BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

HYPERLTL MODEL CHECKING

HYPERLTL MODEL CHECKING

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **ONDREJ ALEXAJ**
AUTOR PRÁCE

SUPERVISOR **Ing. ONDŘEJ LENGÁL, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

154514

Institut:	Department of Intelligent Systems (DITS)
Student:	**Alexaj Ondrej**
Programme:	Information Technology
Title:	**HyperLTL Model Checking**
Category:	Formal Verification
Academic year:	2023/24

Assignment:

1. Study the theory of omega automata with a focus on the construction of a complement automaton and testing language inclusion.
2. Study the HyperLTL logic and its model checking.
3. Design an optimized algorithm for HyperLTL model checking.
4. Implement the designed algorithm and compare the performance of the implementation with existing tools.
5. Evaluate the obtained results and discuss possible extensions.

Literature:

- E. Grädel, W. Thomas, and T. Wilke, Eds., *Automata, logics, and infinite games*. in Lecture notes in computer science, no. 2500. Berlin ; New York: Springer, 2002.
- B. Finkbeiner, “Logics and Algorithms for Hyperproperties,” *ACM SIGLOG News*, vol. 10, no. 2, pp. 4–23, July 2023, doi: 10.1145/3610392.3610394.
- V. Havlena, O. Lengál, Y. Li, B. Šmahlíková, and A. Turrini, “Modular Mix-and-Match Complementation of Büchi Automata,” in *Tools and Algorithms for the Construction and Analysis of Systems*, S. Sankaranarayanan and N. Sharygina, Eds., in Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2023, pp. 249–270. doi: 10.1007/978-3-031-30823-9_13.
- R. Beutner and B. Finkbeiner, “AutoHyper: Explicit-State Model Checking for HyperLTL,” in *Tools and Algorithms for the Construction and Analysis of Systems*, S. Sankaranarayanan and N. Sharygina, Eds., in Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2023, pp. 145–163. doi: 10.1007/978-3-031-30823-9_8.

Requirements for the semestral defence:
1-3

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Lengál Ondřej, Ing., Ph.D.**
Consultant:	Havlena Vojtěch, Ing., Ph.D.
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	9.5.2024
Approval date:	6.11.2023

Abstract

HyperLTL model checking is an approach to verifying a system against a given hyperproperty, which is able to relate multiple executions of a system. The algorithmic approach based on automata which relies on standard ω-automata operations is well established. The aim of this work is to outperform the complete state-of-the-art HyperLTL model checker AutoHyper by employing more efficient partial automata operations, in particular complementation and inclusion. The implementation of HyperLTL model checking in a novel modular-based complementation tool Kofola resulted in a significant enhancement in performance compared to the reference tool. Finally, our approach to language inclusion checking shows a notable improvement in terms of the generated state space. As a commonly used automata operation, it could potentially contribute to the advancement of other areas of verification.

Abstrakt

HyperLTL model checking je technika pre overenie systému voči danej hypervlastnosti vyjadrenej logikou HyperLTL, ktorá dokáže prepojiť viaceré spustenia systému. Hoci bol vytvorený algoritmický prístup založený na automatoch, spolieha sa na štandardné operácie ω-automatov. Cieľom tejto práce je prekonať kompletný state-of-the-art HyperLTL model checker AutoHyper využitím efektívnejších čiastkových operácií nad automatmi, najmä komplementácie a inklúzie. Implementácia HyperLTL model checkingu v modulárne založenom nástroji pre komplementáciu, Kofola, viedla k výraznému zvýšeniu výkonu v porovnaní s referenčným nástrojom. Napokon, náš prístup ku kontrole jazykovej inklúzie vykazuje výrazné zmenšenie generovaného stavového priestoru. Keďže ide o bežne používanú operáciu nad automatmi, náš prístup by potenciálne mohol prispieť k pokroku aj v iných oblastiach verifikácie.

Keywords

formal verification, model checking, HyperLTL, TGBA, language inclusion, on-the-fly, language emptiness

Kľúčové slová

formálna verifikácia, model checking, HyperLTL, TGBA, jazyková inklúzia, prázdnosť jazyka

Reference

ALEXAJ, Ondrej. *HyperLTL Model Checking.* Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Ondřej Lengál, Ph.D.

Rozšírený abstrakt

V oblasti hardvérových a softvérových systémov je formálna verifikácia procesom dokazovania alebo vyvrátenia správnosti systému vzhľadom na danú vlastnosť. Dosahuje sa to pomocou formálnych metód, ktoré poskytujú matematický základ pre špecifikáciu vlastností a modelovanie správania systému. HyperLTL model checking (MC) je potom technika, ktorá umožňuje overenie systému voči danej hypervlastnosti vyjadrenej logikou HyperLTL, ktorá sa vzťahuje na viaceré spustenia systému. Hoci bol vytvorený algoritmický prístup založený na automatoch (ABV), spolieha sa na štandardné operácie ω-automatov. Každá formula vyjadrená pomocou HyperLTL obsahuje kvantifikátory nad rôznymi spusteniami systému. Vo všeobecnosti môže ísť o ľubovoľnú sekvenciu existenčných a univerzálnych kvantifikátorov. V ABV sa však výskyt univerzálnych kvantifikátorov prevádza na existenčný pomocou zákona dvojitej negácie. To v skratke znamená, že vždy keď dôjde k alternácii kvantifikátorov (t. j. existenčný na univerzálny alebo naopak), je do konštrukcie vnesená požiadavka na komplementáciu ω-automatov. Ako je dobre známe, ide o drahú operáciu, s asymptoticky exponenciálnou stavovou explóziou. Pri riešení HyperLTL MC existuje špeciálny prípad (vyskytujúci sa pomerne často), kedy môže byť výhodnejšie zisťovať jazykovú inklúziu medzi takýmito automatmi, čo okrem komplementácie využíva aj algoritmus pre rozhodnutie prázdnosti jazyka.

Cieľom tejto práce je prekonať kompletný (teda teoreticky dokáže vyriešiť ľubovoľnú úlohu pre HyperLTL model checking) state-of-the-art nástroj pre HyperLTL model checking, a to využitím efektívnejších čiastkových operácií nad automatmi, hlavne spomínanej komplementácie a inklúzie. S využitím modulárneho nástroja na komplementáciu Büchiho automatov (intuitívne, s podmienkou nekonečne veľakrát dosiahnuť akceptačný stav) Kofola [26] sme referenčný nástroj AutoHyper [4] dokázali v rýchlosti riešenia prekonať na väčšine testovacích prípadov. K efektívnej existujúcej komplementačnej procedúre bol v tejto práci pridaný vylepšený algoritmus pre zisťovanie prázdnosti jazyka zovšeobecnených Büchiho automatov (automaty so zložitejšou akceptačnou podmienkou), ktoré dokopy tvoria zlepšený algoritmus pre zisťovanie inklúzie. Konkrétne optimalizuje známy algoritmus pre test prázdnosti od autorov Gaiser a Schwoon [25]. Okrem optimalizácií pre skoré odhalenie neprázdnosti jazyka boli navrhnuté a implementované aj techniky pre orezanie stavového priestoru, ak je jazyk automatu prázdny. Pôvodný algoritmus v takomto prípade generoval stavový priestor celý. Konkrétne ide o identifikovanie tzv. subsumpcií, teda relácii medzi stavmi, ktoré takéto orezávanie (aj v prípade neprázdnosti) stavového priestoru umožňujú na základe ich štruktúry, bez znalosti zatiaľ nepreskúmaných častí automatu. Pričom štruktúra stavov je daná práve komplementačnou procedúrou nástroja Kofola. Dôvodom pre snaženie o minimalizáciu skúmaného stavového priestoru je fakt, že zložitosť inklúzie je priamo závislá na zložitosti komplementácie, teda stavový priestor tiež môže exponenciálne narásť.

Navrhnuté vylepšenia boli spolu s procedúrou pre HyperLTL model checking implementované v spomínanom nástroji Kofola, ktorý je implementovaný v jazyku C++ a postavený nad knižnicou Spot [21]. Okrem testovania samotného model checkingu sme sa zamerali aj na testovanie navrhnutého prístupu k inklúzii. Tu sme boli schopní častokrát orezať stavový priestor aj na polovicu, v extrémnych prípadoch sme dokonca nevygenerovali žiadny stav. Testovanie voči nástroju Spot ukázalo, že v tejto metrike ho často porážame, veľakrát vďaka technikám predstaveným touto prácou. Nakoniec sme otestovali aj rýchlosť tejto procedúry s ostatnými aktuálnymi nástrojmi, kde sme takmer vo všetkých prípadoch konštatovali víťazstvo, a to aj napriek tomu, že tento údaj pre nás nebol primárnym cieľom.

HyperLTL Model Checking

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Ondřej Lengál, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Ondrej Alexaj
May 7, 2024

Acknowledgements

I would like to thank my supervisor Ing. Ondřej Lengál, Ph.D. for his guidance, help and patience during my work on this thesis.

Contents

Chapter 1

Introduction

In the domain of hardware and software systems, formal verification is the process of proving or disproving the correctness of a system with respect to a given property. This is achieved through the use of formal methods which provide a mathematical basis for specifying properties and modeling system behavior. Formal languages, automata theory, and logics are some of the most important formal methods used in verification tasks [2, 22]. Model checking [2] is an automated verification method that systematically checks whether a property holds in the modeled system or not. The main advantage of this approach is the ability to provide a counterexample in case the property does not hold.

Hyperproperties were defined by Clarkson and Schneider in 2008 [10] as a set of trace properties. They point out properties that cannot be formulated as properties of a single execution. In contrast to the properties of a single execution trace, which is satisfied by a trace, a hyperproperty is satisfied by a set of traces. Robustness [16], path planning [28], generalized non-interference [34], etc. are examples of such properties. HyperLTL [9] is then an extension of linear temporal logic (LTL) that serves as a formal base to express a class of linear hyperproperties. An approach often referred to as Automata-Based Verification (ABV) [23], has been established to perform model checking. Automata-Based Verification in HyperLTL model checking relies on utilizing automata over infinite words, ω-automata. Although the ABV approach is decidable [23], it suffers from common ω-automata problems. The automata operations it consists of are the costliest ones. Namely, it includes complementation, automata product, and inclusion checking.

In this thesis, we first settle the common theoretic notions regarding automata theory. We then move on to the definition of HyperLTL and provide an example of verifying a simple system against a simple property, based on the formal definition of the semantics of this logic. Automata-Based Verification is then explained, along with a brief example of how this check can be done algorithmically. Then we briefly describe the complementation tool Kofola [26], which we use for the HyperLTL model checking. Subsequently, we formally describe the approaches used in our implementation of the language inclusion procedure (as an important part of the ABV), mainly based on subsumption relations [8] that imply language inclusion between pairs of states, taking advantage of the specifics implied by the use of Kofola. We can divide the approaches into two main categories, those that help report counterexamples earlier than constructed, and those that prune the state space even if no counterexample exists. Finally, an experimental evaluation comparing the state-of-the-art tools is provided for both HyperLTL model checking and inclusion checking.

Chapter 2

Preliminaries

We need to define the necessary theory and notation for the reader to understand the subsequent chapters. Firstly, we define fundamental terms in automata theory relevant to this thesis. Then we focus on the concepts regarding omega automata. Lastly, we delve into the definition of Linear-Time Temporal Logic (henceforth referred to as LTL) and its extension HyperLTL.

2.1 Automata

An *alphabet* is a finite, nonempty set of symbols denoted by Σ. We call x a (finite) *string* over Σ if and only if $x = x_1 x_2 \dots x_n$ where each $x_i \in \Sigma$ for $1 \leq i \leq n$. The *empty string* is denoted by ε. The set of all finite strings over Σ is represented by Σ^*. A *language* over Σ is defined as a set $L \subseteq \Sigma^*$.

A finite automaton is a five-tuple $A = (Q, \Sigma, \delta, I, F)$ where Q is a finite set of states, Σ is an alphabet, $\delta \colon Q \times \Sigma \to 2^Q$ is a transition function, $I \subseteq Q$ is a set of initial states, and $F \subseteq Q$ is a set of accepting states. We say that the automaton A accepts a string $x_1 x_2 \dots x_n$ over Σ when there exists a sequence $q_0 x_1 q_1 x_2 \dots x_n q_n$ such that $q_0 \in I$, $q_n \in F$ and $q_{i+1} \in \delta(q_i, x_{i+1})$ for all $0 \leq i < n$. The set of all strings accepted by A is called the *language of the automaton* A, denoted by $L(A)$. Consider a state $q_n \in Q$; if there is no sequence $q_0 x_1 q_1 x_2 \dots x_n q_n$ such that $x_{i+1} \in \Sigma, q_0 \in I$, and $q_{i+1} \in \delta(q_i, x_{i+1})$ for all $0 \leq i < n$, we say that q_n is *unreachable*.

A language L is called a *regular language* if there exists a finite automaton A such that $L = L(A)$. Regular languages can also be described by *regular expressions*. In fact, for each regular expression, there exists an equivalent finite automaton and vice versa. Regular expressions are composed according to the following rules [22]:

- ε and $\emptyset$ are regular expressions:
  $L(\varepsilon) = \{\varepsilon\}$ and $L(\emptyset) = \emptyset$,
- a is a regular expression if $a \in \Sigma$:
  $L(a) = \{a\}$,
- if r and s are both regular expressions, $r + s$ is also a regular expression:
  $L(r + s) = L(r) \cup L(s)$,
- if r and s are both regular expressions, $r.s$ is also a regular expression:
  $L(r.s) = \{xy \mid x \in L(r) \wedge y \in L(s)\}$,

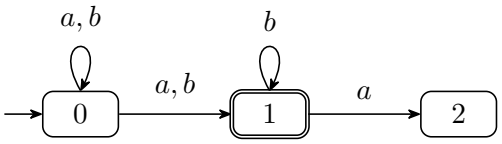


Figure 2.1: A directed graph representing a finite automaton $A = (\{0,1,2\}, \{a,b\}, \{(0,a,\{0,1\}), (0,b,\{0,1\}), (1,a,\{2\}), (1,b,\{1\})\}, \{0\}, \{1\})$ recognizing the language $L(A) = (a+b)^+$. States are (in this case) represented as rectangles with rounded corners, transitions as arrows, the initial state has an incoming arrow, and an accepting state is marked with a double border.

- if r is a regular expression, then r^* is also a regular expression: $L(r^*) = \{x_1 \dots x_n \mid n \in \mathbb{N} \text{ and } x_i \in L(r) \text{ for all } 0 \leq i \leq n\}$,
- if r is a regular expression, then r^+ is also a regular expression: $L(r^+) = \{x_1 \dots x_n \mid n \in \mathbb{N} \text{ and } x_i \in L(r) \text{ for all } 1 \leq i \leq n\}$.

Refer to Figure 2.1 for a graphical representation of a finite automaton and the language it recognizes, expressed by a *regular expression*.

2.2 Büchi automata

Let Σ be an alphabet. Then Σ^ω is the set of all infinite words (ω-words) over Σ. An ω-language is a subset of Σ^ω. Consider an ω-word X over Σ. $X(i)$, for $i \in \mathbb{N}$, represents the i -th symbol of X starting from 0. The substring of X from i to j inclusive will be denoted as $X(i,j) = X(i) \dots X(j)$. Automata operating over ω-words are called *ω-automata*.

A *state-based* Büchi automaton (BA) is an ω-automaton defined as a five-tuple $A = (Q, \Sigma, \delta, I, F)$, where

- Q is a finite set of states,
- Σ is an alphabet,
- $\delta \subseteq Q \times \Sigma \times Q$ is a set of transitions,
- $I \subseteq Q$ denotes the set of initial states, and
- $F \subseteq Q$ is a set of accepting states.

If needed, δ can be treated as a transition function $\delta\colon Q \times \Sigma \to 2^Q$. We say that A is *deterministic* if and only if $|\delta(q,a)| \leq 1$ for all $q \in Q, a \in \Sigma$; if $|\delta(q,a)| \geq 1$ the automaton A is *complete*. Figure 2.2 provides an example of an automaton that is neither *deterministic* nor *complete*.

For convenience, we often use a *transition-based* Büchi automaton, as it may provide a more compact representation. To maintain uniformity in further definitions of different types of ω-automata, we define it as a special type of a non-deterministic *transition-based Emerson-Lei automaton* (TELA) over Σ. It is represented by the tuple $A_{EL} = (Q, \delta, I, \Gamma, p, Acc)$, where Q is a finite set of states, $\delta \subseteq Q \times \Sigma \times Q$ is a set of transitions, $I \subseteq Q$ is the set of initial states, $\Gamma = \{0, \dots, k-1\}$ is a set of $k \in N^+$ colors, $p\colon \delta \to 2^\Gamma$

(a) $L(A) = (a+b)^*.(b)^\omega$ (b) $L(A_{tba}) = (a+b)^*.(b)^\omega$

Figure 2.2: Examples of state-based (a) and transition-based (b) Büchi automata.

is a coloring function of transitions, and Acc is any acceptance condition generated by the following grammar:

$$\alpha ::= Inf(c) \mid Fin(c) \mid (\alpha \wedge \alpha) \mid (\alpha \vee \alpha)$$

such that $c \in \Gamma$. We refer to it as a *transition-based* Büchi automaton when $\Gamma = \{0\}$ and $Acc = Inf(0)$, denoting it as $A_{tba} = (Q, \Sigma, \delta, I, F)$, where $F = p^{-1}(0)$ (the set of all accepting transitions) [26].

2.2.1 Run

Let $x \in \Sigma^\omega$. A run r of A on x is an ω-word over the alphabet of states Q, such that $r(0) = q_{in}$ and $r(i+1) \in \delta(r(i), x(i))$ for all $i \geq 0$. Consider the *state-based* Büchi automaton $A = (Q, \Sigma, \delta, \{q_{in}\}, F)$ and let $S_{\text{inf}}(r)$ be the set of all states that occur infinitely often in the run r. The run r is accepting if and only if $S_{\text{inf}}(r) \cap F \neq \emptyset$ [1]. Similarly, for the *transition-based* case, let $T_{\text{inf}}(r)$ be the set of all transitions that occur infinitely often in the run r. The run r is accepting iff $T_{\text{inf}}(r) \cap F \neq \emptyset$.

2.2.2 Language of Büchi automaton

A Büchi automaton A accepts an infinite word α if there is an accepting run r of A on α. The language recognized by A is defined as follows: $L(A) = \{\alpha \in \Sigma^\omega \mid A \text{ accepts } \alpha\}$. The complementary language is defined as $\overline{L(A)} = \{\alpha \in \Sigma^\omega \mid A \text{ does not accept } \alpha\}$.

Let R be a regular expression with $\varepsilon \notin L(R)$, then R^ω is called the *infinite concatenation*. $L(R^\omega) = L(R)^\omega$, where $L(R)^\omega = \{x_0 x_1 \ldots \mid x_i \in L(R) \text{ for all } i \in \mathbb{N}\}$. A language L is *ω-regular* if $L = \bigcup_{i=1}^{n} R_i S_i^\omega$, where R_i and S_i are regular languages. Such languages are precisely those recognized by Büchi automata. When it comes to *closure properties*, ω-regular languages are closed under union, intersection, and complement. [22, 38]

2.2.3 Strongly connected component

A non-empty set of states $C \subseteq Q$ is a maximal *strongly connected component* (SCC) if for each $p, q \in C$, q is reachable in C from p and p is reachable in C from q, and C is a maximal set with these properties [26]. A strongly connected component is called *trivial* if it consists only of one state with no self-loops. In the further text, SCC always refers to the maximal strongly connected component.

2.3 Complementation of Büchi automata

Complementation of Büchi automata is a crucial task, integral to termination analysis, model checking procedures, etc. For a given Büchi automaton A, complementation is the process of obtaining a Büchi automaton A^c such that $L(A^c) = \overline{L(A)}$.

The complement of a deterministic BA has at most $2n$ states. However, non-determinism enhances the expressiveness of BAs [22]. Consequently, there exist nondeterministic Büchi automata for which no deterministic equivalent exists (e.g., $L = (a+b)^*b^\omega$ can only be recognized by a nondeterministic BA). Non-determinism is a factor that makes complementation difficult.

The worst-case state explosion resulting from the general complementation procedures is $O((0.76n)^n)$ [1]. Several main general complementation approaches are identified:

- determinization-based [36],
- slice-based [29],
- rank-based [31],
- Ramsey-based [7], and
- subset-tuple construction [1].

Additionally, there are algorithms that leverage the structure of the input automaton, applying specialized complementation procedures to enhance the bounds on state explosion. An overview and a brief description of the structure of special automata types is listed below [27, 26].

- Deterministic BA: defined above, complement size is at most $2n$.
- Semi-deterministic BA: consists of an initial part without accepting states/transitions and a deterministic part containing accepting states/transitions. The transitions from initial to deterministic part are one-way only. The size of the complement is $O(4^n)$ as a result of the *NCSB* construction [6].
- Inherently weak BA: within each SCC, all cycles are accepting, or all cycles are rejecting. The size of the complement is $O(3^n)$ as a result of the *Miyano-Hayashi* construction [35].
- Elevator BA: each SCC is deterministic or inherently weak. Complement size is $O(4^n)$ [26].

2.4 Generalized Büchi automata

A transition-based Generalized Büchi automaton (TGBA) can be defined as the transition-based Emerson-Lei automaton $A_{TGBA} = (Q, \delta, I, \Gamma, p, Acc)$ over Σ such that $\Gamma = \{0, \ldots, k-1\}$ and $Acc = Inf(0) \wedge \ldots \wedge Inf(k-1)$ [20]. An example of a TGBA is shown in Figure 2.3.

2.5 Intersection of Büchi automata

Consider (TG)BAs $A = (Q_a, \delta_a, I_a, \Gamma_a, p_a, Acc_a)$ and $B = (Q_b, \delta_b, I_b, \Gamma_b, p_b, Acc_b)$ over the same alphabet Σ, and without loss of generality, we assume $Q_a \cap Q_b = \emptyset$ and $\Gamma_a \cap \Gamma_b = \emptyset$. Then the automaton $P = (Q', \delta', I', \Gamma', p', Acc')$ over Σ that recognizes the intersection of their languages $L(P) = L(A) \cap L(B)$ can be defined as follows:

- $Q' = Q_a \times Q_b$,

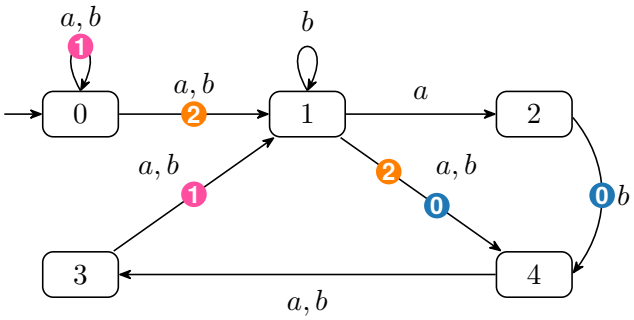


Figure 2.3: Example of TGBA A over $\Sigma = \{a, b\}$ with $Acc = Inf(0) \wedge Inf(1) \wedge Inf(2)$ and $L(A) = \Sigma^\omega$.

- $\delta'((q_{a_1}, q_{b_1}), a) = \delta_a(q_{a_1}, a) \times \delta_b(q_{b_1}, a)$, for $a \in \Sigma$,
- $I' = I_a \times I_b$,
- $\Gamma' = \Gamma_a \cup \Gamma_b$,
- $p'((q_{a_1}, q_{b_1}) \xrightarrow{a} (q_{a_2}, q_{b_2})) = p_a(q_{a_1} \xrightarrow{a} q_{a_2}) \cup p_b(q_{b_1} \xrightarrow{a} q_{b_2})$, and
- $Acc' = Acc_a \wedge Acc_b$.

Refer to Figure 2.4 for a demonstration of the construction of the *product* (states are the product of the original automata states).

Theorem 1. *For TGBA P it holds that $L(P) = L(A) \cap L(B)$.*

2.6 Language emptiness

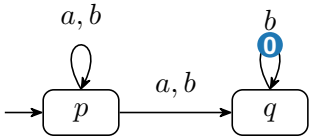
Consider an automaton A. The *language emptiness problem* is the task of deciding whether $L(A) = \emptyset$ holds. For automata over infinite words (ω-automata) it boils down to checking the existence of a lasso that satisfies the respective acceptance condition.

2.7 Language inclusion

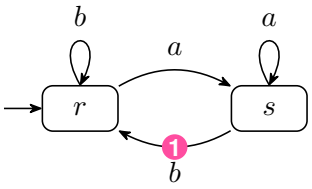
Consider automata A and B. The *language inclusion problem* is the task of deciding whether $L(A) \subseteq L(B)$ holds. The theoretically optimal solution is to decide the following problem $L(A) \cap \overline{L(B)} \stackrel{?}{=} \emptyset$. This entails complementation, subsequent intersection, and finally an emptiness check of the resulting automaton. In practice, we can avoid constructing the entire product automaton. More specifically, to check if the resulting automaton is empty, we can use techniques that significantly restrict the generated state space.

2.8 HyperLTL

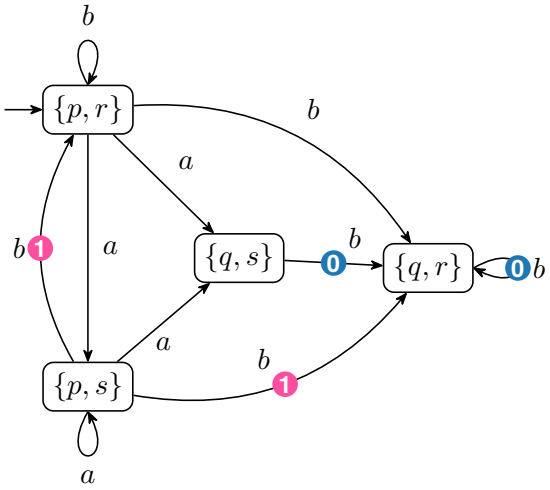
We fix a finite set of *atomic propositions* AP. A trace over AP is a map $t\colon \mathbb{N} \to 2^{AP}$, which can be expressed as a sequence $t(0)t(1)\ldots$ The set of all traces over AP is then denoted as $(2^{AP})^\omega$.



(a) $L(A) = (a + b)^*.(b)^\omega$



(b) $L(B) = b^*.(a^+.b^+)^\omega$



(c) The resulting automaton with $Acc = Inf(0) \wedge Inf(1)$ recognizes the language $L(P) = L(A) \cap L(B) = \emptyset$.

Figure 2.4: Example of *product* construction, in accordance to definition in Section 2.5, used to obtain an automaton that recognizes the intersection of languages.

Linear-time temporal logic (LTL) can be seen as a logic describing dynamic worlds, i.e. it is a modal logic. To do so, temporal operators are defined to express changes in time. LTL formulas are generated by the following grammar [23]:

$$\varphi ::= a \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathrm{X}\varphi \mid \varphi \mathrm{U} \varphi$$

where $a \in AP$ is an atomic proposition, $\neg$ and $\wedge$ are standard Boolean operators (other commonly used Boolean operators can be defined in the usual way) and X, U are temporal operators *next* and *until* respectively.

Suppose a trace t and an LTL formula φ. By notation $t, i \models \varphi$ we express that the trace t at a position $i \in \mathbb{N}$ satisfies the formula φ. See the following definition [23]:

$$\begin{aligned}
&t, i \models a && \text{iff } a \in t(i),\\
&t, i \models \neg\varphi && \text{iff } t, i \not\models \varphi,\\
&t, i \models \varphi_1 \wedge \varphi_2 && \text{iff } t, i \models \varphi_1 \text{ and } t, i \models \varphi_2,\\
&t, i \models X\varphi && \text{iff } t, i + 1 \models \varphi,\\
&t, i \models \varphi_1 U \varphi_2 && \text{iff } \exists k \geq i \colon t, k \models \varphi_2 \text{ and } \forall i \leq j < k \colon t, j \models \varphi_1.
\end{aligned}$$

The trace t satisfies ϕ, denoted as $t \models \phi$, if $t, 0 \models \phi$.

(a) $X\varphi$. (b) $\varphi_1 U \varphi_2$.

Figure 2.5: Visualization of *next* (a) and *until* (b) operators. The states for which the corresponding operator holds are colored.

Intuitively, $X\varphi$ means that φ is satisfied at the next position and $\varphi_1 U \varphi_2$ says that at some point (position) φ_2 is satisfied, but until that moment φ_1 holds. One can also come across other derived temporal operators (syntactic sugar), such as:

- *eventually*: $\mathrm{F}\varphi \overset{def}{\iff} true\mathrm{U}\varphi$,
- *globally*: $\mathrm{G}\varphi \overset{def}{\iff} \neg\mathrm{F}\neg\varphi$,
- *weak until*: $\varphi\mathrm{W}\psi \overset{def}{\iff} (\varphi\mathrm{U}\psi) \vee \mathrm{G}\varphi$, and
- *release*: $\varphi\mathrm{R}\psi \overset{def}{\iff} \neg(\neg\varphi\mathrm{U}\neg\psi)$.

For a better understanding of the *next* and *until* temporal operators, see the visualization in Figure 2.5.

However, LTL can only be used for reasoning about a single path. HyperLTL extends LTL formulas with quantification over traces to explicitly express the relations of executions and their properties. HyperLTL formulas are generated by the following grammar [23, 9]:

$$\phi ::= \exists\pi.\phi \mid \forall\pi.\phi \mid \psi$$

$$\psi ::= a_\pi \mid \neg\psi \mid \psi \wedge \psi \mid \mathrm{X}\psi \mid \psi\mathrm{U}\psi$$

where $a \in AP$, $\pi \in \mathcal{V}$ is a trace variable with $\mathcal{V}$ being the infinite supply of trace variables. The body of the HyperLTL formula ψ is essentially an LTL formula. A HyperLTL formula is considered *closed* if each occurrence of the trace variable is bound by a quantifier [9], refer to Figure 2.6 for an illustration.

2.8.1 Semantics

To define the semantics of HyperLTL we need to introduce a trace assignment $\Pi\colon \mathcal{V} \to (2^{AP})^\omega$ for mapping trace variables to actual traces (of the system). If we want to map some trace variable $\pi \in \mathcal{V}$ to a particular trace t using our mapping Π, we denote updating Π so that $\Pi(\pi) = t$ as $\Pi[\pi \to t]$. For the satisfaction of a *closed* HyperLTL formula ϕ over Π and a set of traces T at a position $i \in \mathbb{N}$, we use the notation $T, \Pi, i \models \phi$, defined as follows [23]:

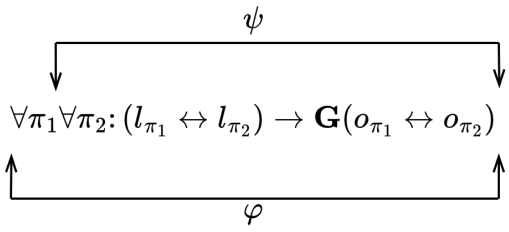


Figure 2.6: Formula φ is closed, while formula ψ is not closed.

$$
\begin{array}{lll}
T, \Pi, i \models a_\pi & \text{iff} & a \in \Pi(\pi)(i) \quad (1)\\
T, \Pi, i \models \neg\psi & \text{iff} & T, \Pi, i \not\models \psi \quad (2)\\
T, \Pi, i \models \psi_1 \wedge \psi_2 & \text{iff} & T, \Pi, i \models \psi_1 \text{ and } T, \Pi, i \models \psi_2 \quad (3)\\
T, \Pi, i \models X\psi & \text{iff} & T, \Pi, i+1 \models \psi \quad (4)\\
T, \Pi, i \models \psi_1 U \psi_2 & \text{iff} & \exists k \geq i\colon T, \Pi, i \models \psi_2 \text{ and } \forall i \leq j < k\colon T, \Pi, j \models \psi_1 \quad (5)\\
T, \Pi, i \models \exists\pi\colon \phi & \text{iff} & \exists t \in T\colon T, \Pi[\pi \rightarrow t], i \models \phi \quad (6)\\
T, \Pi, i \models \forall\pi\colon \phi & \text{iff} & \forall t \in T\colon T, \Pi[\pi \rightarrow t], i \models \phi \quad (7)
\end{array}
$$

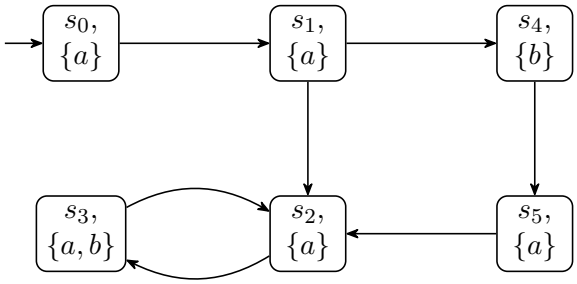
Similarly to LTL, we say that a set of traces T satisfies the property ϕ (written as $T \models \phi$) if $T, \Pi_\emptyset, 0 \models \phi$, where $\Pi_\emptyset$ denotes a mapping with the empty domain.

2.8.2 Kripke structure

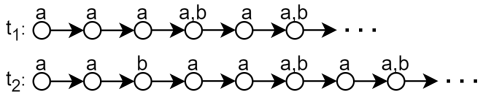
Modeling the traces (behavior) of a specific system can be achieved through the use of a *Kripke structure* (sometimes referred to as a *transition system*) [5, 23]. A *Kripke structure* is a tuple $\mathcal{K} = (S, s_0, \delta, AP, L)$ with the items of the following meaning:

- S is a finite set of states,
- s_0 is the initial state,
- δ is a transition function $\delta\colon S \rightarrow 2^S$,
- AP is the set of atomic propositions, and
- L is a labeling function $L\colon S \rightarrow 2^{AP}$.

When dealing with properties that imply infinite traces, it is necessary for all states $s \in S$ to have $|\,\delta(s)\,| \geq 1$. An infinite sequence $s_0 s_1 \ldots \in S^\omega$ is a *path* of a Kripke structure, with s_0 being the initial state and $s_{i+1} \in \delta(s_i)$ for each $i \in \mathbb{N}$. A trace corresponding to a path $s_0 s_1 \ldots$ is an infinite sequence of labels $l_0 l_1 \ldots$, where each $l_i = L(s_i)$. We use $Tr(\mathcal{K}, s)$ to represent the set of all traces whose corresponding paths start in the state s of a Kripke structure $\mathcal{K}$. Given the set of traces, we can establish the satisfaction of Kripke structure $\mathcal{K}$ with respect to the HyperLTL formula φ as $\mathcal{K} \models \varphi$ if and only if $Tr(\mathcal{K}, s_0) \models \varphi$.



(a) Graphical representation of the Kripke structure $\mathcal{K}$.



(b) The only two traces starting at the initial state.

Figure 2.7: Kripke structure and traces starting at the initial state.

2.8.3 Example

To be more illustrative and to gain some intuition over HyperLTL and its semantics, we provide a simple example. Consider the following HyperLTL formula (expressing a made up hyperproperty):

$$\varphi = \forall \pi_1 \exists \pi_2 \colon a_{\pi_1} \mathrm{U} b_{\pi_2},$$

informally, read as *for each trace* π_1 *there exists a trace* π_2 *such that* a *holds on the trace* π_1 *until* b *holds on the trace* π_2. Also, consider the following Kripke structure $\mathcal{K}$ (depicted in Figure 2.7a):

- $S = \{s_0, s_1, s_2, s_3, s_4, s_5\}$,
- s_0 is the initial state,
- $\delta = \{(s_0, \{s_1\}), (s_1, \{s_2, s_4\}), (s_2, \{s_3\}), (s_3, \{s_2\}), (s_4, \{s_5\}), (s_5, \{s_2\})\}$,
- $AP = \{a, b\}$, and
- $L = \{(s_0, \{a\}), (s_1, \{a\}), (s_2, \{a\}), (s_3, \{a, b\}), (s_4, \{b\}), (s_5, \{a\})\}$.

Because $\mathcal{K}$ is structurally simple, we observe that it contains only two distinct paths starting from the initial state: $s_0 s_1 (s_2 s_3)^\omega$ and $s_0 s_1 s_4 s_5 (s_2 s_3)^\omega$. That means $Tr(\mathcal{K}, s_0)$ contains traces $t_1 = \{a\}\{a\}(\{a\}\{a, b\})^\omega$ and $t_2 = \{a\}\{a\}\{b\}\{a\}(\{a\}\{a, b\})^\omega$ (Figure 2.7b shows traces t_1 and t_2, respectively). In order to decide whether $Tr(\mathcal{K}, s_0) \models \varphi$ holds, we need to inspect every possible assignment to the path variable π_1 from our formula.
Let π_1 **be** t_1. Clearly, when we assign π_2 trace t_2, formula $a_{\pi_1} \mathrm{U} b_{\pi_2}$ holds. Starting from position 0, a holds throughout the entire trace t_1, therefore it is true that a holds until position 2, where b holds on the trace t_2.
Let π_1 **be** t_2. By assigning π_2 the trace t_2 again, the formula $a_{\pi_1} \mathrm{U} b_{\pi_2}$ becomes true. Starting from position 0, a holds until position 2, where b holds on the trace t_2.

The previous was an intuitive approach, so let us now apply systematically the rules from Section 2.8.1. Considering $T = Tr(\mathcal{K}, s_0)$, we can rephrase our problem to check if $T \models \varphi$ holds, that is, if $T, \Pi_\emptyset, 0 \models \varphi$. After applying Rule 7 with $t = t_1$, we proceed to rule 6 with $t = t_2$. This results in $\Pi = \{(\pi_1, t_1), (\pi_2, t_2)\}$. Next, we examine the quantifier-free formula $a_{\pi_1} \mathrm{U} b_{\pi_2}$ using Rule 5, selecting $k = 2$ ($i = 0$ from the problem definition). Now we need to verify $\forall i \leq j < k \colon T, \Pi, j \models a_{\pi_1}$ using Rule 1. For all $j \in \{0, 1\}$, we have $a \in \Pi(\pi_1)(j)$, implying that $T, \Pi, j \models a_{\pi_j}$ holds. Similarly, for $k = 2$ and Rule 1 we

confirm that $b \in \Pi(\pi_2)(k)$, and consequently $T, \Pi, j \models b_{\pi_k}$ holds. To complete the proof that $T \models \varphi$, Rule 7 would be applied with $t = t_2$ following a similar procedure.

Chapter 3

HyperLTL model checking

Model checking serves as an automated verification method. Various properties of system behavior, such as mutual exclusion and accessibility, require verification. To compare a system and its specification, automata over infinite words are often employed. For properties related to a single execution of the system, the established approach is the language inclusion check. To be more precise, let the system S be represented by the ω-automaton A_S and the specification φ by the ω-automaton A_φ (typically obtained through the LTL-to-NBA conversion [14]). Finally, the inclusion check $L(A_S) \subseteq L(A_\varphi)$ is performed.

However, when dealing with hyperproperties (expressing relations between multiple system executions), the task becomes more challenging. When comparing hyperproperties to properties expressed by LTL and aiming to represent them through ω-automata, we must also consider the presence of quantifiers and trace variables in the HyperLTL formulas. Quantifiers, more specifically each quantifier alternation, then cause the need for (possibly more) complementations of ω-automata, making the whole procedure difficult.

Recall the example in Section 2.8.3, where we actually performed model checking in a brute-force manner. Our goal is to perform such checks algorithmically. This is feasible because, similar to single-execution properties, it ultimately involves a language emptiness check, which is decidable.

In this chapter, we first introduce Automata-based Verification (ABV) [24, 4, 23], as one of the approaches for algorithmic verification of hyperproperties, followed by an illustrative explanatory example. Subsequently, we briefly describe AutoHyper [4], a tool that implements a slightly modified version of this algorithm. It will be used as a reference when comparing our implementation against the state-of-the-art push button HyperLTL model checker.

3.1 Automata-based algorithm

Consider a Kripke structure $\mathcal{K} = (S, s_0, \delta_\mathcal{K}, AP, L)$ and a *closed* HyperLTL formula φ. The task is to check whether $\mathcal{K} \models \varphi$. The idea behind the automata-based model-checking is to construct a Büchi automaton that is equivalent to the LTL part (body) of the formula φ. Then we iteratively eliminate trace quantifiers, starting with the innermost one. By doing so, an automaton is acquired that combines the system $\mathcal{K}$ and formula φ. To conclude $\mathcal{K} \models \varphi$ or $\mathcal{K} \not\models \varphi$, we need to decide whether the language of the resulting automaton is empty or not.

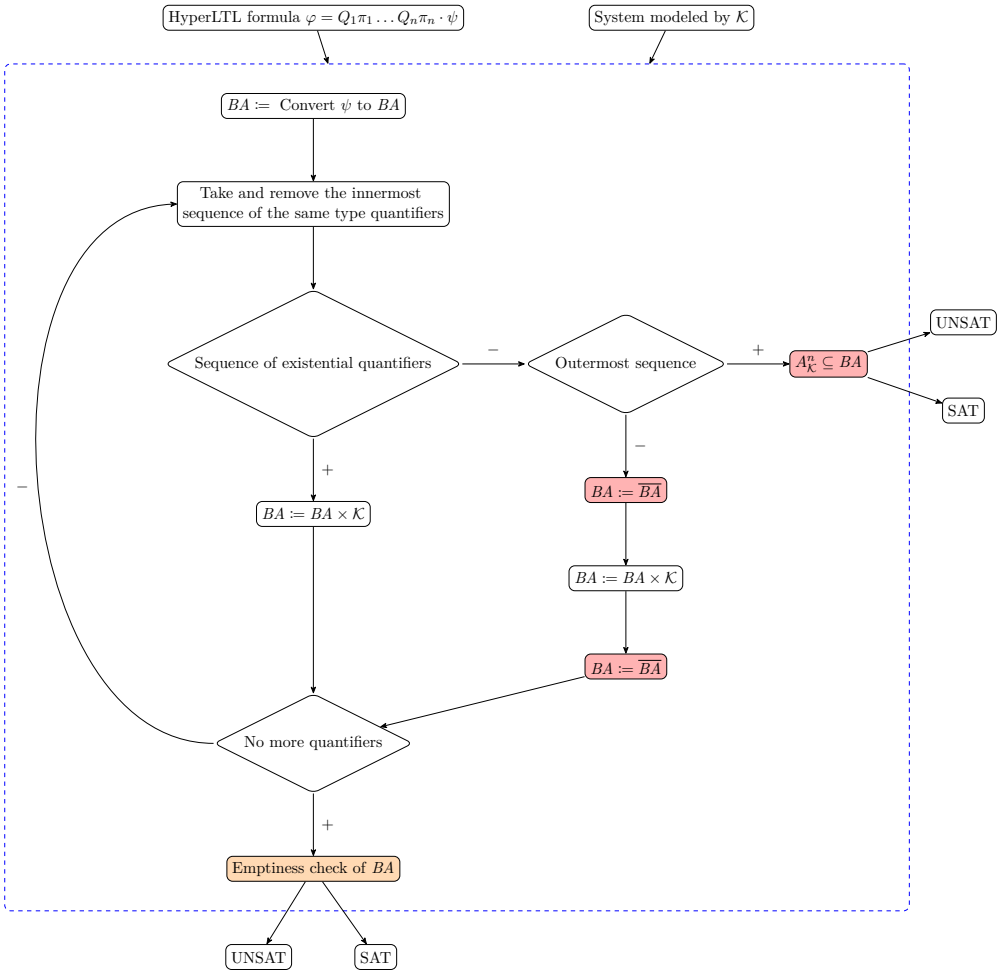
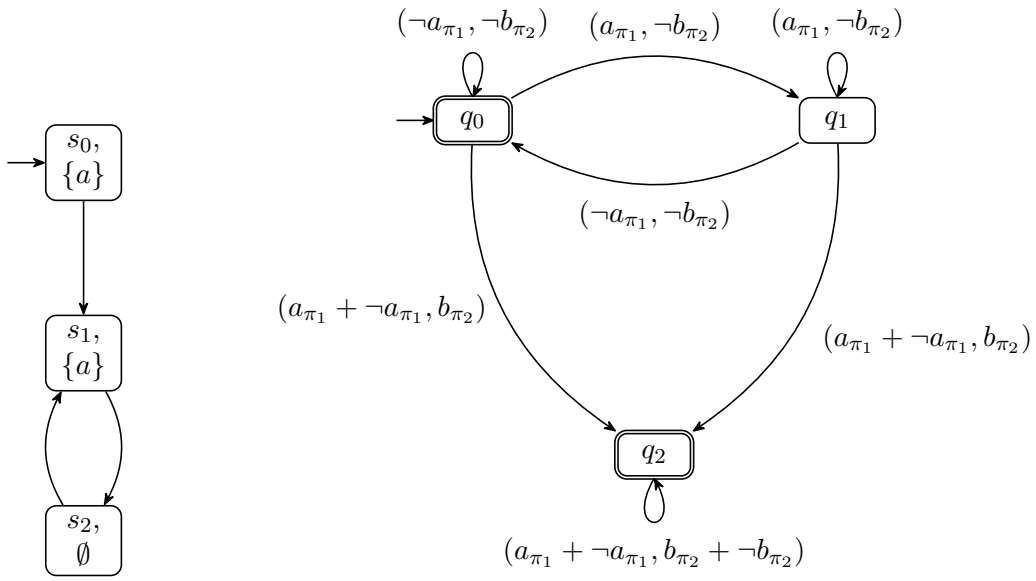


Figure 3.1: The scheme summarizes the steps of the ABV and automata operations it employs. Colors indicate the focus on optimizing specific operations, with red frames indicating a greater focus and orange frames indicating a lower focus.

Here, the algorithm is explained in greater detail. Let $\varphi = Q_1\pi_1 Q_2\pi_2 \dots Q_n\pi_n \colon \varphi^*$, with φ^* denoting the quantifier-free subformula of φ and $Q_i \in \{\exists, \forall\}$ for all $1 \leq i \leq n$. Firstly, a (non-deterministic) Büchi automaton A_{φ^*} equivalent to the LTL body φ^* is constructed. This is accomplished by the standard Tableau construction that creates an automaton accepting exactly ω-words satisfying φ^* [14]. This automaton's alphabet is $\Sigma_{\varphi^*} = (2^{AP})^n$, one set of atomic propositions for each trace quantifier. The next step is to inductively eliminate the trace quantifiers. Suppose the following subformula of φ, $\psi = Q_k\pi_k \colon \varphi_k$. We can safely make an assumption that automaton $A_k = (Q, \Sigma, \delta, q_{in}, F)$ for φ_k is already constructed (automaton A_{φ^*} being the base case). Since Q_k is the k-th quantifier, the alphabet of the automaton A_k is $\Sigma = (2^{AP})^k$. Now, if $Q_k = \exists$, we can perform *existential projection*, which is intuitively the product of A_k and the Kripke structure $\mathcal{K} = (S, s_0, \delta, AP, L)$, necessary to associate the specification with the behavior. Formally, we construct an automaton $A_{k-1} = (Q \times S, \Sigma', \delta', (q_{in}, s_0), F \times S)$ where $\Sigma' = (2^{AP})^{k-1}$ and:

$$\delta'((q,s),(l_1,\dots,l_{k-1})) = \{(r,s') \mid (s,s') \in \delta_{\mathcal{K}} \text{ and } r \in \delta(q,(l_1,\dots,l_{k-1},L(s)))\} \tag{3.1}$$



(a) Kripke structure $\mathcal{K}$.

(b) NBA A for $\mathrm{G}(a_{\pi_1} \rightarrow \mathrm{F}\neg a_{\pi_1}) \vee \mathrm{F}b_{\pi_2}$, adjusted from [30].

Figure 3.2: A simple Kripke structure to describe system behavior and an NBA representing the LTL body of the HyperLTL formula $\mathrm{G}(a_{\pi_1} \rightarrow \mathrm{F}\neg a_{\pi_1}) \vee \mathrm{F}b_{\pi_2}$. To avoid cumbersome notation, sets are given as sums. For example, transition labeled with $(a_{\pi_1} + \neg a_{\pi_1}, b_{\pi_2})$ denotes $(a_{\pi_1} \vee \neg a_{\pi_1}) \wedge b_{\pi_2}$.

where $(l_1, \ldots, l_{k-1})$ and $(l_1, \ldots, l_{k-1}, L(s))$ are letters of automata A_{k-1} and A_k respectively (making it $\Sigma' = (2^{AP})^{k-1}$ for A_{k-1}). An intuitive explanation of this definition is that we read along both the automaton and Kripke structure, choosing only transitions that are acceptable with respect to the current state of the system (Kripke structure). However, we omitted the case where $Q_k = \forall$. This is transformed to the previous scenario using the law of double negation, i.e. $\neg\neg\forall\pi_k\varphi_k = \forall\pi_k\varphi_k$, which we can rewrite as $\neg\neg\forall\pi_k\varphi_k = \neg\exists\pi_k\neg\varphi_k$. Here, the negations raise the need for the complementation procedure of Büchi automata.

After each quantifier is eliminated as described above, we end up with an automaton over the single-letter alphabet $\Sigma = (2^{AP})^0 = \{()\}$. Now we just have to perform an emptiness check on this automaton, which means that $\mathcal{K} \models \varphi$ if and only if the language of the automaton is non-empty.

Consider the Kripke structure $\mathcal{K}$ and the HyperLTL formula φ from the algorithm description. If the following holds: $Tr(\mathcal{K}, s_0), \Pi[\pi_1 \rightarrow t_1, \ldots \pi_n \rightarrow t_n], 0 \models \varphi$ if and only if $(s_0^0 \ldots s_n^0)(s_0^1 \ldots s_n^1) \ldots \in L(A)$, where $t_i = s_i^0 s_i^1 \ldots$ for all traces, we say that automaton A is $\mathcal{K}$-equivalent to the formula φ. Each step of the above algorithm produced a $\mathcal{K}$-equivalent automaton to certain subformula via combining the automaton with $\mathcal{K}$. [24, 4]

One can modify the algorithm, for example, by negating the original formula and finding the nonexistence, so the automaton with the empty language means satisfaction of the formula (in other words, finding the counter-example to prove nonsatisfaction). For a more schematic overview, see Figure 3.1.

3.1.1 Example

To demonstrate the algorithm, consider the HyperLTL formula $\forall\pi_1\exists\pi_2\colon \mathrm{G}(a_{\pi_1} \rightarrow \mathrm{F}\neg a_{\pi_1}) \vee \mathrm{F}b_{\pi_2}$ and a system modeled by the Kripke structure $\mathcal{K} = (S, s_0, \delta, AP, L)$ (Figure 3.2a).

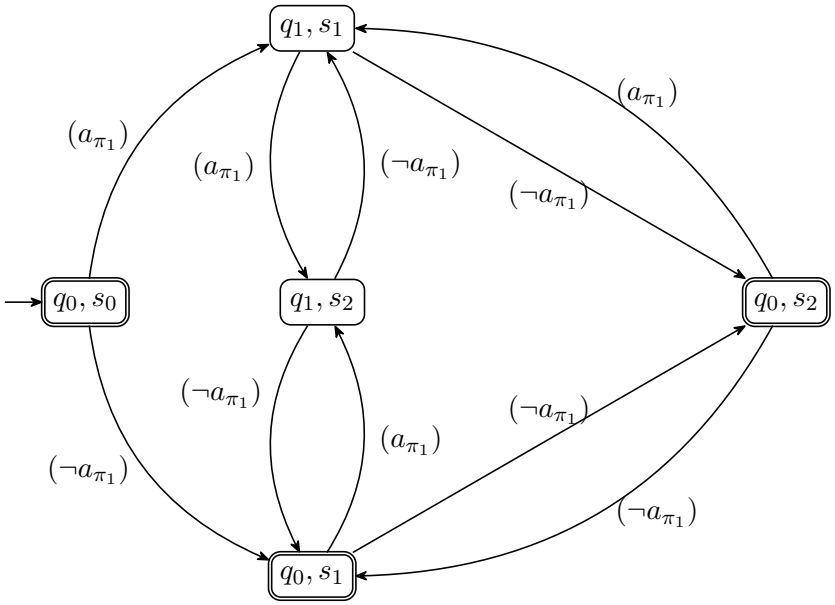


Figure 3.3: NBA A^* as the result of the existential projection of $\mathcal{K}$ onto A.

A nondeterministic Büchi automaton $A = (Q, \Sigma, \delta, \{q_{in}\}, F)$ representing the LTL body of our formula may look like the one in Figure 3.2b.

The transitions of A consist of tuples of size 2, where the first component is the set of APs of trace π_1 and the second one is the set of APs of trace π_2. Since the innermost quantifier is existential, we can now perform the existential projection and build an automaton $A_{\pi_2} = (Q_{\pi_2}, \Sigma_{\pi_2}, \delta_{\pi_2}, I_{\pi_2}, F_{\pi_2})$. To avoid creating unreachable states, we start with the initial states of $\mathcal{K}$ and A and gradually find the successors of each state. According to Section 3.1, the new initial state(s) originate from the product of the initial states of $\mathcal{K}$ and A. Since we have only one initial state in each of them, the new initial state is $I_{\pi_2} = \{(q_0, s_0)\}$, and we have $(q_0, s_0) \in F_{\pi_2}$ since $q_0 \in F$. By applying Eq. 3.1 to find its successors, we obtain the following:

- $(q_1, s_1) \in \delta_{\pi_2}((q_0, s_0), (a_{\pi_1}))$ because s_1 is the successor of s_0 in $\mathcal{K}$, and $q_1 \in \delta(q_0, (a_{\pi_1}, \neg b_{\pi_2}))$ where $L(s_0) = \{a\}$ (indicating that $\neg b$ also holds in that state).
- $(q_0, s_1) \in \delta_{\pi_2}((q_0, s_0), (\neg a_{\pi_1}))$, same as before, s_1 is the successor of s_0 in $\mathcal{K}$, and $q_0 \in \delta(q_0, (\neg a_{\pi_1}, \neg b_{\pi_2}))$ where $L(s_0) = \{a\}$. Moreover, $(q_0, s_1) \in F_{\pi_2}$ due to $q_0 \in F$.
- For example (q_2, s_1) is not a successor of (q_0, s_0) because $\neg b$ holds in s_0, whereas the transition in A from q_0 to q_2 requires b.

The completion of the construction for the automaton A_{π_2} over $\Sigma_{\pi_2} = \{a_{\pi_1}, \neg a_{\pi_1}\}$ follows the same process as described above for each new state resulting in automaton in Figure 3.3. Now that we have eliminated the existential quantifier, a universal quantifier remains in the formula $\forall \pi_1 \colon \varphi^*$. Following the algorithm, we apply double negation, leading to the formula $\neg \exists \pi_1 \colon \neg \varphi^*$. Since we already have the automaton for φ^*, the next step is to complement it. After using Spot [21] to complement A_{π_2}, we obtain the complement automaton $A_C = (Q_C, \Sigma_C, \delta_C, c_0, F_C)$, see Figure 3.4. The next step is to perform existential projection once again, aiming to eliminate the existential quantifier, but this time with respect to the trace variable π_1. Employing the same approach as in the initial elimination, we combine A_C

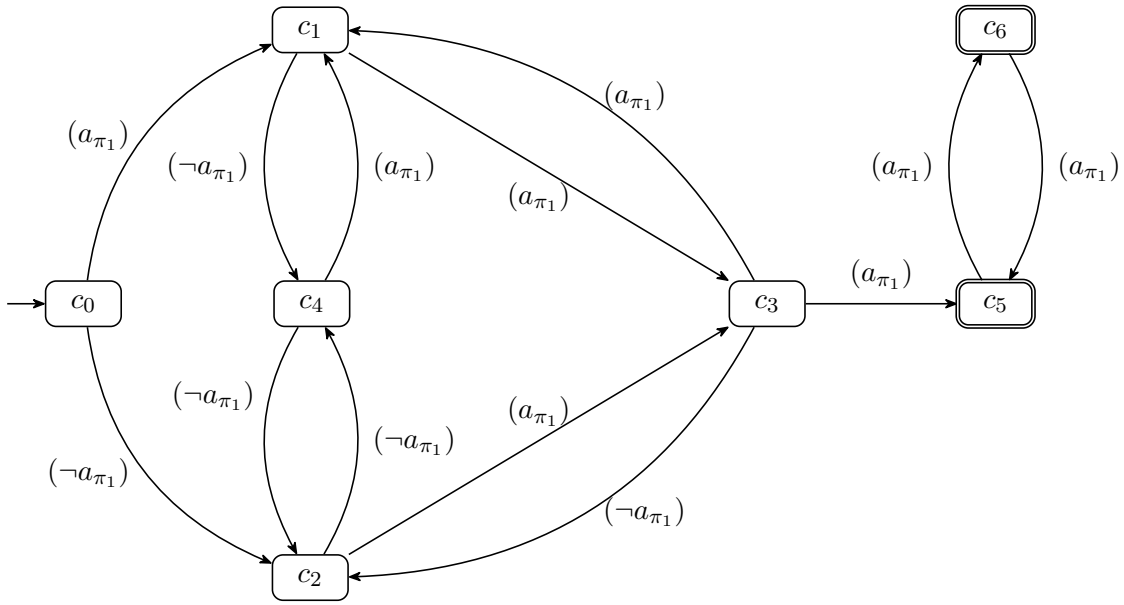


Figure 3.4: NBA A_C as the result of the complementation needed to handle the universal quantifier.

with the Kripke structure $\mathcal{K}$ starting with the initial states. The resulting automaton of this process, denoted as A_r, is depicted in Figure 3.5. The fact that the resulting automaton A_r does not have any accepting states implies that its language is empty. Consequently, we can interpret it as a nonsatisfaction of the system given the specification. However, we must address the negation placed in front of the existential quantifier. This negation would typically entail complementation of A_r. But in this case, it is evident that the complement of an empty language will not be empty. Therefore, our conclusion is that the system does satisfy the given specification. In formal terms, $\mathcal{K} \models \varphi$ holds.

3.2 AutoHyper

In their work [4], Finkbeiner et al. introduce the AutoHyper tool as the first complete push button tool capable of handling model verification of HyperLTL formulas without restrictions on the number of quantifier alternations. AutoHyper employs an automata-based approach introduced above, and they state the following language inclusion property [4]: Let $\varphi^* = \forall\pi_1 \dots \forall\pi_n\varphi$ be a HyperLTL formula (φ may include additional trace quantifiers), and let A_φ be an automaton over $\Sigma = (2^{AP})^n$ that is $\mathcal{K}$-equivalent to φ. Then $\mathcal{K} \models \varphi^*$ if and only if $L(A^n_\mathcal{K}) \subseteq L(A_\varphi)$. Here, $A^n_\mathcal{K}$ is a nondeterministic Büchi automaton over $\Sigma = (2^{AP})^n$, such that for any n-tuple $t_1, t_2, \dots, t_n$ of the traces from $\mathcal{K}$ it holds that $(t_1(0), t_2(0) \dots, t_n(0))(t_1(1), t_2(1) \dots, t_n(1)) \dots \in L(A^n_\mathcal{K})$. The construction of such an automaton would follow Algorithm 1.

While the inclusion check, theoretically optimally done via complementation and the following emptiness check, is no better than the standard automata-based procedure in the worst case scenario [4], AutoHyper capitalizes on the possibility of terminating much earlier when $\mathcal{K} \not\models \varphi^*$ without constructing the entire complement to prove automaton (non)emptiness (by finding an accepting lasso).

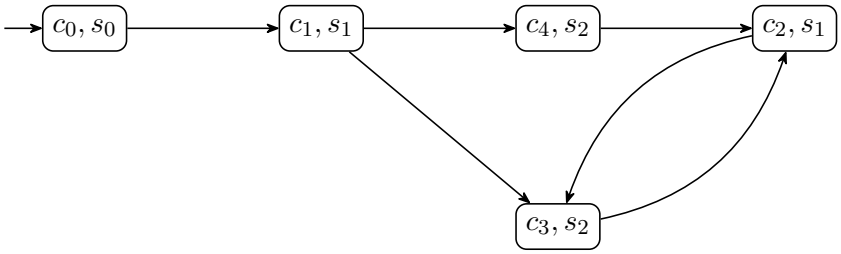


Figure 3.5: The final existential projection results in the NBA A_r over a singleton alphabet, with no accepting states.

AUTOHYPER uses SPOT [21] for LTL-to-NBA conversion, complementation, and as the inclusion checker. In addition to SPOT, it offers several other tools to use as inclusion checkers, namely RABIT [12], BAIT [18], or FORKLIFT [17]. However, in terms of successfully solved instances, SPOT stands out as the most successful among them. On the other hand, there are cases where SPOT is outperformed by some of the alternatives, therefore AUTOHYPER provides the flexibility to use any of them.

Algorithm 1 N-fold self-composition of a Kripke structure

Input: Kripke structure $\mathcal{K} = (S, s_0, \delta, AP, L)$**,** $N \in \mathbb{N}^+$
Output: NBA $A = (Q, \Sigma, \delta, I, F)$
$Q \leftarrow \emptyset$, $\Sigma \leftarrow (2^{AP})^N$, $\delta \leftarrow \emptyset$, $I \leftarrow \{(s_0)^N\}$, $F \leftarrow \emptyset$
Queue q
$q.Enqueue(I)$
while $q.nonempty()$ **do**
  $current \leftarrow q.Dequeue()$
  $trans\_cond \leftarrow 1$ ▷ neutral for conjunction
  $succs \leftarrow \{\emptyset\}^N$
  for each $0 \leq i \leq N$ **do**
    $state \leftarrow current[i]$
    $trans\_cond \leftarrow trans\_cond \wedge L(state)$
    $succs[i] \leftarrow get\_successors\_of(state)$
  end for

  $all\_possible\_comb \leftarrow product(succs, N)$ ▷ product of N same sets of successors
  for each dst_state **of** $all\_possible\_comb$ **do**
    if $state \notin Q$ **then**
      $q.Enqueue(dst\_state)$
      $Q.add(dst\_state)$
    end if
    $\delta.add(current, trans\_cond, dst\_state)$
  end for
end while
$F \leftarrow Q$

return $(Q, \Sigma, \delta, I, F)$

Chapter 4

Subsumption relations

One of the approaches to decide $L(A) \subseteq L(B)$ between ω-automata A and B is to decide whether the language of an automaton resulting from $L(A) \cap \overline{L(B)}$ is empty. When constructing the product automaton on-the-fly, we can use the information gathered before and during construction to predict and cut off unnecessary parts or find counterexamples without explicitly constructing them. This approach extends far beyond the HyperLTL model checking and thus can be utilized in various problems that include emptiness check.

This chapter introduces the necessary notions regarding *simulations* on ω-automata. We then define and combine various relations and theorems for an efficient inclusion check, specifically when leveraging modular complementation from Kofola [26].

4.1 Simulations

There is a wide range of simulations over ω-automata. Simulations allow us to relate states not only by whether they accept the same ω-words, but they allow us also to reason about traces. Simulation can be defined as the game of two players [13], *Spoiler* and *Duplicator*. Consider the following initial configuration of the game. The *Spoiler* starts in the state s_0 and *Duplicator* starts in state d_0. In each round of the game, *Spoiler* chooses $a_i \in \Sigma$ and picks a transition such that $s_i \xrightarrow{a_i} s_{i+1} \in \delta$. *Duplicator* has to pick the corresponding transition such that $d_i \xrightarrow{a_i} d_{i+1} \in \delta$. Assuming the automaton is complete, there are two infinite traces, $\pi_s = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \ldots$ and $\pi_d = d_0 \xrightarrow{a_0} d_1 \xrightarrow{a_1} d_2 \ldots$ According to the winning condition for *Duplicator*, we distinguish the *direct* (di), *delayed* (de) and *fair* (f) simulation. Let $x \in \{di, de, f\}$, the *Duplicator* wins when $\mathcal{C}^x(\pi_s, \pi_d)$ holds [11]:

$$\mathcal{C}^{di}(\pi_s, \pi_d) \overset{\text{def}}{\iff} \forall i \geq 0 \colon s_i \in F \implies d_i \in F, \tag{4.1}$$

$$\mathcal{C}^{de}(\pi_s, \pi_d) \overset{\text{def}}{\iff} \forall i \geq 0 \colon s_i \in F \implies \exists j \geq i \colon d_j \in F, \tag{4.2}$$

$$\mathcal{C}^{f}(\pi_s, \pi_d) \overset{\text{def}}{\iff} \pi_s \text{ is fair} \implies \pi_d \text{ is fair}, \tag{4.3}$$

where an infinite trace is *fair* if and only if it visits the accepting state(s) infinitely often. Whenever the winning condition is not met, the winner is *Spoiler*. To denote that the state p is *direct simulated* by the state q, we use the notation $p \preceq_{di} q$. This definition would be trivially extended to transition-based automata.

4.1.1 Early simulations

This section defines *early* and *early+1* simulations as introduced in [8], but for transition-based Büchi automata.

Definition 1. *Consider the Büchi automaton* $A = (Q, \Sigma, \delta, I, F)$ *and the traces* $\pi_p = p_0 w_0 p_1 w_1 \dots$ *and* $\pi_r = r_0 w_0 r_1 w_1 \dots$ *over the same word* $w = w_0 w_1 \dots \in \Sigma^\omega$*, where each* $p_i, r_i \in Q$*. Trace* π_p *is early simulated by* π_r*, denoted as* $\pi_p \preceq_e \pi_r$*,if and only if*

$$\forall i < j \colon ((p_i \xrightarrow{a} p_{i+1} \in F \vee i = -1) \wedge p_j \xrightarrow{a} p_{j+1} \in F) \implies \exists i < k \leq j \colon r_k \xrightarrow{a} r_{k+1} \in F.$$

Similarly, trace π_p *is early+1 simulated by* π_r*, denoted as* $\pi_p \preceq_{e^{+1}} \pi_r$*,if and only if*

$$\forall i < j \colon (p_i \xrightarrow{a} p_{i+1} \in F \wedge p_j \xrightarrow{a} p_{j+1} \in F) \implies \exists i < k \leq j \colon r_k \xrightarrow{a} r_{k+1} \in F.$$

For *early+1* simulation this means that there is one accepting state in the π_r for every two accepting states in the π_p and *early* simulation also requires that π_r visits accepting state no later than π_p.

To be able to express these relations on the states, [8] provides the following definitions.

Definition 2. *Strategy is a function* $\delta_s \colon Q \times (Q \times \Sigma \times Q) \to (Q \times \Sigma \times Q)$ *such that* $\delta_s(r, (p, a, p')) = (r, a, r')$ *where* $r' \in \delta(r, a)$*.*

In other words, *strategy* function chooses a transition from state r based on transition (p, a, p').

Definition 3. *Strategy for traces is a function* $\delta_t \colon Q \times (Q \times \Sigma)^\omega \to (Q \times \Sigma)^\omega$ *such that* $\delta_t(r_0, \pi_p) = r_0 w_0 r_1 w_1 \dots$ *where* $\delta_s(r_i, (p_i, w_i, p_{i+1})) = (r_i, w_i, r_{i+1})$ *holds for all* $i \geq 0$*.*

It is essentially choosing the successors of r_i following the trace π_p.

Definition 4. *State* p_0 *is early (early+1) simulated by state* r_0*, denoted as* $p_0 \preceq_e r_0$ *(*$p_0 \preceq_{e^{+1}} r_0$*), if and only if there is a strategy function* δ_t *such that* $\pi_p \preceq_e \delta_t(r_0, \pi_p)$ *(*$\pi_p \preceq_{e^{+1}} \delta_t(r_0, \pi_p)$*) holds for each trace* π_p *starting in* p_0*.*

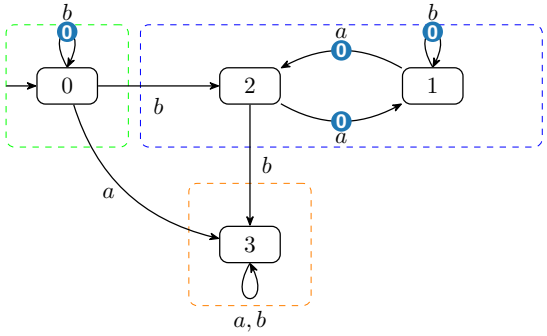
The language of a state $p \in Q$ from (TG)BA $A = (Q, \Sigma, \delta, I, F)$ is defined as $L(p) = \{w \mid \exists$ an accepting trace from p in A over $w\}$. By $\subseteq_L$ we denote the relation of language inclusion of states, $p \subseteq_L q \iff L(p) \subseteq L(q)$ [8].

Proposition 1. *For the relations over the states of BA* A *the following holds [8]:*

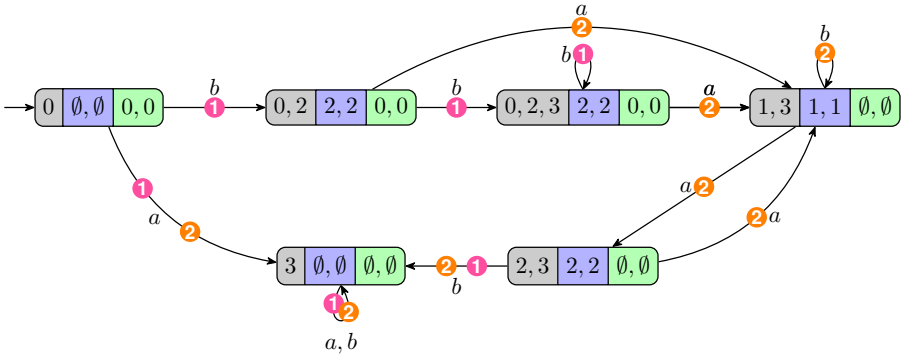
$$\preceq_e \quad \subseteq \quad \preceq_{e^{+1}} \quad \subseteq \quad \subseteq_L .$$

4.2 Modular Complementation of Büchi Automata

This thesis builds on the tool Kofola [26], which employs a modular-based complementation approach (note that whenever we talk about modular complementation, it is a reference to Kofola's approach). Prior to complementation, the input automaton is divided into *partition blocks* (a partition block is a group of strongly connected components, where a group consists of at least one SCC) based on the structure of the strongly connected components. An example of partitions can be seen in Figure 4.1a. Then, for each partition block, the most suitable complementation algorithm is determined. Subsequently, the tool



(a) Identified partitions of automaton A_{ex} (in this case a partition is an SCC). Green and blue frames each contain an inherently weak accepting component, the orange frame contains a non-accepting component (does not produce a partial macrostate).



(b) The outcome of the modular complementation of the automaton A_{ex}. The blue part of the macrostates corresponds to the partition containing the blue SCC from A_{ex} in (a), and the green part corresponds to the partition containing the green SCC from A_{ex}. An instance of the MH procedure is applied to each of the two. The resulting automaton has the acceptance condition $Acc = Inf(1) \wedge Inf(2)$ where $Inf(1)$ is from MH for the green partition and $Inf(2)$ is from MH for the blue partition.

Figure 4.1: Example of modular complementation showing the input automaton and the resulting automaton.

performs the complementation for each partition block using either *synchronous* construction or a *postponed* construction. The acceptance condition of the complement produced by Kofola can potentially be more general - a conjuction of partial acceptance conditions - which is one of the benefits of this approach (it can lead to smaller automata).

Postponed. In the postponed construction, the complementation of each partition block is performed independently. The result is then obtained using the product construction of the partial complements of the partition blocks. This approach is appropriate for applications that require the entire complement, as reductions on the partial results can be applied.

Synchronous. The synchronous construction *synchronizes* the complementation of each partition in each step. For example, consider an input automaton consisting of three SCCs of different types. The states produced by modular complementation are of the form (N, S_1, S_2, S_3), where N is the set for tracking all runs and S_i is the state containing runs within the i-th SCC. In each step, the successors of states in N are computed and with respect to them, the successors of each S_1, S_2, and S_3 are computed according to partial procedures. Creating the whole state of the complement in each step makes it fitting for

on-the-fly applications (such as inclusion testing) because there are cases where we can determine the result without the need for constructing the entire complement automaton. However, compared to the postponed construction, it may suffer from generating useless states.

4.2.1 Inherently Weak Accepting Components

One of the partial algorithms that KOFOLA uses is the standard *Miyano-Hayashi*(`MH`) complementation procedure [35] for inherently weak Büchi automata. If an SCC is *inherently weak*, all its cycles are either accepting or rejecting (for rejecting it means that the SCC has an empty language). The approach in KOFOLA therefore assumes only *inherently weak accepting* components, whose complementation is not trivial. The task is then to track all the runs in such an SCC (or a partition) and to determine whether they leave it eventually. If they leave it infinitely often for a certain ω-word, the complement automaton will accept such a word. To be precise and coherent with KOFOLA, consider the input automaton $A = (Q, \Sigma, \delta, I, F)$ and pick a partition block P of inherently weak accepting SCCs. This procedure within the modular complementation produces *macrostates*[1] of the form (C, B), where C stands for *check* and represents runs in P and $B \subseteq C$ stands for *breakpoint* and contains the runs that are being inspected whether they leave P. Each macrostate has exactly one successor given by the following transition function: $\delta_C(N, (C, B), a) = (C', B')$, where N denotes all current runs within the automaton A, $C' = \delta(N, a) \cap P$ and when $\delta(B, a) \cap C' = \emptyset$ then $B' = C'$, otherwise $B' = \delta(B, a) \cap C'$. The transition $(C, B) \xrightarrow{a} (C', B')$ is accepting when $\delta(B, a) \cap C' = \emptyset$. Details are omitted, the full specification of the algorithm plugged into KOFOLA can be found in [26]. An example can be seen in Figure 4.1.

We can now delve into the definition of the proposed subsumptions. Suppose macrostates $p = (N_p, C_p, B_p)$ and $r = (N_r, C_r, B_r)$ with N, N' representing the sets of all currently visited states. Thus, $C \subseteq N$ and $C' \subseteq N'$. Note that the sets N, N' are always present in the modular procedure, which implies that we can use them freely. Let us define the following subsumption relation:

$$p \sqsubseteq r \overset{def}{\iff} N_p \supseteq N_r \wedge C_p \supseteq C_r.$$

We also define a similar but stronger relation:

$$p \sqsubseteq^B r \overset{def}{\iff} N_p \supseteq N_r \wedge C_p \supseteq C_r \wedge B_p \supseteq B_r.$$

Lemma 1. *The relation $\sqsubseteq$ is an early+1 simulation:*

$$p \sqsubseteq r \implies p \preceq_{e^{+1}} r.$$

Proof. This proof follows the structure of the proof in [8].
We need to find the suitable strategy $\delta_t(r, \pi_p)$ according to Definition 4. As we stated above, the *Miyano-Hayashi* complementation procedure produces deterministic transitions. Therefore, the strategy is implicitly defined by the output's characteristics. Namely, we use strategy $\delta_\sqsubseteq$ which for a transition (p, a, p') chooses a transition (r, a, r'), with $p' = (N_{p'}, C_{p'}, B_{p'})$ and $r' = (N_{r'}, C_{r'}, B_{r'})$, such that $\delta_C(N_r, (C_r, B_r), a) = (C_{r'}, B_{r'})$. And given that $N_r \subseteq N_p$, also $C_{r'} \subseteq C_{p'}$. Putting the strategy in place guarantees $p \sqsubseteq r \implies p' \sqsubseteq r'$, the consequence of determinism and completeness is that (r, a, r') exists. It is also evident that, whenever a state is removed from B_p, it is also removed from B_r if present.

[1] By term *macrostate* we mean a state consisting of more states (typically sets of states of the original automaton).

Now we show that for any two traces $\pi_p = p_0 w_0 p_1 w_1 \dots$ and $\pi_r = \delta_{\sqsubseteq}(r_0, \pi_p) = r_0 w_0 r_1 w_1 \dots$, with $p = p_0$ and $r = r_0$, the condition $\pi_p \preceq_{e^{+1}} \pi_r$ is satisfied. By definition, it is necessary to prove that $\forall i < j \colon (p_i \xrightarrow{w_i} p_{i+1} \in F_C \wedge p_j \xrightarrow{w_j} p_{j+1} \in F_C) \implies \exists i < k \leq j \colon r_k \xrightarrow{w_k} r_{k+1} \in F_C$, where F_C is the set of accepting macrostates of A_C.
Claim 1: For all $i \geq 0$, if $p_i \xrightarrow{w_i} p_{i+1} \in F_C$, then $p_{i+1} \sqsubseteq^B r_{i+1}$.
Proof: Suppose that $p_{i+1} = (N_{p_{i+1}}, C_{p_{i+1}}, B_{p_{i+1}})$ and $r_{i+1} = (N_{r_{i+1}}, C_{r_{i+1}}, B_{r_{i+1}})$. Since $p_i \xrightarrow{w_i} p_{i+1}$ is accepting, it holds that (i) $B_{p_{i+1}} = C_{p_{i+1}}$, (ii) $C_{p_{i+1}} \supseteq C_{r_{i+1}}$ since $p_{i+1} \sqsubseteq r_{i+1} \implies N_{p_{i+1}} \supseteq N_{r_{i+1}} \implies (C_{p_{i+1}} = N_{p_{i+1}} \cap P) \supseteq (C_{r_{i+1}} = N_{r_{i+1}} \cap P)$ and (iii) $C_{r_{i+1}} \supseteq B_{r_{i+1}}$. From $(B_{p_{i+1}} = C_{p_{i+1}}) \supseteq C_{r_{i+1}} \supseteq B_{r_{i+1}}$, it follows that $B_{p_{i+1}} \supseteq B_{r_{i+1}}$, implying $p_{i+1} \sqsubseteq^B r_{i+1}$. ■
Claim 2: If $p_i \sqsubseteq^B r_i$ and $p_j \xrightarrow{w_j} p_{j+1} \in F_C$ for some $i < j$, then there exists $i < k \leq j$ such that $r_k \xrightarrow{w_k} r_{k+1} \in F_C$.
Proof: From $p_i \sqsubseteq^B r_i$ we have $B_{p_i} \supseteq B_{r_i}$ and our strategy guarantees that in this case B_{r_i} will be emptied no later than B_{p_i}. ■

The two claims imply $\pi_p \preceq_{e^{+1}} \pi_r$. The conclusion is that our strategy meets the requirements of Definition 4. □

Lemma 2. *The relation $\sqsubseteq^B$ is an early simulation:*

$$p \sqsubseteq^B r \implies p \preceq_e r.$$

Proof. The strategy $\delta_{\sqsubseteq}$ we use is the same as in the proof of Lemma 1. Now we show that for any two traces $\pi_p = p_0 w_0 p_1 w_1 \dots$ and $\pi_r = \delta_{\sqsubseteq}(r_0, \pi_p) = r_0 w_0 r_1 w_1 \dots$ it follows that $\pi_p \preceq_e \pi_r$. Similarly as in the proof in [8] we restate Definition 1 of *early* simulation in the following conjunction:

$$\forall i < j \colon (p_i \xrightarrow{w_i} p_{i+1} \in F_C \wedge p_j \xrightarrow{w_j} p_{j+1} \in F_C) \implies \exists i < k \leq j \colon r_k \xrightarrow{w_k} r_{k+1} \in F_C, \tag{4.4}$$

$$p_i \xrightarrow{w_i} p_{i+1} \in F \implies \exists k \leq i \colon r_k \xrightarrow{w_k} r_{k+1} \in F. \tag{4.5}$$

The condition 4.4 is the same as for the definition of $\sqsubseteq$ and since $\sqsubseteq^B$ is stronger, it follows that $p \sqsubseteq r$. With strategy $\delta_{\sqsubseteq}$ being the same as in the proof of Lemma 1, Condition 4.4 is satisfied. Condition 4.5 follows from the second claim in the proof of Lemma 1. □

4.2.2 Deterministic Accepting Components

For deterministic accepting components (DACs) Kofola [26] uses an approach based on the NCSB construction [6] for the complementation of semi-deterministic Büchi automata. In this section, we will prove that there exists *early* simulation between states of the particular partial algorithm used within Kofola.

The most important aspect is again the computation of the successors. So let us briefly sum up the definition provided in Kofola. As referred in [26], a partial algorithm `CSB` uses a set B similarly as `MH` for tracking the runs that eventually leave the SCC, further it uses a set S for storing runs which it guessed will not visit accepting transitions anymore (*safe runs*). A set C then contains runs that has not yet been decided as *safe* nor have they been sampled into B. To avoid transitions between SCCs of the partition P (so that we can treat all runs as deterministic) δ_{SCC} is used. Moreover, transition function that only returns accepting transitions (if present) δ_F is used. We define $\delta_{CSB}(N, (C, S, B), a) = U$ [26] such that:

- if $\delta_F(S,a) \neq \emptyset$, then $U = \emptyset$,
- otherwise:
  - U includes (C', S', B'), where:
    - $C' = (\delta(N,a) \cap P) \setminus S'$,
    - $S' = \delta_{SCC}(S,a) \cap P$,
    - $B' = C'$ if $\delta_{SCC}(B,a) = \emptyset$, otherwise $B' = \delta_{SCC}(B,a)$.
      We refer to this type of transition as *emptying*. The transition $(C,S,B) \xrightarrow{a} (C',S',B')$ is accepting if $\delta_{SCC}(B,a) = \emptyset$.
  - If $\delta_{SCC}(B,a) \cap \delta_F(B,a) = \emptyset$, U also contains (C'', S'', C''), where:
    - $C'' = C' \setminus S''$,
    - $S'' = S' \cup B'$.
      The transition $(C,S,B) \xrightarrow{a} (C'', S'', C'')$ is always accepting, and we refer to this type of transition as *safeing*.

Now, similarly as for `MH` macrostates, we define the relation for the `CSB` macrostates. More specifically, consider macrostates $p = (N_p, C_p, S_p, B_p)$ and $r = (N_r, C_r, S_r, B_r)$; then

$$p \sqsubseteq_{CSB} r \overset{def}{\iff} N_p \supseteq N_r \wedge S_p \supseteq S_r \wedge (S_p \cup B_p) \supseteq B_r.$$

Lemma 3. *The relation $\sqsubseteq_{CSB}$ is an early simulation:*

$$p \sqsubseteq_{CSB} r \implies p \preceq_e r.$$

Proof. First, let us state the following facts:

1. $N \subseteq N' \implies \delta(N,a) \subseteq \delta(N',a)$,
2. $S \cup C = N \cap P$, and
3. if a run ρ is moved to S on an infinite trace, then it stays *safe* forever or becomes discontinued.

From each state of the `CSB` construction, there are at most two successors. We use a strategy $\delta_{\sqsubseteq_{CSB}}$ that if $p_i \xrightarrow{a_i} p_{i+1}$ is *safeing*, we also take *safeing* transition from r_i if possible. This is the only scenario when *safeing* transition is picked from r_i. In all other cases, *emptying* transition is chosen.
Claim 3: If $p_i \sqsubseteq_{CSB} r_i$ and $p_i \xrightarrow{a_i} p_{i+1} \notin F_C$ and $r_i \xrightarrow{a_i} r_{i+1} \notin F_C$, then $p_{i+1} \sqsubseteq_{CSB} r_{i+1}$.
Proof: For $S_{p_{i+1}} = \delta_{SCC}(S_{p_i}, a)$ and $B_{p_{i+1}} = \delta_{SCC}(B_{p_i}, a)$. For $B_{r_{i+1}}$, we have $B_{r_{i+1}} = \delta_{SCC}(B_{r_i}, a)$. From $(S_{p_i} \cup B_{p_i}) \supseteq B_{r_i}$, it holds that $(S_{p_{i+1}} \cup B_{p_{i+1}}) = (\delta_{SCC}(S_{p_i}, a) \cup \delta_{SCC}(B_{p_i}, a)) \supseteq \delta_{SCC}(B_{r_i}, a) = B_{r_{i+1}}$ and $S_{p_{i+1}} = \delta_{SCC}(S_{p_i}, a) \supseteq \delta_{SCC}(S_{r_i}, a) = S_{r_{i+1}}$. ■

Claim 4: If $p_i \sqsubseteq_{CSB} r_i$ and $p_j \xrightarrow{a_j} p_{j+1} \in F_C$, then there is a k such that $i \leq k \leq j$ and $r_k \xrightarrow{a_k} r_{k+1} \in F_C$.
Proof: We split the proof into two cases.

(I) $p_j \xrightarrow{a_j} p_{j+1} \in F_C$ is *emptying*.

(I.i) Suppose $B_{p_i} \supseteq B_{r_i}$, then B_{r_i} becomes empty (and emits an accepting mark) no later than B_{p_i}.

(I.ii) Otherwise, there was no *emptying* transition up to r_j, so we use Claim 3. Since $p_j \xrightarrow{a_j} p_{j+1}$ is *emptying*, B_{p_j} becomes empty. Therefore, each run from $(S_{p_j} \cup B_{p_j}) \supseteq B_{r_j}$ is discontinued or safe, which means $\delta_{SCC}(B_{r_j}, a) \cap \delta_F(B_{r_j}, a) = \emptyset$. That proves the existence of the *safeing* transition from r_j (which we take according to our strategy).

(II) $p_j \xrightarrow{a_j} p_{j+1} \in F_C$ is *safeing*.

(II.i) Suppose $B_{p_i} \supseteq B_{r_i}$ then either B_{r_i} becomes empty sooner than $p_j \xrightarrow{a_j} p_{j+1}$ or $B_{p_j} \supseteq B_{r_j}$ and $r_j \xrightarrow{a_j} r_{j+1}$ is also *safeing*.

(II.ii) Suppose that $B_{p_i} \not\supseteq B_{r_i}$ and no *emptying* transition from r_i to r_j (worst case). Then each run from $(S_{p_j} \cup B_{p_j}) \supseteq B_{r_j}$ is moved to safe set. Therefore also the transition $r_j \xrightarrow{a_j} r_{j+1}$ can be *safeing* (which is picked by our strategy).

■

Claim 5: If $p \sqsubseteq_{CSB} r$ then for all i it holds that $S_{p_i} \supseteq S_{r_i}$.
Proof: This follows from our strategy not picking *safeing* transition (*emptying* transition does not alter set S) if not necessary. We can only pick *safeing* transition on π_r on the exact same position as present on π_p, for example position i. Utilizing Claim 3 and *emptying* transition not altering set S, the only alternation happens when *safeing* transition occur. In that case, suppose it happens for $p_i \xrightarrow{a_i} p_{i+1}$ and $r_i \xrightarrow{a_i} r_{i+1}$. Clearly, it holds that $p_i \sqsubseteq_{CSB} r_i$ (according to Claim 3). We start with $N_{p_i} \supseteq N_{r_i} \wedge S_{p_i} \supseteq S_{r_i} \wedge (S_{p_i} \cup B_{p_i}) \supseteq B_{r_i}$. Since *safeing* transition moves all runs from set B to set S, all runs from $(S_{p_i} \cup B_{p_i}) \supseteq B_{r_i}$ become safe together with $S_{p_i} \supseteq S_{r_i}$, leading to $S_{p_{i+1}} \supseteq S_{r_{i+1}}$. ■

Claim 6: If $p \sqsubseteq_{CSB} r$ and $p_i \xrightarrow{a_i} p_{i+1} \in F_C$, then $p_{i+1} \sqsubseteq_{CSB} r_{i+1}$.
Proof: Whenever the accepting transition is encountered, we have $C = B$; additionally from Fact 2 we have $S \cup C = N \cap P$. In this particular case, for $p_{i_{i+1}}$ it holds that $C_{p_{i+1}} = B_{p_{i+1}}$ and thus $S_{p_{i+1}} \cup C_{p_{i+1}} = S_{p_{i+1}} \cup B_{p_{i+1}} = N_{p_{i+1}} \cap P$. From Fact 1 and $p \sqsubseteq_{CSB} r$ we inductively have $N_{p_{i+1}} \supseteq N_{r_{i+1}}$, therefore $N_{p_{i+1}} \cap P \supseteq N_{r_{i+1}} \cap P$. From definition, for r_{i+1} it certainly holds that $N_{r_{i+1}} \cap P \supseteq C_{r_{i+1}} \supseteq B_{r_{i+1}}$. Putting everything together, we have $(S_{p_{i+1}} \cup B_{p_{i+1}}) \supseteq B_{r_{i+1}}$, and thanks to Claim 5 $S_{p_{i+1}} \supseteq S_{r_{i+1}}$. ■
The validity of Claims 3, 4 and 6 concludes the proof, together with Claim 5 ensuring existence of transitions. □

4.3 Early Simulations for TGBA

Due to the fact that the result of the product construction $A \cap B$ of two (TG)BAs, is also a TGBA, we need to define *early* and *early+1* simulation for TGBAs.

Definition 5. *Consider TGBA A with acceptance $Acc = Inf(0) \wedge \ldots \wedge Inf(n)$. Next, suppose two traces $\pi_p = p_0 w_0 p_1 \ldots$ and $\pi_q = q_0 w_0 q_1 \ldots$ of the automaton A. We say that $\pi_p \preceq_{e^{+1}} \pi_q$ if the following holds:*

$$\forall 0 \leq m \leq n \colon \forall i < j \colon (p_i \xrightarrow{w_i} p_{i+1} \in F_m \wedge p_j \xrightarrow{w_j} p_{j+1} \in F_m) \implies$$
$$\implies \exists i < k \leq j \colon r_k \xrightarrow{w_k} r_{k+1} \in F_m.$$

Similarly, for early simulation, $\pi_p \preceq_e \pi_q$ if:

$$\forall 0 \leq m \leq n \colon \forall i < j \colon ((p_i \xrightarrow{w_i} p_{i+1} \in F_m \vee i = -1) \wedge p_j \xrightarrow{w_j} p_{j+1} \in F_m) \implies$$
$$\implies \exists i < k \leq j \colon r_k \xrightarrow{w_k} r_{k+1} \in F_m.$$

4.4 Early termination in Kofola

This section provides observations that can be used within KOFOLA for an efficient inclusion and emptiness checking.

In this section, we assume a TGBA $A_{TGBA} = (Q, \delta, I, \Gamma, p, Acc)$ over Σ such that $\Gamma = \{0, \ldots, k-1\}$. The fact that there is a transition t such that $p(t) = i$ (i.e., accepting) for some selected color $0 \leq i \leq k-1$ on the path over u from state p to state q, is denoted by $p \overset{u}{\underset{\bullet}{\rightsquigarrow}} q$. By $L(p)$ we denote the set of ω-words accepted from state p and $L(p)^i$, moreover, adds a restriction to $\Gamma = \{i\}$ (intuitively simplifying the acceptance condition of A_{TGBA} to $Acc = Inf(i)$).

Firstly, we begin with the observations that help in reporting counterexamples without fully constructing them. Then, additional observations are introduced to reduce the explored state space (which is particularly useful when inclusion holds).

4.4.1 Early counterexample reporting

The first observation is that whenever we reach the state q from the state p over some $u \in \Sigma^*$ such that $p \preceq_{e^{+1}} q$ and there were at least two accepting transitions between them, the *early+1* simulation ensures infinite generating of accepting transitions from q, so we can decide the language of the automaton as nonempty at this moment. For TGBAs, it means seeing at least two accepting transitions for each color $0 \leq i \leq k-1$.

Theorem 2. *If TGBA A has $Acc = Inf(0) \wedge \ldots \wedge Inf(k-1)$ then it holds that*

$$(\forall\, 0 \leq i \leq k-1 \colon p \overset{u}{\underset{\bullet}{\rightsquigarrow}}\overset{u}{\underset{\bullet}{\rightsquigarrow}} q \wedge p \preceq_{e^{+1}} q) \implies L(p)^i \neq \emptyset.$$

Lemma 4. $(p \overset{u}{\underset{\bullet}{\rightsquigarrow}}\overset{u}{\underset{\bullet}{\rightsquigarrow}} q \wedge p \preceq_{e^{+1}} q) \implies u^\omega \in L(p)^i$

Proof. In this proof we use the notation $\pi_x(u)$ representing the finite trace from the state x when reading $u \in \Sigma^*$. We also use $\pi_x^\omega(u)$ to denote the infinite trace from state x over $u^\omega \in \Sigma^\omega$.
Suppose a finite trace $\pi_p(u) = p_0 \to p_1 \to \ldots \to p_i$, where $p_0 = p, p_i = q$, and a finite trace $\pi_q(u) = q_0 \to q_1 \to \ldots \to q_i$, where $q_0 = q$. The existence of π_q is given by the fact that the strategy function exists, from Definition 4. Let us say, that accepting transitions are $p_j \to p_{j+1}$ and $p_m \to p_{m+1}$ where $0 \leq j < m < i$. Since $p \preceq_{e^{+1}} q$, the trace π_q must see the accepting transition in position k such that $j < k \leq m$. That means that also the transition $q_k \to q_{k+1}$ is accepting. With trace $\pi_q^\omega(u) = q_0 \to q_1 \ldots \to q_i \ldots$ being the suffix of trace $\pi_p^\omega(u) = p_0 \to p_1 \ldots \to q_0 \to q_1 \to \ldots \to q_i \to \ldots$, the exact same transition is accepting on trace $\pi_p^\omega(u)$. Now we have that the position of $p_j \to p_{j+1}$ on trace $\pi_p^\omega(u)$ is j, the position of $p_m \to p_{m+1}$ on $\pi_p^\omega(u)$ is m, position of $q_k \to q_{k+1}$ on $\pi_p^\omega(u)$ is $i+k$ and k on $\pi_q^\omega(u)$. Clearly, it holds that

$$j_{\pi_p^\omega(u)} < k_{\pi_q^\omega(u)} \leq m_{\pi_p^\omega(u)} < (i+k)_{\pi_p^\omega(u)}.$$

In other words, for accepting transitions on position m and $i+k$ on trace $\pi_p^\omega(u)$ there has to be some accepting transition $q_n \to q_{n+1}$ on $\pi_q^\omega(u)$, therefore also $q_n \to q_{n+1}$ on position $i+n$ on $\pi_p^\omega(u)$, such that

$$j_{\pi_p^\omega(u)} < k_{\pi_q^\omega(u)} \leq m_{\pi_p^\omega(u)} < n_{\pi_q^\omega(u)} \leq (i+k)_{\pi_p^\omega(u)} < (i+n)_{\pi_p^\omega(u)}.$$

Such generation will continue infinitely, which concludes Lemma 4. Theorem 2 is the direct consequence of Lemma 4. □

The next observation is similar to the one in Theorem 2, but for *early* simulation it is sufficient to see only one accepting transition.

Theorem 3. $(\forall\, 0 \leq i \leq k-1 \colon p \overset{u}{\underset{\bullet}{\rightsquigarrow}} q \wedge p \preceq_e q) \implies L(p) \neq \emptyset$

Lemma 5. $(p \overset{u}{\underset{\bullet}{\rightsquigarrow}} q \wedge p \preceq_e q) \implies u^\omega \in L(p)^i$

Proof. The proof would be carried out in a way similar to the proof of Lemma 4. Intuitively, *early* simulation ensures existence of the accepting transition on the path from p to q within the trace π_p implies the accepting transition on the path from q within the trace π_q. That, however, creates a second accepting transition on the trace π_p without a corresponding accepting transition on the trace π_q. *Early* simulation ensures that there is another accepting transition on π_q, which also implies another accepting transition on π_p. This happens infinitely often, which proves Lemma 5. The validity of Lemma 5 proves Theorem 5. □

Now we would like to extend such an approach to Kofola's macrostates, making use of the partial macrostates included in them. Therefore, we have to consider Definition 5 provided for TGBA.

Theorem 4. *Consider macrostates* $p = (A_0^p, \ldots, A_n^p)$ *and* $q = (A_0^q, \ldots, A_n^q)$ *such that* $A_i^p \preceq_{e+1} A_i^q$ *for each* $0 \leq i \leq n$. *Then the following holds:*

$$p \preceq_{e+1} q.$$

Lemma 6. *There is a strategy function* δ_t *such that* $\pi_p \preceq_{e+1} \delta_t(q_0, \pi_p)$ *for each trace* π_p *starting in* p_0.

Proof. We define the strategy function δ_t with respect to partial strategy functions whose existence is given by the *early+1* relation between partial macrostates. More specifically, function δ_i for each pair A_i^p, A_i^q. The strategy δ_t is then defined as follows:

$$\delta_t(q_0, \pi_p) = q_0 \overset{w_0}{\to} q_1 \overset{w_1}{\to} \ldots,$$

such that

$$\delta(q_i, (p_i, w_i, p_{i+1})) = q_i \overset{w_i}{\to} q_{i+1},$$

where

$$q_{i+1} = (\delta_0(A_0^{q_i}, (A_0^{p_i}, w_i, A_0^{p_{i+1}})), \ldots, \delta_n(A_n^{q_i}, (A_n^{p_i}, w_i, A_n^{p_{i+1}}))).$$

The sequence of j-th macrostate components, where $0 \leq j \leq n$, creates a subtrace, for which the Definition 1 (for *early+1*) holds for $F = p^{-1}(j)$. All n such subtraces create one trace of macrostates. Therefore, for each $0 \leq j \leq n$ Definition 1 of *early+1* simulation holds, where $F_j = p^{-1}(j)$. That is exactly Definition 5. This proves validity of Lemma 6, which proves Theorem 4.

□

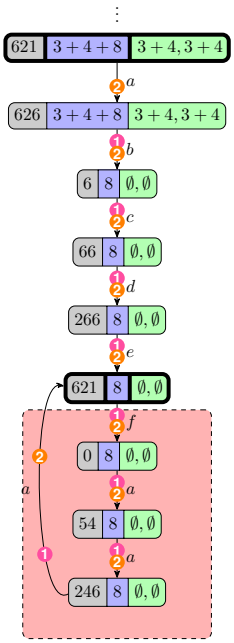


Figure 4.2: Fragment of the product automaton $A \cap B_{compl}$ with $Acc = Inf(1) \wedge Inf(2)$ demonstrates the effect of an optimization that uses Theorem 3 and Theorem 4, where $p = (621 \mid 3+4+8 \mid 3+4, 3+4)$ and $q = (621 \mid 8 \mid \emptyset, \emptyset)$. Each of them being the product macrostate of the form $(x \mid N \mid C, B)$ such that $x \in Q_A$, $N \subseteq Q_B$ and C, B being sets from the `MH` construction (with elements belonging to the same set presented as sums). Then, the exploration of the red part can be omitted completely and the algorithm can terminate with stating nonemptiness.

Theorem 5. *Consider macrostates $p = (A_0^p, \ldots, A_n^p)$ and $q = (A_0^q, \ldots, A_n^q)$ such that $A_i^p \preceq_e A_i^q$ for each $0 \leq i \leq n$. Then the following holds:*

$$p \preceq_e q.$$

Lemma 7. *There is a strategy function δ_t such that $\pi_p \preceq_e \delta_t(q_0, \pi_p)$ for each trace π_p starting in p_0.*

Proof. The proof would be exactly like the proof of Lemma 6, leveraging the existence of strategy functions for partial macrostates. □

Theorems 3, 5 that reduce state space in practice can be seen in Figure 4.2. To finish this section, we point out that since *early* and *early+1* simulations under-approximate language inclusion [8], if for two states $L(p) \subseteq L(q)$ and it was already decided that $L(q) = \emptyset$ we can conclude $L(p) = \emptyset$ without further exploration (which we utilize in Algorithm 3).

4.4.2 Reducing state space

Another optimization in terms of the early termination of inclusion (emptiness) check includes the *direct simulation* relation between the states of automaton A and automaton B. Intuitively, we want to prove that as soon as a product macrostate of $A \cap \overline{B}$ contains the state p from A and the state q from B such that $L(p) \subseteq L(q)$, there is no need to explore further from this macrostate, as it cannot produce any counterexample.

Theorem 6. *Consider the product macrostate $p = (p_A, p_C)$ such that $p_C = (N, P_0, \ldots, P_i)$ is a* KOFOLA *macrostate. Then we have*

$$(\exists q \in N \colon q \in Q_B \wedge L(p_A) \subseteq L(q)) \implies L(p) = \emptyset.$$

Proof. Suppose $q \in N$, if it holds that $L(p_A) \subseteq L(q)$, it means that q accepts every ω-word p_A does. Therefore, if there is an accepting trace from p_A over $u \in \Sigma^\omega$, there exists one from q too, meaning $u \notin L(p_C)$ (state of the complement cannot accept word that is accepted in the original automaton) which implies $u \notin L(p)$. The case when there is a non-accepting trace from p_A over $u \in \Sigma^\omega$ trivially leads to $u \notin L(p)$. Since a trace from p_A of any kind leads to non-acceptance, $L(p) = \emptyset$, which proves Theorem 6. □

Next, we present the reasoning for the special relations that can make the emptiness check more efficient.

Definition 6. *A relation $\sqsubseteq_{GFEE}$ is good for the early emptiness check (GFEE) if:*

$$p \sqsubseteq_{GFEE} q \iff (p_i \xrightarrow{a} p_{i+1} \in F \implies \exists j \leq i \colon q_j \xrightarrow{a} q_{j+1} \in F).$$

Theorem 7. *Let i be an initial state of BA A. Then we have the following.*

$$L(i) \neq \emptyset \implies \exists \pi_i \colon \neg\exists p, q \in Q \text{ and } p, q \in \pi_i \colon q \sqsubseteq_{GFEE} p \wedge p \rightsquigarrow q \tag{4.6}$$

Proof. For the sake of contradiction suppose that $L(i) \neq \emptyset$ and that $\forall \pi_i \cdot \exists p, q \in Q$ and $p, q \in \pi_i \cdot q \sqsubseteq_{GFEE} p \wedge p \rightsquigarrow q$. In this proof, we will show that this statement necessarily leads to existence of a trace that satisfies the right-hand side of the implication 4.6, which is clearly a contradiction. Firstly, let us label each such trace (of the potentially infinite number of traces) as π_x for $x \in \mathbb{N}$. Secondly, for each trace, there must exist a first pair of states p_x, q_x such that $p_x, q_x \in \pi_i$ and $q_x \sqsubseteq_{GFEE} p_x \wedge p_x \rightsquigarrow q_x$. For that purpose, we define the mapping $First \colon \pi \mapsto Q \times \mathbb{N} \times Q \times \mathbb{N}$. For trace π it returns $First(\pi) = (p + i, q + j)$ such that j is the minimum position for each $r, s \in Q$ on the trace π where $r \sqsubseteq_{GFEE} s \wedge s \rightsquigarrow r$. We often refer to i or j as $Pos(p)$ or $Pos(q)$ and only use $First(\pi) = (p, q)$. We also define a mapping $Prefix \colon \pi \times \mathbb{N} \mapsto Q^{\mathbb{N}}$ that returns a prefix of the trace π that forms a string of states with the length specified by the second argument.

Claim 1: For a trace π_x with $First(\pi_x) = (p_x + i, q_x + j)$ there exists another trace π_y such that $Prefix(\pi_x, Pos(p_x)) = Prefix(\pi_y, Pos(p_x))$.

Proof: That is implied by the fact that from p_x we need to see an accepting mark sooner that we do on the trace π_x (because $q_x \preceq_{di} p_x$), therefore the need for the other trace π_y. The existence of such a trace with the equality of the corresponding prefixes is trivial. ■

In the subsequent claims, we will show that existence of accepting trace π_x from Claim 1 necessarily leads to generating an accepting trace π_i such that $\neg\exists p, q \in Q$ and $p, q \in \pi_i \cdot q \sqsubseteq_{GFEE} p \wedge p \rightsquigarrow q$. In other words, it cannot generate traces π_j such that $\exists p, q \in Q$ and $p, q \in \pi_j \cdot q \sqsubseteq_{GFEE} p \wedge p \rightsquigarrow q$ infinitely often. Note that this trace can already exist; we only prove that it always exists. We also add the definition of function $Acc \colon \pi \times Q \times \mathbb{N} \mapsto \mathbb{N}$ returning position of the n-th accepting mark from q_x on the accepting trace we start with, on another trace from certain state.

Claim 2: Consider the traces π_x and π_y from Claim 1 and let $First(\pi_y) = (p_y + g, q_y + h)$. It holds that $Pos(p_x) < Pos(q_y)$ on trace π_y.

Proof: Suppose $Pos(p_x) \geq Pos(q_y)$ holds, from $Prefix(\pi_x, Pos(p_x)) = Prefix(\pi_y, Pos(p_x))$ it follows that $First(\pi_x) = (p_y, q_y)$, which contradicts $First(\pi_x) = (p_x, q_x)$. ■

Claim 3: Consider the traces π_x and π_y from Claim 2. It holds that if there is a state r with an outgoing accepting transition on the trace π_x such that $r \overset{\bullet}{\rightsquigarrow} p_x$, then the trace π_y contains the same prefix $Prefix(\pi_x, Pos(r)+1)$ (here we overload the function Pos, as it is contextually evident what we mean).
Proof: That is implied by the fact that $Prefix(\pi_x, Pos(p_x)) = Prefix(\pi_y, Pos(p_x))$, which is longer than or equal to $Prefix(\pi_x, Pos(r)+1)$. ■

Claim 4: Consider the traces π_x and π_y from Claim 2. It holds that $Acc(\pi_x, p_x, i) > Acc(\pi_y, p_x, i)$.
Proof: The $Acc(\pi_x, p_x, i) > Acc(\pi_y, p_x, i)$ is given by the fact that $|\, p_x, q_x \,| \geq 1$ (the length of path from p_x to q_x) and $Acc(\pi_y, p_x, i) \leq Acc(\pi_x, p_x, i) - |\, p_x, q_x \,| < Acc(\pi_x, p_x, i)$ (caused by $\sqsubseteq_{GFEE}$). ■

Claim 5: Consider the traces π_x and π_y from Claim 1. If a path from p_x to q_y contains $k \in \mathbb{N}$ accepting marks on the trace π_y, it holds that $Acc(\pi_x, q_x, k+1) > Acc(\pi_y, q_y, k+1)$.
Proof: Relation $\sqsubseteq_{GFEE}$ gives us $Acc(\pi_x, q_x, i) \geq Acc(\pi_y, p_x, i)$ for $i \geq k+1$, resulting in $Acc(\pi_x, p_x, i) > Acc(\pi_x, q_x, i) \geq Acc(\pi_y, p_x, i)$. From $Pos(p_x) < Pos(q_y)$ on π_y and the fact that there is only k accepting marks between p_x and q_y on π_y, we have $Acc(\pi_y, q_y, k+1) = Acc(\pi_y, p_x, k+1) - (Pos(q_y) - Pos(p_x))$, forming the inequality

$$Acc(\pi_x, p_x, k+1) > Acc(\pi_x, q_x, k+1) \geq Acc(\pi_y, p_x, k+1) > Acc(\pi_y, q_y, k+1).\blacksquare$$

Corollary 1. *According to Claim 5 from π_x, inductively there is a point where a trace π_m is generated such that $Acc(\pi_m, q_m, k+1) < 0$, which means $p_{m-1} \overset{\bullet}{\rightsquigarrow} p_m \rightsquigarrow q_m$.*

Corollary 2. *From Claim 3 and Corollary 1 it holds that traces $\pi_m, \ldots, \pi_{m1}$ such that $Acc(\pi_m, q_m, k+2) > \cdots > 0 > Acc(\pi_{m1}, q_{m1}, k+2)$ share the same prefix with the accepting mark.*

Inductively applying Corollary 2 at most $|\, Q \,|$ times, we obtain a trace π_{mQ} such that it contains $Prefix(\pi_m, Pos(r_m)+1), Prefix(\pi_{m1}, Pos(r_{m1})+1), \ldots, Prefix(\pi_{mQ}, Pos(r_{mQ})+1)$. Each prefix has the property of not containing any $First(p_{mi})$, and it contains an accepting mark. We thus certainly have a trace that reaches state t and after at most $|\, Q \,|$ transitions again t while seeing an accepting state. We give the desired trace, proving Theorem 7. □

A simple example showing the reduction of the state space using Theorem 7 with $\preceq_{di}$ being $\sqsubseteq_{GFEE}$ is shown in Figure 4.3.

Proposition 2. *Direct simulation $\sqsubseteq_{di}$ and early simulation $\preceq_e$ are $\sqsubseteq_{GFEE}$.*

Corollary 3. *Let i be an initial state of TGBA A, then:*

$$L(i) \neq \emptyset \implies \exists \pi_i \cdot \neg \exists p, q \in Q \text{ and } p, q \in \pi_i \colon q \sqsubseteq_{GFEE} p \wedge p \rightsquigarrow q.$$

Proof. Proof is given by the proof of Theorem 7, applying the same reasoning successively for each color $c \in \Gamma$. □

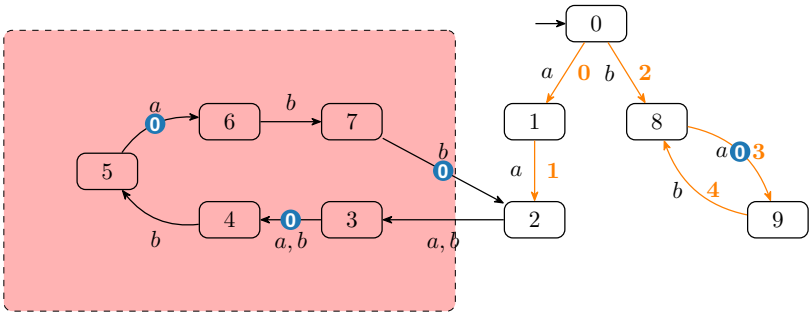


Figure 4.3: This figure shows exploring given BA A over a, b for emptiness check when utilizing Theorem 7. The orange transitions with orange numbers represent the DFS traversal and its order. And because $2 \preceq_{di} 0$ and $0 \rightsquigarrow 2$, we do not explore the red area from state 2.

Algorithm 2 Merge SCC

```
1: Function mergeSCCaccMarks(dst):
2:     cond ← ∅;
3:     do
4:         (u, C) ← SCCs.pop();
5:             if (not First(u)) or u is not Root then cond ← cond ∪ C;  ▷ To avoid acc. mark outside the SCC
6:             if cond = Γ then empty = false;decided = true;return true;
7:     while (u.dfsnum > dst.dfsnum);
8:     SCCs.push((u, cond));
9:     return false;
```

4.5 Inclusion check

For the algorithm that orchestrates the emptiness check for our inclusion, we chose to the best of our knowledge the best state-of-the-art algorithm for the emptiness check of the TGBA when it comes to number of generated states. The algorithm introduced by Gaiser and Schwoon [25] builds on the standard Tarjan's algorithm [37] to search strongly connected components, which is also well suited for on-the-fly checking. We provide an adaptation of this algorithm using our optimizations from previous sections. This algorithm is an amendment of Couvreur's algorithm [14], which is, to the best of our knowledge, implemented in Spot. There also exists an algorithm for the generic acceptance condition [3] (also implemented in Spot for some cases) that we do not use since it works with the SCCs being found completely, which is not suitable for our goal. If edited to work on-the-fly, so far no significant advantage has been found over the algorithm from [14].

The standard emptiness check algorithm is shown in Algorithms 2 and 3 without the lines inside the colored boxes. The lines in the orange boxes correspond to theorems in Subsection 4.4.1 and the green boxes correspond to Theorem 7. Theorem 6 is trivially implemented within the computation of the successors (necessary precomputation of *direct* simulation on a disjoint union of automata A and B [13]). The function `isEmpty` takes two arguments, a state and the set of accepting marks incoming to this state. We use the function `post` to obtain successors together with accepting marks on the incoming transitions.

(a) State q is not explored due to $q \preceq_e p$. The state between p and q stores the state q being pruned.

(b) The search continues as indicated by the numbers on the edges, eventually finding the accepting lasso.

Figure 4.4: Example of a wrong guess to prune a state when checking TGBA with $Acc = Inf(0) \wedge Inf(1)$.

4.5.1 Plugging relation $\sqsubseteq_{GFEE}$

In this section we provide the intuition of adjusting the Algorithm in [25] so that Theorem 7 can be applied. The idea behind it is to only search for the trace from Theorem 4.6, therefore each time we encounter states p, q on the *searchpath* [25] such that $p \rightsquigarrow q$ and $q \sqsubseteq_{GFEE} p$, we do not explore this state q. Note that we use *early* simulation as an instance of $\sqsubseteq_{GFEE}$ simulation. However, each state between such two states needs to have the information about the state q and the exploration from q being cut off; therefore, if some of the states between p and q is encountered again, exploration is "redirected" to the state that was cut off. Intuitively, this can be done, as we surely know that between p and q there was no accepting mark; if there was a cycle containing the accepting mark, the pruned state would have been explored already.

Proposition 3. *Algorithm 3 is correct.*

Figure 4.4 shows the situation when the guess of no need to explore the state q still leads to correctly finding the accepting lasso. To provide more details, after it is decided that $q \preceq_e p$ (green part in Line 15 of Algorithm 3), the state between p and q stores the information about q being cut off. The search then continues with a witnessing lasso containing the accepting mark (0), which means the existence of a path that violates $p \rightsquigarrow q \wedge p \preceq_e q$. Therefore, it is allowed to jump to q and continue the exploration from there.

The situation where such a guess actually helped is provided in Figure 4.3, although using a stronger *direct* simulation.

Algorithm 3 TGBA emptiness check

Input: TGBA $A = (Q, \delta, I, \Gamma, p, Acc)$
Output: true/false
Global: $empty = true$, $decided = false$, $emptyL = \emptyset \subseteq Q$, U $= -1$, index $= 0$, SCCs $= stack()$, tarjanStack $= stack()$

1: **Function** emptinessCheck(A):
2: **foreach** $q_I \in I$
3: **if** $q_I \notin Q_{emp}$ **then**
4: isEmpty($q_I, \emptyset$);
5: **return** $empty$;
6:
7: **Function** isEmpty($q, accMarks$):
8: **if** $\exists p$ on $searchpath$: $(p \overset{\leadsto}{\bullet} q \wedge p \preceq_e q) \vee (p \overset{\leadsto}{\bullet}\overset{\leadsto}{\bullet} q \wedge p \preceq_{e^{+1}} q)$ **then**
9: $empty = false$; $decided = true$;
10: **return**;
11: q.dfsnum $\leftarrow$ index; index++;
12: SCCs.push((q, $accMarks$)); tarjanStack.push(q);
13: **foreach** $(dst, marks) \in post(q)$ **do**
14: **if** $dst \in \lceil emptyL \rceil$ **then continue**; ▷ iff $\exists r \in emptyL$: $L(dst) \subseteq L(r)$
15: **else if** dst.dfsnum $=$ U **and** $\neg\exists r \in Q$ on $searchpath$: $dst \preceq_e r$ **then**
16: isEmpty(dst, $marks$);
17: **if** $decided$ **then return**;
18: **else if** dst.dfsnum $\neq$ U **then**
19: **if** $dst \in$ tarjanStack **and** mergeSCCaccMarks(dst) **then return**;
20: **foreach** cutoff $\in$ jumpToCutOffs[dst] **do**
21: **if** cutoff.dfsnum $=$ U **and** $\neg\exists r \in Q$ on $searchpath$: cutoff $\preceq_e r$ **then**
22: isEmpty(cutoff, $\emptyset$);
23: **if** $decided$ **then return**;
24: **else if** cutoff $\in$ tarjanStack **and** mergeSCCaccMarks(cutoff) **then return;**
25: **else if** $dst \in$ tarjanStack **and** mergeSCCaccMarks(dst) **then return**;
26:
27: **if** SCCs.top() $= (q, X)$ **then**
28: SCCs.pop()
29: **do**
30: $u \leftarrow$ tarjanStack.pop();
31: $emptyL$.add(u);
32: **while** $(u \neq q)$;
33: **return**;

Chapter 5

HyperLTL model checking as part of Kofola

Framework Kofola [26] is a command-line tool implemented in C++ on top of the Spot library [21] providing mainly complementation of Büchi automata. Spot is used for common automata manipulations such as reading input automaton, providing internal representation for automata, etc. The intriguing operations like *inclusion* and *emptiness check* are, with the goal of outperforming the state-of-the-art approaches in some metrics, implemented as the result of this thesis.

The implementation relevant to this thesis (extending Kofola) can be found in the public repository[1], which is a fork from repository[2]. In the spirit of Kofola's modularity, the HyperLTL model checking is also implemented in a modular way. More specifically, the emptiness check can be given any algorithm implementing desired methods, with getting successors being the most important one. This is convenient because we use the emptiness check in two different cases, (i) inclusion and (ii) emptiness of the automaton resulting from HyperLTL model checking, both on-the-fly. Moreover, it provides the potential for a straightforward extension when supporting new ω-automata types.

At first sight, the most easily recognizable difference compared to AutoHyper is the on-the-fly emptiness check when the formula is of the type $\exists^*\colon \varphi$. In addition, the inclusion that Spot (used by AutoHyper) uses first complements the automaton B and then makes the on-the-fly product (if possible). Our solution also makes use of Kofola being well-suited for an on-the-fly construction of the complement. Then we utilize the subsumptions introduced in the previous chapter. That can lead to a significant state-space reduction when on-the-fly emptiness check is performed.

Apart from the previous optimizations, comparing to theoretic approach for HyperLTL model checking, we also use the fact that a formula of the type $(\exists^*\forall^*)^*\varphi$ can be transformed into $\neg(\forall^*\exists^*)^*\neg\varphi$, which in practice makes model checking much faster. Next, the product automaton for the sequence of existence quantifiers is performed at once. Nevertheless, such observations are also implemented in AutoHyper, negating the possibility of a significant advantage.

[1] https://github.com/OndrejAlexaj/kofola/tree/inclusion-test
[2] https://github.com/VeriFIT/kofola/tree/devel

```
HOA: v1
States:  4
Start:  3
AP: 3 "h_0" "l_0" "o_0"
acc-name:  all
Acceptance:  0 t
properties:  state-labels explicit-labels
--BODY--
State:  [!0&!1&!2] 2
3
State:  [!0&!1&!2] 3
4
State:  [2&!0&!1] 4
5
State:  [2&!0&!1] 5
2
--END--
```

Figure 5.1: Behavior of a system in the `HOA` format. Each state of the system is specified between `--BODY--` and `--END--` lines, where the conjunction in the square brackets expresses which APs hold in the specific state. To the right there is the state specifier. Under the state there is a line with the successor states.

5.1 Input format

Our formats differ from those used in AutoHyper. We only support a specific input format for the specification of a system behavior, so the necessity for parsing different input formats is eliminated.

5.1.1 System

As input format for system behavior, we decided to use the `HOA` [19] format so that it can be easily parsed and stored by Spot as a Kripke structure. An example of such an input file is shown in Figure 5.1.

5.1.2 HyperLTL formula

For LTL body of the HyperLTL formula we support the exact format that Spot supports. However, each atomic proposition (AP) is of the format `{ap_sys}_{trace_var}` with `ap_sys` standing for the atomic proposition used within the system and `trace_var` stands for the quantified trace. The formula with quantifiers is then generated by the following syntax:

$$\varphi ::= (\texttt{(forall trace\_var.)}^* \ \texttt{(exists trace\_var.)}^*)^* \ \texttt{LTL}$$

$$\texttt{trace\_var} ::= \texttt{string}$$

An example of the GNI property for the system in Figure 5.1 is the following:

```
forall A. forall B. exists C.
(G ("{h_0}_{A}" <-> "{h_0}_{C}")) & (G("{l_0}_{B}" <-> "{l_0}_{C}"))
& (G("{o_0}_{B}" <-> "{o_0}_{C}"))
```

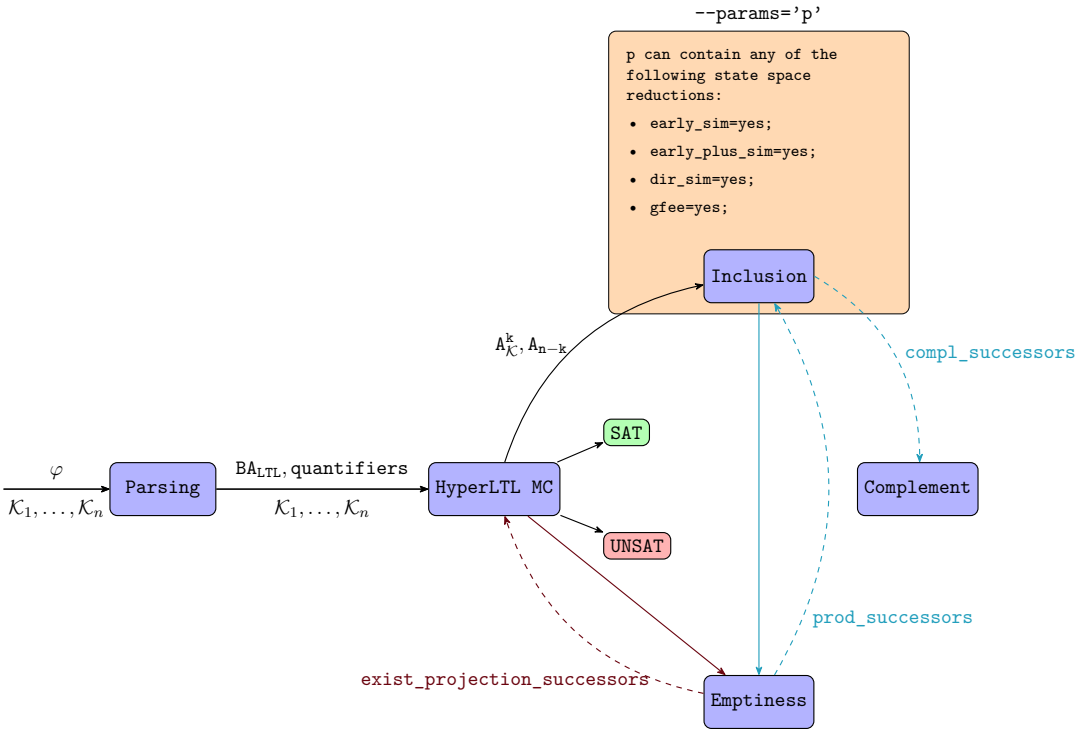


Figure 5.2: Overview of the architecture of HyperLTL model checking within KOFOLA, where dashed arrows represents only usage of the given component. Solid arrows mean handing over control to the other component. The orange frame summarizes the command line arguments regarding inclusion; none are used by default. Some arrows are colored differently to differentiate `Emptiness` in different uses.

5.2 Usage

The architecture of the HyperLTL model checking within KOFOLA is depicted in Figure 5.2 together with command line arguments, which can be used in the following way:

$$\texttt{kofola --hyperltl\_mc } \varphi\ \mathcal{K}_1 \ldots \mathcal{K}_n \texttt{ --params='p'}$$

where if more Kripke structures are provided, then $\mathcal{K}_i$[3] corresponds to the i-th quantified trace in the HyperLTL formula φ for $1 \leq i \leq n$.

5.2.1 Inclusion checker

One can use KOFOLA as an inclusion checker in the following way:

```
kofola --inclusion buchi_A.hoa buchi_B.hoa --params='p'
```

with both Büchi automata in the `HOA` [19] format.

[3] $\mathcal{K}_i$ represents a *file containing the i-th Kripke structure*, and φ represents a *file containing the HyperLTL formula*.

Chapter 6

Experimental evaluation

Firstly, we want to evaluate our inclusion check in terms of the generated state space against Spot [21] and show the effect of the proposed optimizations. Next, we compare our HyperLTL model checker with the state-of-the-art tool AutoHyper [4] together with other relevant inclusion checkers in terms of execution time. All experiments were performed on a Debian GNU/Linux 12 (bookworm) system with 32 GiB RAM and an Intel(R) 2.67GHz Xeon(R) X5650 CPU.

6.1 Kofola vs Spot

The key metric that we wanted to outperform Spot [21] in is the number of visited states. More specifically, we incremented a counter every time a state was put on the `tarjanStack`. Spot's inclusion checking approach is directly comparable to ours, since it

Table 6.1: Statistics for our experiments. The table shows a comparison of the state space generated by solving the inclusion. That is, it shows how individual optimizations behave and compares the approach utilizing all proposed optimizations (Kofola MAX) to Spot. The Kofola MAX no direct then refers to not employing direct simulation optimization, and the Kofola and GFEE refers to only utilizing Theorem 7. Kofola basic is then only the implementation of Algorithm from [25]. The column **solved** contains values separated by a colon, with the following meaning **(number of solved when inclusion is violated : number of solved when inclusion holds)**, where the number of cases where inclusion does not hold is 762 and 825 where it holds. Values in the columns **mean** and **median** are separated by the colon with the following meaning **(all test cases : inclusion violated : inclusion holds)**. The column **"wins"/"losses"** contains a number of cases where Kofola MAX produced strictly less/more states, where (number) means how many times it was due to the other's approach timeout. The column **TOs** (timeouts) shows how many times the approach could not decide the inclusion within 7 min.

tool	solved	mean	median	wins	losses	TOs
Kofola basic	720 : 774	540 : 51 : 995	31 : 13 : 99.5	620 (0)	11 (11)	93
Kofola and early(+1)	719 : 768	162 : 49 : 268	29 : 13 : 73	637 (8)	23 (20)	100
Kofola and direct	700 : 780	379 : 48 : 676	14 : 13 : 21.5	420 (0)	5 (5)	107
Kofola and GFEE	718 : 775	507 : 50 : 932	31 : 13 : 96	769 (5)	24 (24)	94
Kofola MAX no direct	718 : 768	160 : 48 : 264	28 : 13 : 73	498 (8)	20 (19)	101
Kofola MAX	700 : 775	89 : 46 : 127	14 : 13 : 17	-	-	112
Spot	744 : 819	21,478 : 21,896 : 21,098	41 : 40.5 : 41	988 (1)	495 (89)	24

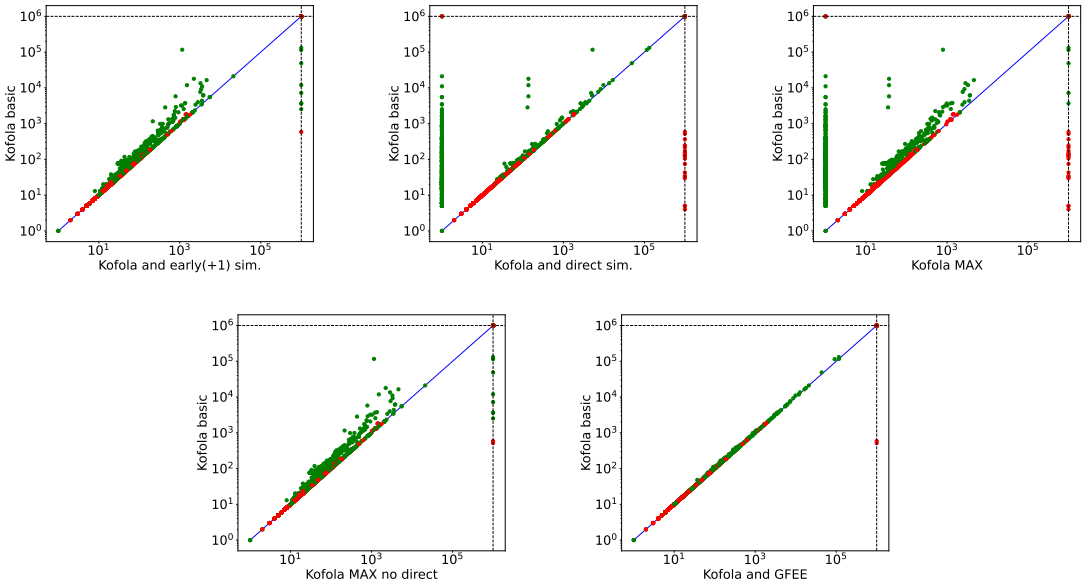


Figure 6.1: The scatter plots compare the state space generated by implementation of algorithm from [25] provided by Gaiser & Schwoon , referred to as *Kofola basic*, against its extensions by various optimizations proposed in this work. The dashed lines represent the timeouts, which was set to 7 min. Green marks indicate instances where inclusion holds, and red marks otherwise. As the axes are logarihtmic, cases with 0 generated states are represented as 1 (10^0).

uses a similar orchestrating algorithm [15] (based on Tarjan's algorithm [37]). More specifically, we compared our approach with Spot's command line tool `autfilt`, which provides the `--included-in` parameter for inclusion checking. In addition to comparison, `autfilt` was used to test the correctness of our inclusion checking implementation.

6.1.1 Dataset

To observe the effect of the proposed optimizations, we tested Kofola in the scenarios where the automaton B in the question of $A \subseteq B$ contains deterministic accepting components (DACs) or inherently weak accepting components. More specifically, we used automata used in [8] from [32] and benchmarks originating from HyperLTL model checking [33]. Both repositories contain pairs of automata (suffix `A.hoa` and `B.hoa`) with the mentioned properties. And since both automata in such a pair have the same alphabet, to obtain more test cases, all four combinations (i.e. $A \overset{?}{\subseteq} B$, $A \overset{?}{\subseteq} A$, $B \overset{?}{\subseteq} B$, and $B \overset{?}{\subseteq} A$) were tried and kept those whose corresponding complementation by Kofola produced a TGBA. The total number of test cases is 1,587. The mean number of automata states from repositories is 891, the median is 14 states, the maximum number of states is 88,304, and the minimum is 1.

6.1.2 Results

The results shown in Figure 6.1 show a great (and expected) impact of precomputation of the *direct simulation* between the states of automata A and B. The huge amount of

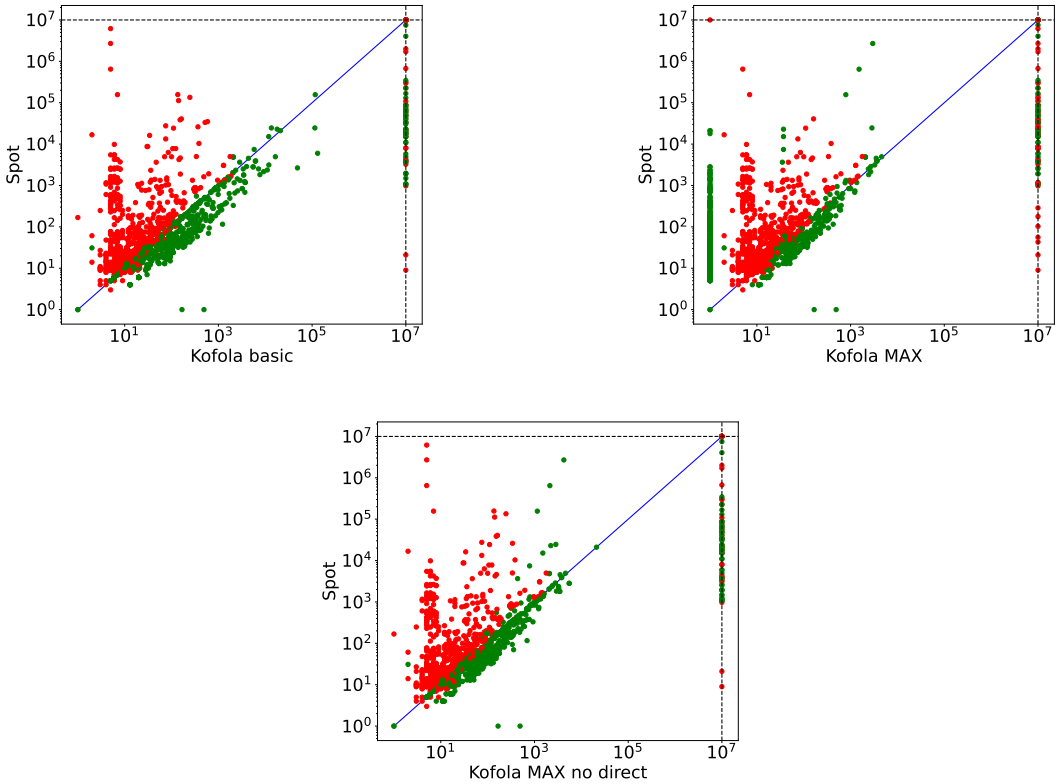


Figure 6.2: Here we provide comparison of the explored state space by Spot [21] against our approaches. The dashed lines represent the timeouts, which was set to 7 min. Green marks indicate instances, where the inclusion holds, red marks otherwise. As the axes are logarihtmic, cases with 0 generated states are represented as 1 (10^0).

instances where we produced zero states is also not surprising, since if the language inclusion holds between the initial states of the automata A, B (implied by the *direct* simulation), then it holds between the automata as well (which is expected since many test cases are of type where $A = B$). We can state that our optimizations seem to significantly reduce the state space when the inclusion holds; the opposite case is only slightly better when utilizing our optimizations (see also Table 6.1). In Figure 6.1 it may seem that there is not much of an impact of the optimization *GFEE* (based on Theorem 7, green parts in Algorithm 3), but Table 6.1 shows some improvement especially when the inclusion holds (as expected). Figure 6.2 provides a comparison of our approach with Spot. We can see that we improved the cases where the basic implementation of the emptiness check algorithm loses against Spot the most - when the inclusion holds. Although the explored state space size is in our favor, we timed out visibly more than Spot. Detailed statistical evaluation for summarization is provided in Table 6.1. It shows that we are able to significantly (in almost 40% of the benchmarks) reduce the state space explored by the basic implementation of the algorithm from Gaiser & Schwoon [25], moreover in more than 60% of the test cases we generated a smaller state space than Spot. Although our maximally optimized approach shows significantly better numbers, we have to bear in mind that those are the numbers taken when the tool did not time out. From the **TOs** column in Table 6.1 it is clear that our most optimized procedure timed out in 88 more cases than Spot. If we compare the

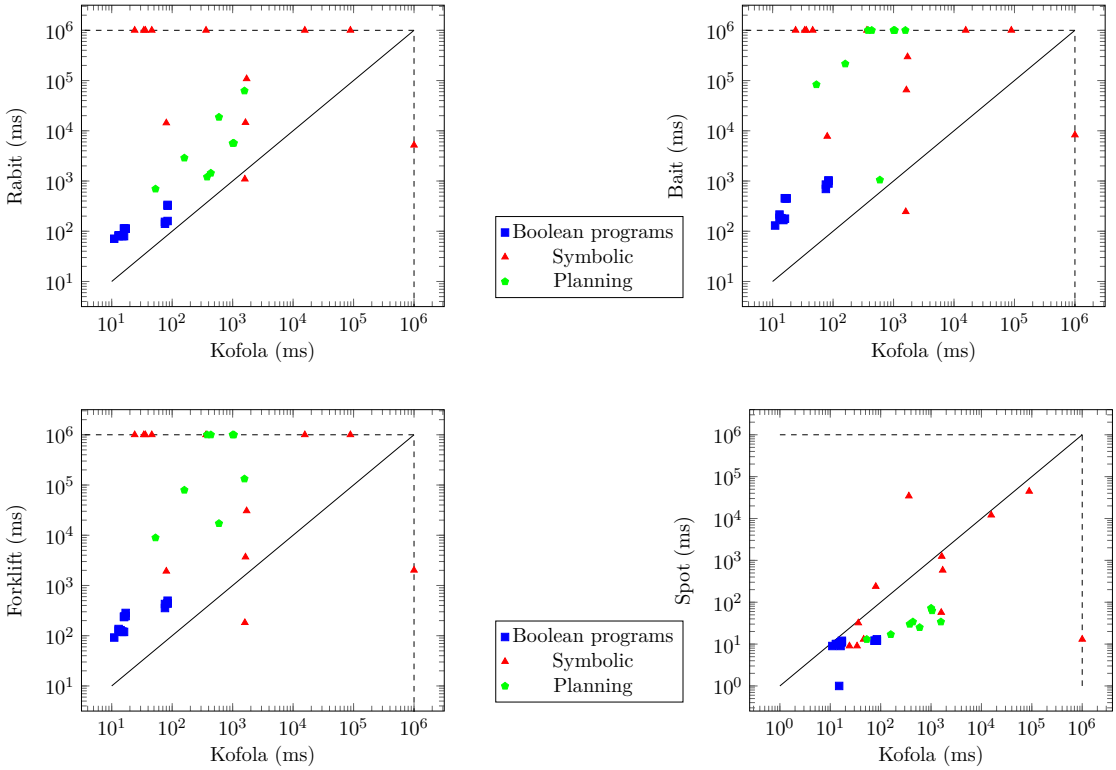


Figure 6.3: Comparison of execution times for KOFOLA's inclusion checking (all optimizations used) used as a backend checker for AUTOHYPER vs other state-of-the-art tools. The dashed lines represent timeout, which was set to 10 minutes.

cases where both tools successfully solved inclusion, we obtain **median** value 37 for SPOT and 14 for KOFOLA MAX, and **mean** value 3340 for SPOT and 89 for KOFOLA MAX, which is still significantly better. Here, we have to point out that there is an undeniable effect of the KOFOLA's complementation. Next, in Table 6.1 we can observe that in terms of solved problems, we have slightly better success rate (no timeout) in cases where the inclusion holds, but still it is roughly only 94% versus the 99% success rate of SPOT.

Lastly, we separately tested our procedure that utilizes all optimizations except for the precomputation of the *direct simulation*, since for large automata, only this operation itself causes timeout. This can be seen in both Figures 6.1 and 6.2 and also in a separate row in Table 6.1, referred to as KOFOLA MAX NO DIRECT. We can conclude that although the number of timeouts decreased by 11 (compared to the maximally optimized approach), the mean of the generated state space is approximately twice as large.

6.2 Kofola vs AutoHyper

In Figure 6.3 we can see a comparison between inclusion checkers RABIT [12], BAIT [18], FORKLIFT [17], SPOT [21], and KOFOLA. More specifically, these are the execution times that the inclusion checkers spent when solving the HyperLTL model checking within AUTOHYPER [4] (KOFOLA was also used by AUTOHYPER as a backend solver in order for the results to be fair), when solving the exact same 35 benchmarks as in [4] (actually it was 36, but in one case inclusion was not used). From the results in Figure 6.3 we can

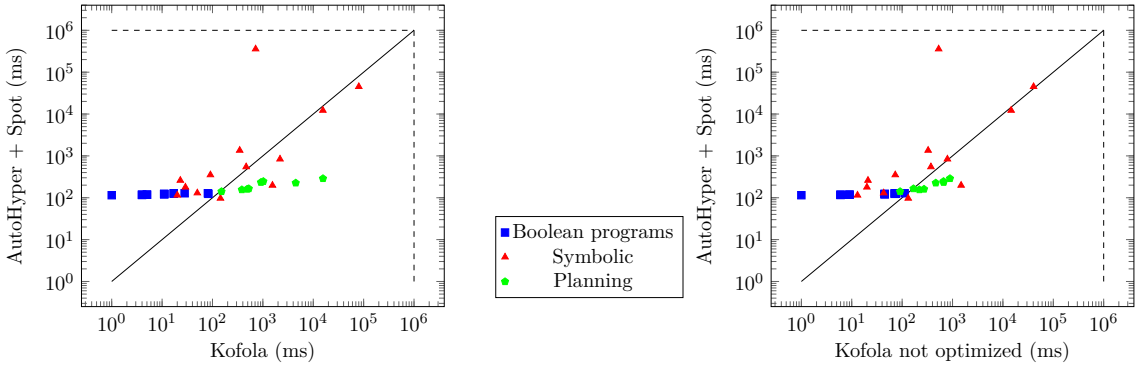


Figure 6.4: The scatter plot compares the execution time of KOFOLA and AUTOHYPER (with SPOT as inclusion checker). Here, no mark lies on the dashed line, therefore there were no timeouts (10 minutes). In the left hand side plot KOFOLA with maximally optimized inclusion checking is tested and in the right hand side no optimizations are used.

conclude that we highly outperform RABIT, BAIT, and FORKLIFT. However, there is a case where our tool timed out. This was due to the precomputation of direct simulation in our approach. When disabled, we again outperformed the tools. When we look at the plot compared to SPOT, we managed to beat it twice, which might be surprising given that our implementation is not optimized for this metric.

In Figure 6.4 we provide a comparison of the time it takes to solve 36 instances also tested in [4]. We solved the model checking problem faster in 25 out of 36 test cases. As witnessed before, our inclusion checking is slower than the one implemented in SPOT, therefore, our win rate could be caused by the fact that we use SPOT's highly optimized internal representation for ω-automata and their attributes. After using more optimized inclusion checking, the KOFOLA's model checking procedure tends to slow down (non-optimized is faster in 27 out of 36 test cases against AUTOHYPER), which is expected unless their implementation is optimized as well.

AUTOHYPER also served for checking correctness of our HyperLTL MC implementation.

Chapter 7

Conclusion

This thesis presents several optimizations regarding the language inclusion check on ω-automata (more specifically, we focus on TGBA), one of the most crucial operations used not only in HyperLTL model checking. To optimize this operation, we set our goal to reduce the generated state space. To do so, we came up with the relations between states of the automaton that leverage only the structure of the states themselves, without the need to know the entire automaton. We tried to come up with not only a reporting counterexample as early as we can, but also techniques to reduce state space when the resulting language is empty.

As an inclusion checker, our tool is able to improve the existing algorithm in the generated state space and report better results than the other similarly working state-of-the-art tools. Our procedure was able to outperform the reference tool in more than 60% benchmark cases, and in almost 40% of the benchmarks, we were able to generate a smaller state space than the algorithm we used as a base for our inclusion check. Other relevant inclusion checking tools are shown to be slower on the inclusion problems from HyperLTL model checking than us. Finally, as HyperLTL model checker, we are also able to outperform the state-of-the-art push button tool in the execution time.

7.1 Future work

When it comes to the inclusion check introduced in this thesis, the implementation was not meant to be performance-optimized; therefore, there is plenty of room for improvement in this area.

One of the future directions is definitely to extend Theorem 7, as we believe it has more to offer. As Kofola is planned to be able to output ω-automata with generic acceptance condition as a result of complementation. Another way can be to come up with an optimized version of inclusion check for these generic automata. In addition, there are more partial complementation procedures implemented within Kofola, so bringing up a similar subsumption relations as we worked with here is also a sensible continuation. It seems that inclusion is the right direction to take to improve HyperLTL model checking, since this operation is often encountered and is the source of interesting inclusion problems.

Bibliography

[1] Allred, J. D. and Ultes Nitsche, U. A Simple and Optimal Complementation Algorithm for Büchi Automata. In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. New York, NY, USA: Association for Computing Machinery, 2018, p. 46–55. LICS '18. ISBN 9781450355834. Available at: https://doi.org/10.1145/3209108.3209138.

[2] Andrés, M. E. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems*. 2011.

[3] Baier, C.; Blahoudek, F.; Duret Lutz, A.; Klein, J.; Müller, D. et al. Generic Emptiness Check for Fun and Profit. In: Chen, Y.-F.; Cheng, C.-H. and Esparza, J., ed. *Automated Technology for Verification and Analysis*. Cham: Springer International Publishing, 2019, p. 445–461. ISBN 978-3-030-31784-3.

[4] Beutner, R. and Finkbeiner, B. *AutoHyper: Explicit-State Model Checking for HyperLTL*. 2023.

[5] Beutner, R. and Finkbeiner, B. Model Checking Omega-Regular Hyperproperties with AutoHyperQ. In: Piskac, R. and Voronkov, A., ed. *Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. EasyChair, 2023, vol. 94, p. 23–35. EPiC Series in Computing. ISSN 2398-7340. Available at: https://easychair.org/publications/paper/d1VW.

[6] Blahoudek, F.; Heizmann, M.; Schewe, S.; Strejček, J. and Tsai, M.-H. Complementing Semi-deterministic Büchi Automata. In: Chechik, M. and Raskin, J.-F., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, p. 770–787. ISBN 978-3-662-49674-9.

[7] Büchi, J. R. *On a decision method in restricted second order arithmetic, Logic, Methodology and Philosophy of Science (Proc. 1960 Internat. Congr.)*. Stanford Univ. Press, Stanford, Calif, 1962.

[8] Chen, Y.-F.; Heizmann, M.; Lengál, O.; Li, Y.; Tsai, M.-H. et al. Advanced automata-based algorithms for program termination checking. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2018, p. 135–150. ISBN 9781450356985. Available at: https://doi.org/10.1145/3192366.3192405.

[9] Clarkson, M. R.; Finkbeiner, B.; Koleini, M.; Micinski, K. K.; Rabe, M. N. et al. Temporal Logics for Hyperproperties. In: Abadi, M. and Kremer, S.,

ed. *Principles of Security and Trust*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, p. 265–284. ISBN 978-3-642-54792-8.

[10] Clarkson, M. R. and Schneider, F. B. Hyperproperties. In: *2008 21st IEEE Computer Security Foundations Symposium*. 2008, p. 51–65.

[11] Clemente, L. and Mayr, R. Advanced Automata Minimization. *CoRR*, 2012, abs/1210.6624. Available at: http://arxiv.org/abs/1210.6624.

[12] Clemente, L. and Mayr, R. Efficient reduction of nondeterministic automata with application to language inclusion testing. *CoRR*, 2017, abs/1711.09946. Available at: http://arxiv.org/abs/1711.09946.

[13] Clemente, L. and Mayr, R. Efficient reduction of nondeterministic automata with application to language inclusion testing. *CoRR*, 2017, abs/1711.09946. Available at: http://arxiv.org/abs/1711.09946.

[14] Couvreur, J.-M. On-the-fly Verification of Linear Temporal Logic. In: Wing, J. M.; Woodcock, J. and Davies, J., ed. *FM'99 — Formal Methods*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, p. 253–271. ISBN 978-3-540-48119-5.

[15] Couvreur, J.-M. On-the-Fly Verification of Linear Temporal Logic. In:. September 1999, p. 253–271. ISBN 978-3-540-66587-8.

[16] D'Argenio, P. R.; Barthe, G.; Biewer, S.; Finkbeiner, B. and Hermanns, H. Is Your Software on Dope? In: Yang, H., ed. *Programming Languages and Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, p. 83–110. ISBN 978-3-662-54434-1.

[17] Doveri, K.; Ganty, P. and Mazzocchi, N. FORQ-Based Language Inclusion Formal Testing. In: Shoham, S. and Vizel, Y., ed. *Computer Aided Verification*. Cham: Springer International Publishing, 2022, p. 109–129. ISBN 978-3-031-13188-2.

[18] Doveri, K.; Ganty, P.; Parolini, F. and Ranzato, F. Inclusion Testing of Büchi Automata Based on Well-Quasiorders. In: Haddad, S. and Varacca, D., ed. *32nd International Conference on Concurrency Theory (CONCUR 2021)*. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, vol. 203, p. 3:1–3:22. Leibniz International Proceedings in Informatics (LIPIcs). ISBN 978-3-95977-203-7. Available at: https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.CONCUR.2021.3.

[19] Duret Lutz, A.; Blahoudek, F.; Křetínský, J.; Strejček, J. and Klein, J. *The Hanoi Omega-Automata Format*. February 2015. Available at: https://adl.github.io/hoaf/#authors.

[20] Duret Lutz, A.; Poitrenaud, D. and Couvreur, J.-M. On-the-fly Emptiness Check of Transition-Based Streett Automata. In: Liu, Z. and Ravn, A. P., ed. *Automated Technology for Verification and Analysis*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, p. 213–227. ISBN 978-3-642-04761-9.

[21] Duret Lutz, A.; Renault, E.; Colange, M.; Renkin, F.; Aisse, A. G. et al. From Spot 2.0 to Spot 2.10: What's New? In: *Proceedings of the 34th International*

Conference on Computer Aided Verification (CAV'22). Springer, August 2022, vol. 13372, p. 174–187. Lecture Notes in Computer Science.

[22] Finkbeiner, B. *Automata, Games, and Verification*. Saarland University, 2015. Available at: https://www.react.uni-saarland.de/teaching/automata-games-verification-15/downloads/notes.pdf.

[23] Finkbeiner, B. Logics and Algorithms for Hyperproperties. *ACM SIGLOG News*. New York, NY, USA: Association for Computing Machinery, jul 2023, vol. 10, no. 2, p. 4–23. Available at: https://doi.org/10.1145/3610392.3610394.

[24] Finkbeiner, B.; Rabe, M. N. and Sánchez, C. Algorithms for Model Checking HyperLTL and HyperCTL*. In: Kroening, D. and Păsăreanu, C. S., ed. *Computer Aided Verification*. Cham: Springer International Publishing, 2015, p. 30–48. ISBN 978-3-319-21690-4.

[25] Gaiser, A. and Schwoon, S. Comparison of Algorithms for Checking Emptiness on Buechi Automata. *CoRR*, 2009, abs/0910.3766. Available at: http://arxiv.org/abs/0910.3766.

[26] Havlena, V.; Lengál, O.; Li, Y.; Šmahlíková, B. and Turrini, A. Modular Mix-and-Match Complementation of Büchi Automata. In: Sankaranarayanan, S. and Sharygina, N., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer Nature Switzerland, 2023, p. 249–270. ISBN 978-3-031-30823-9. Available at: https://doi.org/10.1007/978-3-031-30823-9_13.

[27] Havlena, V.; Lengál, O. and Šmahlíková, B. Sky Is Not the Limit. In: Fisman, D. and Rosu, G., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2022, p. 118–136. ISBN 978-3-030-99527-0.

[28] Hsu, T.-H.; Sánchez, C. and Bonakdarpour, B. Bounded Model Checking for Hyperproperties. In: Groote, J. F. and Larsen, K. G., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2021, p. 94–112. ISBN 978-3-030-72016-2.

[29] Kähler, D. and Wilke, T. Complementation, Disambiguation, and Determinization of Büchi Automata Unified. In: Aceto, L.; Damgård, I.; Goldberg, L. A.; Halldórsson, M. M.; Ingólfsdóttir, A. et al., ed. *Automata, Languages and Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, p. 724–735. ISBN 978-3-540-70575-8.

[30] Kauffman, S.; Havelund, K. and Fischmeister, S. What can we monitor over unreliable channels? *International Journal on Software Tools for Technology Transfer*, Aug 2021, vol. 23, no. 4, p. 579–600. ISSN 1433-2787. Available at: https://doi.org/10.1007/s10009-021-00625-z.

[31] Kupferman, O. and Vardi, M. Y. Weak alternating automata are not that weak. *ACM Trans. Comput. Logic*. New York, NY, USA: Association for Computing Machinery, jul 2001, vol. 2, no. 3, p. 408–429. ISSN 1529-3785. Available at: https://doi.org/10.1145/377978.377993.

[32] Lengál, Ondřej. *Automata-benchmarks* https://github.com/ondrik/automata-benchmarks/tree/master/omega/advanced-automata-for-termination/hoa. GitHub, 2024.

[33] Lengál, Ondřej. *Automata-benchmarks* https://github.com/ondrik/automata-benchmarks/tree/master/omega/autohyper. GitHub, 2024.

[34] McCullough, D. Noninterference and the composability of security properties. In: *Proceedings. 1988 IEEE Symposium on Security and Privacy*. 1988, p. 177–186.

[35] Miyano, S. and Hayashi, T. Alternating finite automata on ω-words. *Theoretical Computer Science*, 1984, vol. 32, no. 3, p. 321–330. ISSN 0304-3975. Available at: https://www.sciencedirect.com/science/article/pii/0304397584900495.

[36] Safra, S. On the complexity of omega -automata. In: *[Proceedings 1988] 29th Annual Symposium on Foundations of Computer Science*. 1988, p. 319–327.

[37] Tarjan, R. Depth-first search and linear graph algorithms. In: *12th Annual Symposium on Switching and Automata Theory (swat 1971)*. 1971, p. 114–121.

[38] Viswanathan, M. *Automata on Infinite Words*. 2018. Available at: https://courses.engr.illinois.edu/cs498mv/fa2018/wAutomata.pdf.

Appendix A

Contents of the included storage media

Everything needed to create this text (electronic version is also included) can be found in the directory `text_resources` (`.tex` files, figures, ...). The implementation of the tool Kofola can be found in the directory `kofola`, which contains a `README.md` with instructions on how to build and run an executable file. This thesis extends the functionality of Kofola, therefore, the source codes of the whole tool are included; the actual work of this thesis is implemented in files in the `src` directory shown in Figure A.1 (although necessary changes were also made in `main.cpp`, `complement_sync.{c,h}pp` and `abstract_complement_alg.{c,h}pp`).

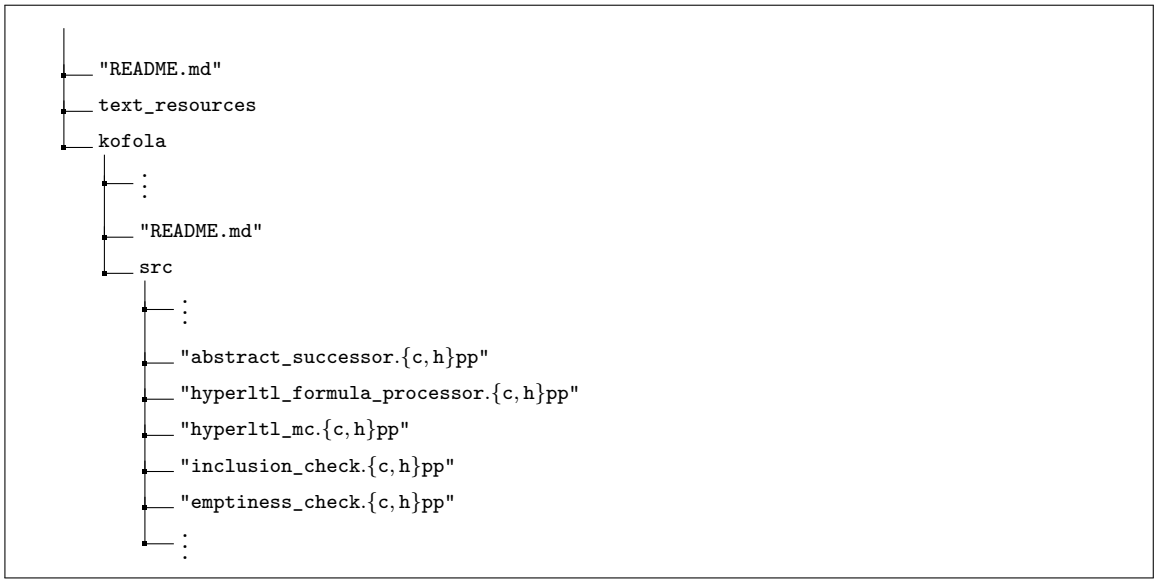


Figure A.1: Contents of the included storage media as a directory tree that also shows important files (these are marked with a "" to distinguish them from directories).