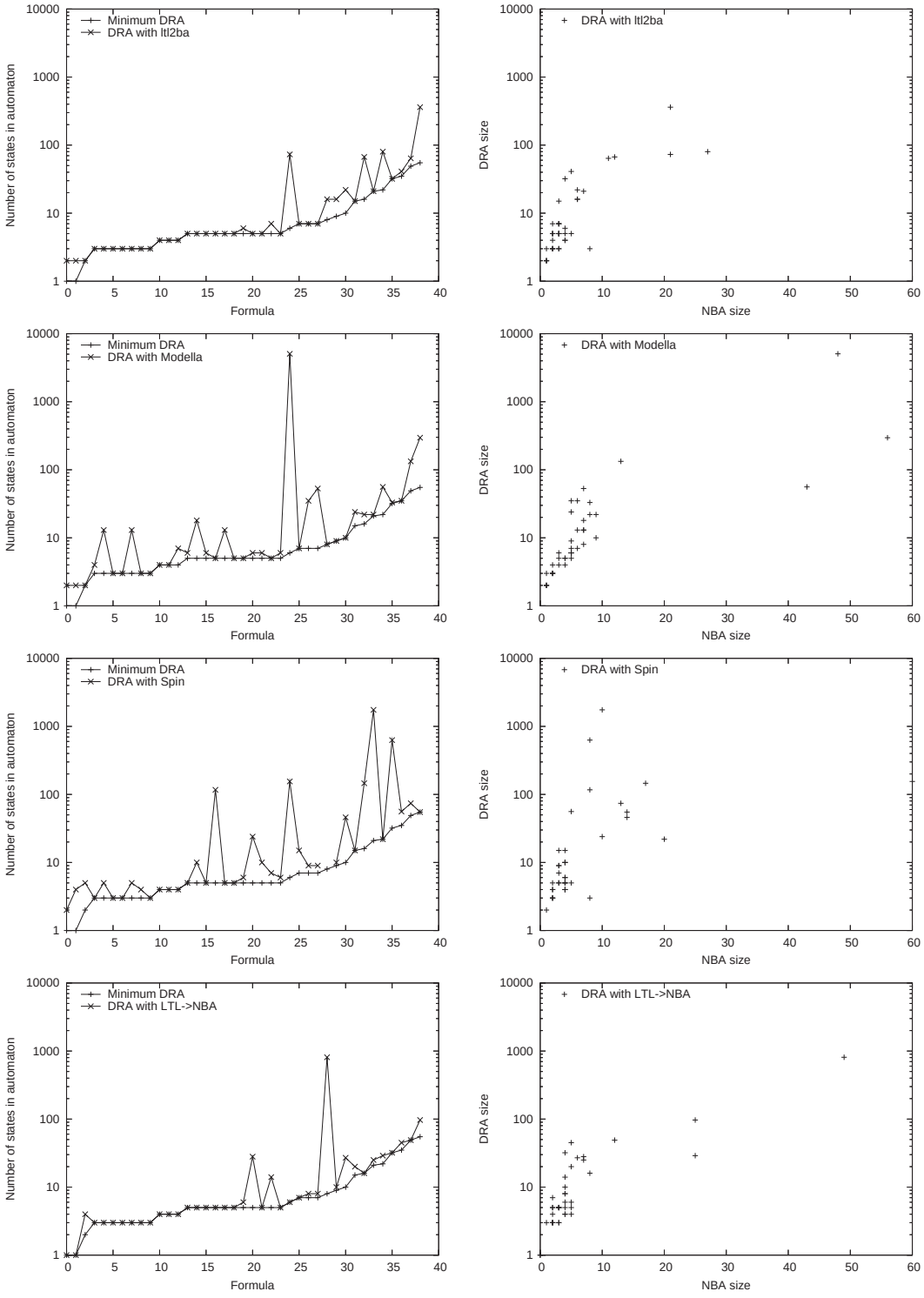


Figure 5.5: Comparison of LTL→NBA tools for the **39 formulas from [SB00] and [EH00]**, sorted by size of the smallest DRA. Right: Comparison between the size of the NBA and the size of the DRA.

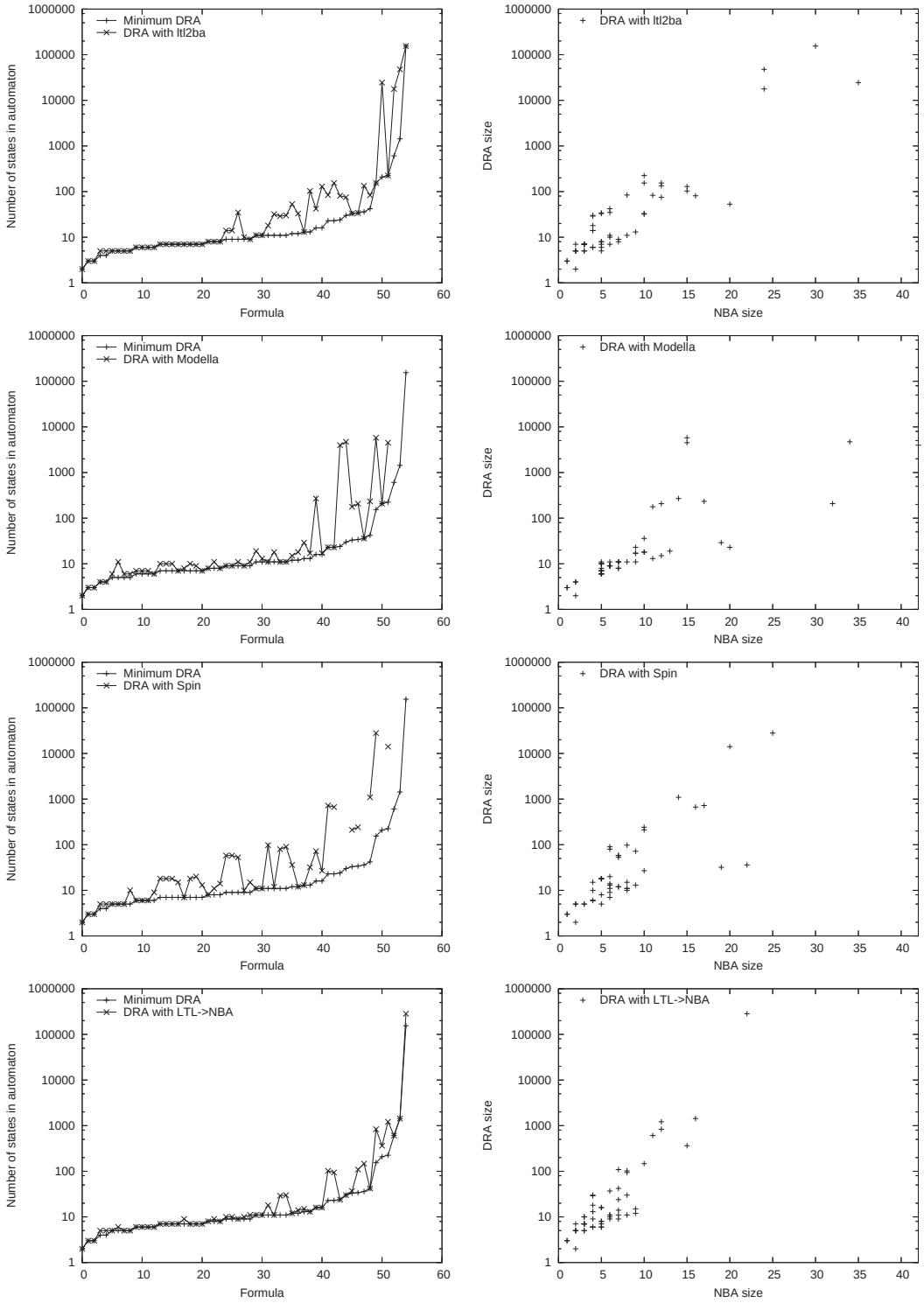


Figure 5.6: Comparison of LTL→NBA tools for the **55 pattern formulas**, sorted by size of the smallest DRA. Right: Comparison between the size of the NBA and the size of the DRA.

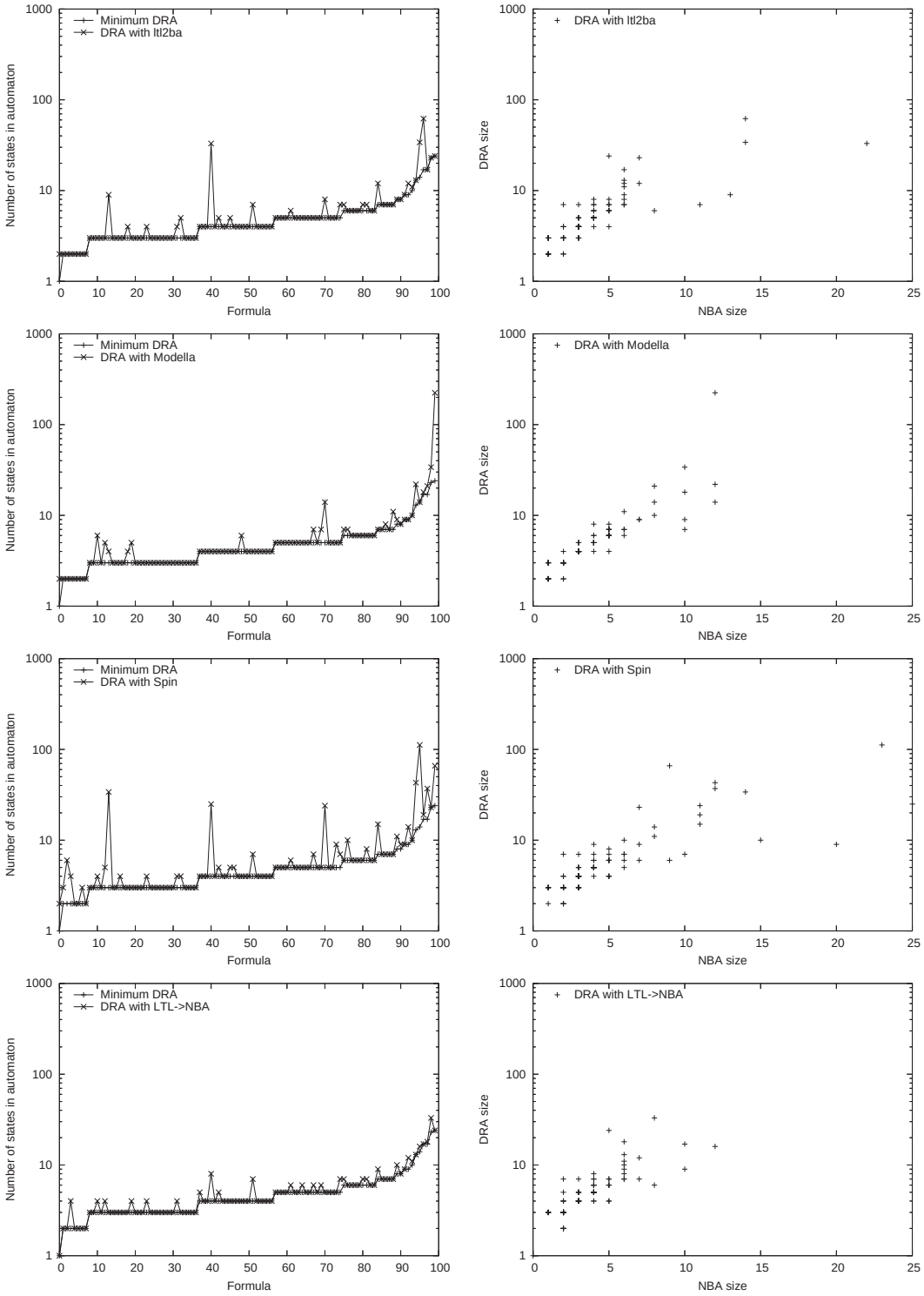


Figure 5.7: Comparison of LTL→NBA tools for **100 random formulas**, sorted by size of the smallest DRA. Right: Comparison between the size of the NBA and the size of the DRA.

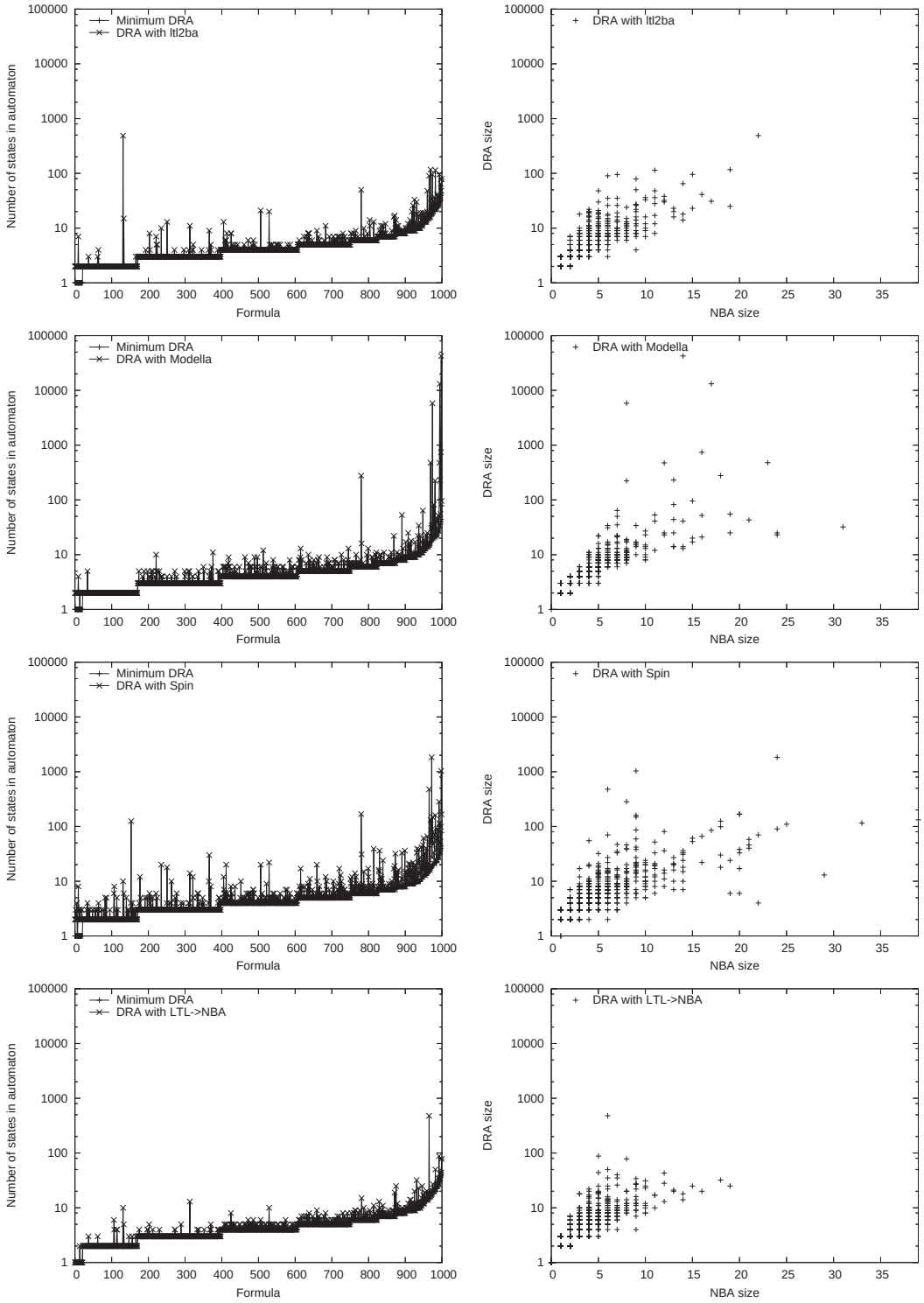


Figure 5.8: Comparison of LTL→NBA tools for **1000 random formulas**, sorted by size of the smallest DRA. Right: Comparison between the size of the NBA and the size of the DRA.

Interpreting the results

For the automata sizes, the two programs using alternating Büchi automata internally (ltl2ba and LTL→NBA) have the lead and were the only programs without failures to calculate the DRA. Both programs had formulas were one was clearly better than the other, so it is not possible to declare a clear winner. If we consider the running time, ltl2ba is clearly the fastest. How much of the longer running time of LTL→NBA is due to the implementation in Python and how much of it is due to the more complex algorithm should be further investigated, as well as the reason why for some formulas the NBA were larger than the NBA from ltl2ba.

Modella performed better than spin. How much of the advantage of Modella is due to the ”more deterministic” NBA it aims to create and how much is due to other optimizations is unclear and should be further investigated.

It will be interesting to see how other, new approaches and tools will behave in the context of ltl2dstar.

6 Conclusion

We have considered Safra's construction in the context of translating LTL formulas to deterministic Rabin and Streett automata and evaluated the performance of its implementation in the tool `ltl2dstar` and the effect several proposed optimizations had on the automata size. Additionally, four LTL→NBA were evaluated in the context of subsequent determinization.

These are the main observations:

- For many formulas, Safra's construction (with the presented optimizations) is usable in practice and results in deterministic ω-automata with acceptable sizes. Nevertheless, there are clearly formulas where determinization using Safra's construction results in very big automata, even when the NBA are relatively small (under 50 states), as is to be expected due to the complexity of determinization.
- The proposed optimizations have a big impact (overall reductions of 70% and more) in practice and contribute a great deal to the practical feasibility of using Safra's construction for LTL formulas.
- If the user has the flexibility to handle both deterministic Rabin and Streett automata, this should be utilized, as it can provide significantly smaller automata and the chance that a particular formula results in an automaton with acceptable size is much higher.
- The duality of Rabin and Streett automata provides a very useful tool for easy complementation of the automata and was used multiple times to good effect. A similar easy complementation is not available for nondeterministic Büchi automata.
- Perhaps surprisingly, the simple direct bisimulation optimization performed extremely well in practice on the DRA and DSA, despite the more complex acceptance condition which lead to a bigger initial partition. This can indicate that there exist redundancies in Safra's construction. It would be interesting to research more complex simulation relations on DRA and DSA.
- The choice of the external LTL→NBA translator has a big impact on the size of the deterministic automata.

6.1 Further research opportunities

This section lists some further optimizations and approaches that might be worthwhile to explore and implement in `ltl2dstar`.

6.1.1 Alternatives to Safra's construction

There exist different approaches to the determinization of Büchi automata and to the translation of LTL formulas to deterministic ω-automata:

1. Muller and Schupp [MS95] propose a different algorithm for the conversion from nondeterministic Büchi automata to deterministic Rabin automata based on tree automata. Like Safra's construction, the algorithm by Muller and Schupp is optimal. It would be very interesting to compare its performance to that of Safra's construction in practice.
2. Schneider [Sch97] proposes an algorithm for generating deterministic automata directly from LTL formulas by using a bottom-up approach using the closure under $\vee$, $\wedge$, `X` and `V`. The generated automata use additional input variables to simulate nondeterminism.
3. [Red99] proposes a translation from ω-regular expressions directly to deterministic ω-automata.

6.1.2 Detecting Büchi-type automata

In [KPB94], Krishnan, Puri and Brayton propose a polynomial time algorithm to check if the language of a deterministic Rabin automaton is also realizable by a deterministic Büchi automaton (*DBA-realizable*). They show that if a DRA is DBA-realizable, the corresponding DBA can have the same transition structure, the two automata differ only in their acceptance condition, i.e. it just has to be determined which states in the DRA become accepting states in the DBA, which can also be done in polynomial time.

6.1.3 Minimization of deterministic weak Büchi automata

In [Löd01], Löding presents a $\mathcal{O}(n \cdot \log n)$ algorithm for constructing the minimum automaton for the subclass of deterministic *weak* Büchi automata.

6.1.4 Converting to minimal-pair DRA

Definition 6.1.1 (Rabin index). For an ω-regular language $\mathcal{L}$, the *Rabin index* of $\mathcal{L}$ is the minimum number k of Rabin acceptance pairs needed to realize $\mathcal{L}$ as a deterministic Rabin automaton.

Wagner [Wag79] showed that the Rabin index induces a strict hierarchy of ω-regular languages, as for every n, there exists a language that can not be realized with fewer than n Rabin pairs.

In [KPB95], Krishnan, Puri and Brayton propose an algorithm that can transform a DRA $\mathcal{A}$ with n states and h acceptance pairs into an equivalent DRA $\mathcal{A}'$ with $\mathcal{O}(n \cdot h^k)$ states and k acceptance pairs as the acceptance condition, where k is the Rabin index of $\mathcal{L}(\mathcal{A})$.

This transformation might be useful in situations were the number of acceptance pairs is a bigger concern than the number of states of a DRA.

Bibliography

[CKS81] Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.

[CY95] Costas Courcoubetis and Mihalis Yannakakis. The complexity of probabilistic verification. *Journal of the ACM*, 42(4):857–907, 1995.

[dA97] Luca de Alfaro. *Formal Verification of Probabilistic Systems*. PhD thesis, Stanford University, Department of Computer Science, 1997.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999.

[EH00] Kousha Etessami and Gerard J. Holzmann. Optimizing Büchi automata. In *CONCUR*, volume 1877 of *Lecture Notes in Computer Science*, pages 153–167. Springer, 2000.

[EWS01] Kousha Etessami, Thomas Wilke, and Rebecca A. Schuller. Fair simulation relations, parity games, and state space reduction for Büchi automata. In *ICALP'2001*, volume 2076 of *Lecture Notes in Computer Science*, pages 694–707. Springer, 2001.

[FKV04] Ehud Friedgut, Orna Kupferman, and Moshe Y. Vardi. Büchi complementation made tighter. In *ATVA*, volume 3299 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2004.

[FOS03] Rudolf Freund, Marion Oswald, and Ludwig Staiger. omega-p automata with communication rules. In *Workshop on Membrane Computing*, volume 2933 of *Lecture Notes in Computer Science*, pages 203–217. Springer, 2003.

[Fri03] Carten Fritz. Constructing Büchi automata from linear temporal logic using simulation relations for alternating Büchi automata. In *Implementation and Application of Automata. Eighth International Conference (CIAA 2003)*, volume 2759 of *Lecture Notes in Computer Science*, pages 35–48. Springer, 2003.

[GO01] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In *Computer Aided Verification (CAV'2001), Proceedings*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2001.

[GPVW95] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *PSTV'95, Proc.*, volume 38 of *IFIP Conference Proceedings*, pages 3–18. Chapman & Hall, 1995.

[GTW02] Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. *Automata Logics, and Infinite Games: A Guide to Current Research*, volume 2500 of *LNCS*. Springer, 2002.

[Hol97] Gerard. J. Holzmann. The Model Checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[KPB94] Sriram C. Krishnan, Anuj Puri, and Robert K. Brayton. Deterministic ω Automata vis-a-vis Deterministic Buchi Automata. In *Algorithms and Computation, 5th International Symposium (ISAAC'94)*, volume 834 of *Lecture Notes in Computer Science*, pages 378–386. Springer, 1994.

[KPB95] Sriram C. Krishnan, Anuj Puri, and Robert K. Brayton. Structural complexity of ω-automata. In *Symposium on theoretical aspects of computer science (STACS'95)*, volume 900 of *Lecture Notes in Computer Science*, pages 143–156. Springer, 1995.

[KV98] Orna Kupferman and Moshe Y. Vardi. Freedom, weakness, and determinism: From linear-time to branching-time. In *Proc. 13th IEEE Symposium on Logic in Computer Science*, pages 81–92, 1998.

[KV99] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. In *Computer Aided Verification (CAV'99), Proceedings*, volume 1633 of *Lecture Notes in Computer Science*. Springer, 1999.

[Lat02] Timo Latvala. On model checking safety properties. Research Report A76, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, December 2002.

[Löd98] Christof Löding. Methods for the transformation of omega-automata: Complexity and connection to second order logic. Diploma thesis, Christian-Albrechts-University of Kiel, Germany, 1998.

[Löd01] Christof Löding. Efficient minimization of deterministic weak omega-automata. *Information Processing Letters*, 79(3):105–109, 2001.

[MP90] Zohar Manna and Amir Pnueli. A hierarchy of temporal properties. In *PODC*, pages 377–410, 1990.

[MS95] David E. Muller and Paul E. Schupp. Simulating alternating tree automata by nondeterministic automata: New results and new proofs of the theorems of Rabin, McNaughton and Safra. *Theoretical Computer Science*, 141(1–2):69–107, 17 April 1995.

[Red99] Roman R. Redziejowski. Construction of a deterministic ω-automaton using derivatives. *ITA*, 33(2):133–158, 1999.

[Saf89] Shmuel Safra. *Complexity of Automata on Infinite Objects*. PhD thesis, The Weizmann Institue of Science, Rehovot, Israel, 1989.

[SB00] Fabio Somenzi and Roderick Bloem. Efficient Büchi automata from LTL formulae. In *Computer Aided Verification (CAV'2000), Proc.*, volume 1855 of *Lecture Notes in Computer Science*, pages 248–263. Springer, 2000.

[Sch97] Klaus Schneider. Translating linear temporal logic to deterministic ω-automata. In *GI/ITG/GMM Workshop: Methoden des Entwurfs und der Verifikation digitaler Systeme*, pages 149–158, 1997.

[Sis94] A. Prasad Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6(5):495–512, 1994.

[ST03] Roberto Sebastiani and Stefano Tonetta. "More Deterministic" vs. "Smaller" Büchi Automata for Efficient LTL Model Checking. In *CHARME 2003, Proc.*, volume 2860 of *Lecture Notes in Computer Science*, pages 126–140. Springer, 2003.

[Tau00] Heikki Tauriainen. Automated testing of Büchi automata translators for linear temporal logic. Research report, Helsinki University of Technology, Laboratory for Theoretical Computer Science, December 2000.

[Tho90] Wolfgang Thomas. Automata on infinite objects. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages 133–192. Elsevier and MIT Press, 1990.

[Tho97] Wolfgang Thomas. Languages, automata, and logic. *Handbook of formal languages*, 3:389–455, 1997.

[Var94] Moshe Y. Vardi. Nontraditional applications of automata theory. In *TACS*, volume 789 of *Lecture Notes in Computer Science*, pages 575–597. Springer, 1994.

[Var96] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Banff Higher Order Workshop*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer, 1996.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *LICS*, pages 332–344. IEEE Computer Society, 1986.

[Wag79] K. Wagner. On ω-Regular Sets. *Information and Control*, 43:123–127, 1979.

[Wol82] Pierre Wolper. *Synthesis of Communicating Processes from Temporal Logic Specifications*. PhD thesis, Stanford University, 1982.

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Joachim Klein