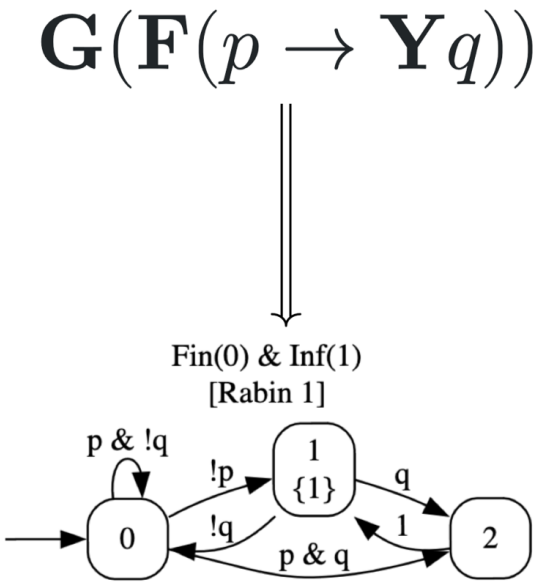
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Implementing and Evaluating pLTL to Deterministic Rabin Automata Conversion

Master's Thesis in Computer science and engineering

Fangsong Long

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's Thesis 2025

Implementing and Evaluating pLTL to Deterministic Rabin Automata Conversion

Fangsong Long

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Implementing and Evaluating pLTL to Deterministic Rabin Automata Conversion
Fangsong Long

Supervisor: David Lidell, Department of Computer Science and Engineering
Examiner: Robert Feldt, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Fangsong Long
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Linear Temporal Logic with Past (pLTL) extends standard LTL by introducing past-time operators, enabling more concise specifications of system properties. While theoretically advantageous, applying pLTL in practical verification tasks requires converting its formulas into ω-automata, a process that remains computationally challenging. A recent algorithm proposed by Azzopardi et al.[1] provides a direct construction of deterministic Rabin automata from pLTL formulas, yet no functional implementation of this algorithm currently exists. Without such an implementation, researchers and practitioners lack validation of its feasibility, efficiency, and applicability.

This thesis bridges this gap by developing the first working implementation of Azzopardi et al.'s algorithm [1] and systematically evaluating its performance. The study benchmarks the implementation against existing methods that indirectly achieve the same transformation via Büchi automata and determinization tools. We measured construction time and memory usage across both real-world and synthetic pLTL formulas. The results assess the algorithm's scalability and check whether it offers practical advantages over traditional approaches.

Findings from this research contribute to the field of formal methods by providing the first working implementation of the direct pLTL-to-deterministic automata conversion algorithm and offering insight into its performance. It is also the only known tool capable of converting certain types of pLTL formulas with more than 15 temporal operators to ω-automata within reasonable time and memory constraints, a feat previous tools could not achieve. Practitioners could use the produced tool for faster model checking and reactive synthesis, enabling more effective use of pLTL in real-world applications.

Keywords: formal methods, Linear Temporal Logic, automata theory

Acknowledgements

I would like to express my gratitude to the following individuals for their participation in this project.

Shaun Azzopardi, David Lidell & Nir Piterman
The entirety of the implementation and evaluation work is based upon their excellent work, which provided the foundation for the algorithm.

Additional gratitude is extended to *David Lidell*, who served as my supervisor. He presented the opportunity to undertake this thesis and provided guidance throughout the progress of the work.

Robert Feldt
For his valuable suggestions as an examiner throughout the entire process of writing this thesis.

Fangsong Long, Gothenburg, June 2025

Contents

List of Figures

List of Tables

1

Introduction

Linear Temporal Logic (LTL), introduced by Pnueli in 1977 [23], is a fundamental framework in formal methods. It's widely used to specify and analyze temporal properties of systems, particularly in concurrent and distributed environments. By enabling reasoning about sequences of events over time, LTL plays a crucial role in verifying system behaviors and ensuring correctness in complex computing systems.

To apply LTL in practice, formulas are commonly converted into an ω-automaton [27, 12], a type of automaton designed to process infinite sequences. This transformation enables various tasks, such as verifying system properties [16] or generating systems that satisfy LTL specifications [22].

An extension of LTL, known as LTL with past (pLTL) [20], introduces past-time operators like "Yesterday" and "Since." While pLTL is equally expressive as standard LTL, it is significantly more succinct [20] in some cases. However, this added expressiveness introduces additional challenges when converting pLTL formulas into automata. This is because the implementation may need a way to remember past events to evaluate the formula, whereas standard LTL focuses only on the future.

A recent research [1] proposed a new method to convert pLTL into deterministic Rabin automata. This thesis is aimed for implement and benchmark it.

1.1 Problem Description

A recent algorithm proposed in [1] offers a method to convert pLTL into deterministic Rabin automata. However, the lack of a functional implementation prevents researchers and practitioners from testing its feasibility, assessing its performance, or integrating it into existing verification workflows. Without an implementation, the algorithm remains purely theoretical, limiting its impact on advancing practical tools for temporal logic analysis.

Additionally, the absence of comparative studies between this algorithm and existing tools hinders a comprehensive understanding of its strengths, limitations, and real-world applicability. This gap in the literature restricts the adoption of pLTL in system specification and verification, despite its potential benefits in usability.

To address this gap, this research aims to develop a working implementation of the algorithm proposed in [1]. This implementation will allow for rigorous performance evaluations through controlled experiments, comparing it against existing tools. The results will provide valuable insights into the algorithm's efficiency and practicality, contributing to the broader field of system verification and enhancing the real-world applicability of pLTL.

1.2 Purpose of the Study

The purpose of the study includes:

- Developing a functional implementation of the algorithm proposed in [1] for converting pLTL into deterministic Rabin automata and implement necessary optimizations.
- Evaluating the performance of the implementation by comparing it against existing tools.
- Providing practical insights into the scalability, usability, and efficiency of the algorithm.

1.3 Significance of the Study

The challenge of converting pLTL into deterministic Rabin automata remains largely theoretical due to the lack of a functional implementation of the latest proposed algorithm. This gap limits our understanding of the algorithm's practical feasibility, efficiency, and scalability. While its theoretical properties have been established, critical questions remain unanswered: How does it perform under real-world conditions? How does its computational complexity compare to existing tools? How do variations in problem parameters affect its behavior? Without empirical validation, researchers and practitioners lack the necessary insights to determine whether this algorithm can enhance system verification workflows.

This study directly addresses these challenges by developing the first working implementation of the algorithm and systematically evaluating its performance. It investigates key aspects such as its efficiency relative to existing tools, trade-offs in automaton size and complexity, and scalability with increasingly complex pLTL specifications. Through controlled experiments, this research provides empirical benchmarks that were previously unavailable, offering a clearer understanding of the algorithm's strengths and limitations.

Beyond its theoretical contributions, this study has significant practical implications for software engineering and formal verification. Many real-world applications, such as model checking for distributed systems, verification of safety-critical software, and reactive synthesis, rely on efficient automaton-based reasoning. If the proposed algorithm proves computationally viable, it could improve the practicality of pLTL-based specifications, enabling more robust system verification techniques with reduced computational overhead. This advancement could benefit industries such as robotics, aerospace, automotive, and cybersecurity, where formal methods play a crucial role in ensuring system correctness.

By bridging the gap between theory and implementation, this research contributes to both the advancement of temporal logic methodologies and the enhancement of real-world software verification practices.

1.4 Research question and Hypotheses

RQ 1: How efficient is the implementation of [1]'s algorithm compared to existing tools?

To assess its efficiency, we will measure performance metrics, including execution time and memory consumption of the implementation. We will analyze the runtime efficiency, memory usage, and scalability of the implemented algorithm under controlled experimental conditions. This comparison aims to determine whether the new implementation is competitive with or superior to existing approaches.

Hypothesis 1: The implemented tool performs at least as efficiently as existing tools in terms of runtime and memory usage when handling complex pLTL specifications.

RQ 2: How well does the implementation scale with the size of pLTL formulas?

To evaluate scalability, the time and space required to convert a pLTL formula to automata will be measured. This will be done with an increasing size of the formula until the largest pLTL formulas that the implementation can process within a reasonable time and memory limit are identified. This analysis will help determine the tool's practical usability for increasingly complex specifications.

1.5 Overview of the Results

The implementation demonstrates performance and memory consumption comparable to existing tools for most small-sized real-world pLTL formulas. Notably, for certain larger pLTL formulas, this tool represents the sole known method for achieving the conversion.

Despite the algorithm's double-exponential complexity, we achieved strong overall scalability by leveraging its inherent parallelization. Our benchmarks demonstrate that the tool can handle real-world pLTL formulas with up to 12 to 18 temporal operators and 4 to 6 atomic propositions. This is a significant improvement, as current tools often cannot even convert such formulas to an undeterministic ω-automata. Examples like benchmark cases A.22 and A.23 in the appendix illustrate this capability.

In conclusion, this tool offers a relatively effective and often the first known feasible solution for converting pLTL formulas.

1.6 Thesis Outline

The thesis is structured as follows:

Chapter 2 introduces the related work, covering the history of research on (p)LTL and ω-automata.

Chapter 3 presents the theoretical concepts used in this thesis, which are essential for understanding both the algorithm and our implementation.

Chapter 4 outlines the implementation and evaluation methods, including the features we implemented, the data we collected, and the approach we used to analyze the results.

Chapter 5 discusses the results of evaluating and comparing the implemented tool with other existing tools.

Chapter 6 provides the conclusions drawn from the study.

2

Related Work

2.1 ω-automata

In 1961, Büchi formally introduced the Büchi automaton, which accepts infinite sequences [21]. This marked the beginning of the theory of ω-automata, establishing the foundation for reasoning about properties of infinite execution paths. In the late 1960s, Büchi and Landweber applied ω-automata to solve the reactive system synthesis problem proposed by Church, thus paving the way for the integration of infinite game theory and automata theory [3].

2.2 Linear temporal logic(LTL)

Amir Pnueli [23] introduced LTL as a framework for reasoning about temporal properties, which has since become foundational in system verification.

2.3 LTL and ω-automata

In 1986, Vardi, Wolper, and colleagues developed the automaton-based method for LTL verification: converting an LTL formula into an equivalent Büchi automaton [29], performing a Cartesian product with the state space of the system, and then checking for a violating path within the resulting automaton to determine if the property is violated. These works established the paradigm of reducing the logic verification problem to a language acceptance problem involving ω-automata.

In 1995, Holzmann et al. introduced a conversion tool for transforming LTL formulas into Büchi automata [17], allowing users to directly use LTL formulas as specifications.

In 2001, Gastin and Oddoux proposed a more efficient algorithm, ltl2ba [12], for converting LTL to Büchi automata, which significantly improved the performance of practical tools.

2.4 LTL and ω-automata in Formal Methods

Pnueli and Lichtenstein proposed the first LTL model checking algorithm in 1985, based on a representation construction (tableau technique). While this algorithm is exponential in time with respect to the size of the formula, it is linear in time with respect to the size of the model (state space) [18].

The construction of the complement of general non-deterministic Büchi automata is highly complex, and efficient methods have long been lacking. In 1988, Safra introduced the famous Safra decision algorithm, which can convert any non-deterministic Büchi automaton into an equivalent deterministic Rabin automaton.

Although this algorithm is complex [26], it provides a feasible path from non-deterministic to deterministic ω-automata, enabling complete operations on ω-automata, such as complement and inclusion checks. In the same year, Rosner and Roni applied LTL to reactive module synthesis, proposing a framework for automatically synthesizing system modules that satisfy specifications derived from LTL [25]. These advancements expanded the application areas of ω-automata and LTL.

In 1989, Holzmann released the renowned SPIN model checker [16], whose first complete version enabled the automatic verification of concurrent systems. SPIN allows users to express properties through both LTL formulas and "never claims," which are subsequently converted to ω-automata and utilized in the model checking process.

Courcoubetis and Yannakakis generalized LTL model checking to discrete-time Markov chains (DTMC) [6], focusing on qualitative (probability 1 or 0) linear temporal properties. They demonstrated the theoretical feasibility of this approach by converting LTL into a deterministic ω-automaton (eliminating nondeterminism) and analyzing its feasibility. This work laid the foundation for the later analysis of probabilistic systems based on ω-normal (LTL) specifications.

2.5 LTL with past (pLTL)

Past-time Linear Temporal Logic (pLTL) extends Linear Temporal Logic (LTL) by incorporating past-time operators, which makes it easier to express specifications about the past and thus enhances its practical utility.

Lichtenstein, Pnueli, and Zuck introduced past-time connectives, significantly improving the usability of LTL and simplifying the specification of certain properties [19].

Moreover, Markey demonstrated that pLTL can be exponentially more succinct than standard LTL, meaning that some properties, which would require exponentially large formulas in LTL, can be represented more compactly in pLTL. These findings underscore both the theoretical and practical advantages of incorporating past-time operators in temporal logic [20].

2.6 pLTL to ω-automata

Following the introduction of pLTL, early attempts were made to translate this extended logic into ω-automata. One of the initial approaches was a "declarative" construction by Lichtenstein and Pnueli, introduced in [19]. This method involved building all possible states of the automaton upfront, based on maximal consistent subsets of the input formula's subformulas. Rather than providing a practical algorithm for automaton construction, this approach served more as a decision procedure for complexity analysis.

Another technique involves using alternating automata as an intermediate step in the translation from LTL (including pLTL) to non-deterministic Büchi automata. Alternating automata extend non-deterministic automata by allowing both existential and universal branching in their transitions. For LTL with past, researchers have employed two-way very-weak alternating automata (2VAA), which can make transitions based on both the future and the past of the input word. The work of Dax and Klaedtke [13] adopts this approach. While the initial translation to an alternating automaton often results in a size that is linear in the length of the formula, the subsequent conversion to a non-deterministic automaton can be computationally expensive and may still lead to an exponential increase in the number of states.

The most recent algorithm, proposed in Azzopardi, et al.'s work [1], can directly produce deterministic Rabin automata, which this thesis aims to implement and evaluate.

2.7 Gap Analysis

As mentioned above, for converting LTL/pLTL to ω-automata, the industry has proposed and implemented several algorithms. However, none of these implementations support both past operators and direct deterministic ω-automata output.

The following table presents a comparative overview of existing research and the objectives of this thesis. To maintain focus, only the current state-of-the-art work is included for each type of conversion.

Table 2.1: (p)LTL to ω-automata conversion methods

Introduced In	Past Operator Support?	Deterministic Result?	Implementation
Fast LTL to Büchi Automata Translation [11]	No	No	`ltl2ba`
From LTL to deterministic automata [9]	No	Yes	`ltl2dra`
LTL with past and two-way very-weak alternating automata [14]	Yes	No	`pltl2ba`
A Direct Translation from LTL with Past to Deterministic Rabin Automata [1]	Yes	Yes	This thesis

3

Theory

This chapter presents the theoretical concepts used in this thesis. The descriptions are brief but essential for understanding the implementation and evaluation of the algorithm.

The chapter includes the following content:

- **Section 3.1** defines **infinite words**, which can be used in practice to model events in a system. We also define **Linear Temporal Logic with Past (pLTL)**, a modal temporal logic that incorporates modalities referring to both past and future time. We describe its syntax and semantics. In practice, pLTL is intended to specify system properties.
- **Section 3.2** introduces the concept of ω-automata, including their acceptance conditions. These automata are widely used in various formal method tools.
- **Section 3.3** introduces the abstract algorithm targeted for implementation, including the necessary concepts introduced by the original paper [1] that describe this algorithm.

3.1 Infinite words and LTL with Past (pLTL)

3.1.1 Infinite words

An infinite word w over a non-empty finite alphabet Σ is an infinite sequence $\sigma_0, \sigma_1, ...$ of letters from Σ.

For example, for $\Sigma = \{p, q\}$, possible letters are $\sigma_0 = \{\}$, $\sigma_1 = \{p\}$, $\sigma_2 = \{q\}$ and $\sigma_3 = \{p, q\}$, and a word w can be $\sigma_0, \sigma_1, \sigma_0, \sigma_1, \cdots$.

Sometimes we need to explicitly write sets of letters, for example $\{\sigma_0, \sigma_1\}$ or $\{\{\}, \{p\}\}$. Letter sets can also be expressed using boolean expressions, such as $\neg q$ in the last example, to make it more concise. This boolean expression notation is especially beneficial within automaton diagrams, which will be presented subsequently.

3.1.2 pLTL Syntax

Given a non-empty finite set of propositional variables AP, the well-formed formulae ψ of pLTL are generated by the following grammar:

$$\psi ::= \top \mid \bot \mid p \mid \neg\psi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \mathbf{X}\psi \mid \psi\mathbf{U}\psi \mid \mathbf{Y}\psi \mid \psi\mathbf{S}\psi$$

where $p \in AP$.

3.1.3 pLTL Semantics

Given a formula ψ, a natural number t, and an infinite word w which the t-th element is σ_t, we write $(w, t) \models \psi$ to denote that w satisfies ψ at index t. Or in other words, ψ holds on time t when given word w.

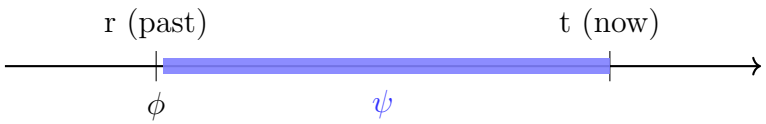
$(w,t) \models \top$
$(w,t) \not\models \bot$
$(w,t) \models p$ iff $p \in \sigma_t$
$(w,t) \models \neg\psi$ iff $(w,t) \not\models \psi$
$(w,t) \models \psi \wedge \phi$ iff $(w,t) \models \psi$ and $(w,t) \models \phi$
$(w,t) \models \psi \vee \phi$ iff $(w,t) \models \psi$ or $(w,t) \models \phi$
$(w,t) \models \mathbf{X}\psi$ iff $(w,t+1) \models \psi$
$(w,t) \models \mathbf{Y}\psi$ iff $t > 0$ and $(w,t-1) \models \psi$
$(w,t) \models \psi\mathbf{U}\phi$ iff $\exists r \geq t.((w,r) \models \phi$ and $\forall s \in [t,r).(w,s) \models \psi)$
$(w,t) \models \psi\mathbf{S}\phi$ iff $\exists r \leq t.((w,r) \models \phi$ and $\forall s \in (r,t].(w,s) \models \psi)$

The semantics of $\top, \bot, p, \neg\psi, \psi \wedge \psi, \psi \vee \psi$ should be familiar for the target readers. For the other 4 temporal operands, informally:

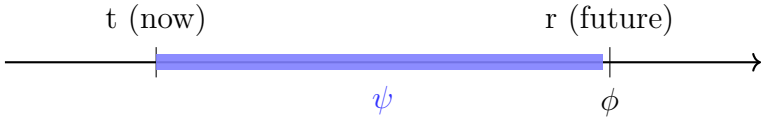
$(w,t) \models \mathbf{X}\psi$ means ψ holds for ne**X**t state, this is part of the standard LTL.

$(w,t) \models \mathbf{Y}\psi$ means ψ holds for last (**Y**esterday's) state, note $\mathbf{Y}\psi$ does not hold at $t = 0$.

$(w,t) \models \psi\mathbf{S}\phi$ means ψ remains holding **S**ince ϕ happened, and it additionally requires that ϕ did indeed occur at some point in the past.



S is dual to **U** in standard LTL. Specifically, $(w,t) \models \psi \mathbf{U} \phi$ means ψ holds now, and will remain holding **U**ntil ϕ happens, further requiring that ϕ will indeed happen at some point in the future.



There are also some operators which can be derived from those:

$$\mathbf{F}\varphi := \top \mathbf{U} \varphi \qquad \mathbf{O}\varphi := \top \mathbf{S} \varphi \qquad \mathbf{G}\varphi := \neg \mathbf{F} \neg \varphi$$
$$\mathbf{H}\varphi := \neg \mathbf{O} \neg \varphi \qquad \varphi \mathbf{W} \psi := \varphi \mathbf{U} \psi \vee \mathbf{G}\varphi \qquad \varphi \tilde{\mathbf{S}} \psi := \varphi \mathbf{S} \psi \vee \mathbf{H}\varphi$$
$$\varphi \mathbf{M} \psi := \psi \mathbf{U} (\varphi \wedge \psi) \qquad \varphi \mathbf{B} \psi := \psi \mathbf{S} (\varphi \wedge \psi) \qquad \varphi \mathbf{R} \psi := \psi \mathbf{W} (\varphi \wedge \psi)$$
$$\varphi \tilde{\mathbf{B}} \psi := \psi \tilde{\mathbf{S}} (\varphi \wedge \psi) \qquad \tilde{\mathbf{Y}}\varphi := \mathbf{Y}\varphi \vee \neg \mathbf{Y} \top$$
$$\varphi \rightarrow \psi := \neg \varphi \vee \psi \qquad \varphi \leftrightarrow \psi := (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$$

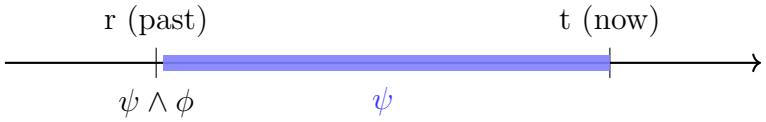
$(w,t) \models \mathbf{F}\psi$ means ψ will hold in the **F**uture, or ψ eventually holds.

$(w,t) \models \mathbf{G}\psi$ means ψ holds **G**lobally, or ψ always holds. Except its definition, $\mathbf{G}\psi$ is also equivalent with $\psi \mathbf{W} \bot$.

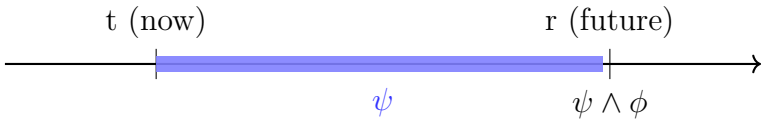
$(w,t) \models \mathbf{O}\psi$ means ψ **O**nce holds.

$(w,t) \models \mathbf{H}\psi$ means ψ **H**istorically holds.

$(w,t) \models \phi \mathbf{B} \psi$ means ψ remains holding **B**ack to ϕ happened, and it additionally requires that $\psi \wedge \phi$ did indeed occur at some point in the past.



B is dual to **M** in standard LTL. Specifically, $(w,t) \models \psi \mathbf{M} \phi$ means ψ holds until some time $\psi \wedge \phi$ holds at the same time. And it requires that ψ and ϕ will indeed happen at some point in the future.



M is the "strong" or **M**ighty version of **R**. **R** is just like **M** but does not have the requirement that ψ and ϕ will happen at some point in the future.

Similarly,

$\mathbf{W}$ is the "weak" version of $\mathbf{U}$, which $\varphi\mathbf{W}\psi$ does not require ψ will happen in the future.

$\tilde{\mathbf{Y}}$ is the "weak" version of $\mathbf{Y}$, which holds at the $t = 0$.

$\tilde{\mathbf{S}}$ is the "weak" version of $\mathbf{S}$, which $\psi\tilde{\mathbf{S}}\phi$ does not require ϕ occured in the past. It also holds at the $t = 0$.

$\tilde{\mathbf{B}}$ is the "weak" version of $\mathbf{B}$, which $\psi\tilde{\mathbf{B}}\phi$ does not require $\psi \wedge \phi$ occurred in the past. It also holds at $t = 0$.

These operators are referred to as "weak" (or "strong") because they accept a slightly larger (or smaller) set of words when compared to their "strong" (or "weak") counterparts.

The semantics of $\rightarrow$ and $\leftrightarrow$ should be familiar for the target readers.

3.2 ω-automata

An ω-automaton over an alphabet Σ is a quadruple (Q, Q_0, δ, α), where

- Q is a finite set of states
- $Q_0 \subseteq Q$ is a non-empty set of initial states
- $\delta \in Q \times \Sigma \rightarrow 2^Q$ is a (partial) transition function
- α a set constituting its acceptance condition, which we will introduce later.

In the case where $|Q_0| = 1$ and $|\delta(q, \sigma)| \leq 1$ for all $q \in Q$ and all $\sigma \in \Sigma$, the automaton is called deterministic. For deterministic automata we can write $\delta : Q \times \Sigma \rightarrow Q$.

Determinism is useful when doing probabilistic model checking and program synthesis because it provides an unambiguous set of actions for any given input.

Given an ω-automaton $A = (Q, Q_0, \delta, \alpha)$ and an infinite word $w = \sigma_0, \sigma_1, ...$, both over the same alphabet, a run of A on w is a sequence of states $\mathrm{r} = r_0, r_1, ...$ of Q such that $r_0 \in Q_0$ and $r_{i+1} \in \delta(r_i, \sigma_i)$ for all $i \leq 0$.

Given such a run r, we write $\mathrm{Inf}(r)$ to denote the set of states appearing infinitely often in r.

3.2.1 Acceptance conditions

There are many different types of acceptance conditions for ω-automata[10]. In this thesis, we will use only three of them: Büchi acceptance condition, co-Büchi acceptance condition and Rabin acceptance condition. [1]

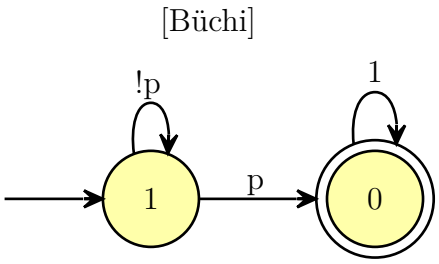
3.2.1.1 Büchi acceptance condition

A Büchi acceptance condition is a set $Q' \subseteq Q$, which means the automata accepts the infinite word w iff there exists a run r on w such that $\text{Inf}(r) \cap Q' \neq \varnothing$. We call an ω-automata with Büchi acceptance condition a Büchi automaton.

In a less formal way, a Büchi automaton accepts a word if, given the word, there exists a run which visits (one of) the accept state(s) infinitely often.

We can observe that Büchi automata are good at representing the fact that "something is guarantee to happen".

For example, the following ω-automaton with a Büchi acceptance condition accepts a word in which p eventually exists, which corresponds to the LTL formula $\mathbf{F}p$.



3.2.1.2 co-Büchi acceptance condition

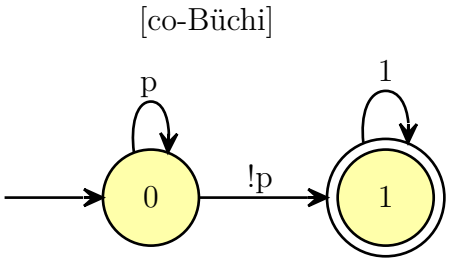
A co-Büchi acceptance condition is dual to a Büchi acceptance condition. It is also a set $Q' \subseteq Q$, which means the automata accepts the infinite word w iff there exists a run r on w such that $\text{Inf}(r) \cap Q' = \varnothing$. We call an ω-automaton with co-Büchi acceptance condition a co-Büchi automaton.

In a less formal way, a co-Büchi automaton accepts a word if, given the word, there exists a run which visits the accept state(s) only finite times.

We can observe that co-Büchi automata are good at representing the fact that "something would never happen".

For example, the following ω-automaton with a co-Büchi acceptance condition accepts a word in which p always exists (or, $\neg p$ never exists), which is corresponding to the LTL formula $\mathbf{G}p$.

[1] Note in our work, we use state based acceptance conditions, which means that the acceptance set is a (or several) set(s) of states. Some tools like Spot [8] may produce transition based acceptance conditions with the same names, and equivalent with the state based versions.



3.2.1.3 Rabin acceptance condition

A Rabin automaton has a set of subsets $R \subseteq 2^Q \times 2^Q$ as acceptance condition, and accepts w iff there exists a run r on w and a pair $(A, B) \in R$ such that $\text{Inf}(r) \cap A = \varnothing$ and $\text{Inf}(r) \cap B \neq \varnothing$.

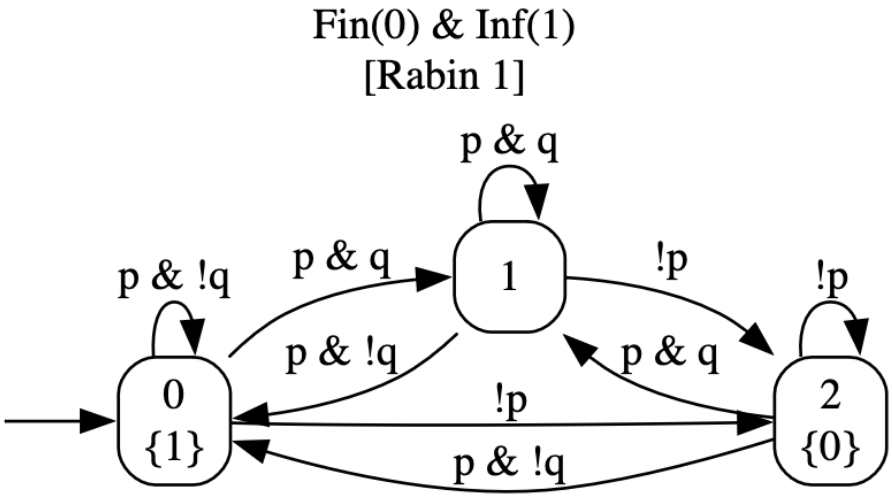
We call an ω-automaton with Rabin acceptance condition a Rabin automaton.

In a less formal way, a Rabin automaton has several pairs of state sets, and it accepts a word if, given the word, there exists a run and a pair (A, B), where the run visits the states in set A finite times, and the states in set B infinite often.

We can observe that Rabin automata combine the benefits of Büchi and co-Büchi automata.

A benefit of introducing Rabin automata is, a deterministic Rabin automata can recognize all ω-languages, meanwhile deterministic Büchi and co-Büchi automata are strictly weaker[26].

For example, the following ω-automaton with Rabin acceptance condition is corresponding to the LTL formula $\mathbf{GF}(p \rightarrow \mathbf{G}q)$.



You cannot find an equivalent deterministic Büchi or co-Büchi automata.

3.3 Abstract Algorithm

This section describes the abstract algorithm that has been implemented.

It covers the following content:

- **Section 3.3.1** introduces the necessary preprocessing steps to be performed on the pLTL formula, which are essential for the subsequent parts to function correctly.
- **Section 3.3.2** introduces the concepts employed by the algorithm to encode information that an automaton may have already processed. This information is used to check whether a part of a pLTL formula is satisfied in the future.
- Each section of **Section 3.3.3** to **Section 3.3.6** details the construction of a sub-automaton or a set of sub-automata.
- **Section 3.3.7** explains the process of merging these sub-automata into a single deterministic Rabin automaton.

3.3.1 Preprocessing

The algorithm requires the pLTL formula to be in its negation normal form, specifically without the operators **F**,**G**,**O** and **H**. This means that all instances of these operators are eliminated using the previously mentioned equivalences, and negations ($\neg$) are pushed down through the formula to the atomic propositions, utilizing the following equivalence relation:

$$
\begin{aligned}
\neg\top &\equiv \bot & \neg\bot &\equiv \top \\
\neg(\psi \wedge \xi) &\equiv \neg\psi \vee \neg\xi & \neg(\psi \vee \xi) &\equiv \neg\psi \wedge \neg\xi \\
\neg(\psi \mathbf{U} \xi) &\equiv (\neg\psi)\mathbf{R}(\neg\xi) & \neg(\psi \mathbf{R} \xi) &\equiv (\neg\psi)\mathbf{U}(\neg\xi) \\
\neg(\psi \mathbf{W} \xi) &\equiv (\neg\psi)\mathbf{M}(\neg\xi) & \neg(\psi \mathbf{M} \xi) &\equiv (\neg\psi)\mathbf{W}(\neg\xi) \\
\neg(\psi \mathbf{S} \xi) &\equiv (\neg\psi)\tilde{\mathbf{B}}(\neg\xi) & \neg(\psi \tilde{\mathbf{B}} \xi) &\equiv (\neg\psi)\mathbf{S}(\neg\xi) \\
\neg(\psi \tilde{\mathbf{S}} \xi) &\equiv (\neg\psi)\mathbf{B}(\neg\xi) & \neg(\psi \mathbf{B} \xi) &\equiv (\neg\psi)\tilde{\mathbf{S}}(\neg\xi) \\
\neg(\mathbf{Y}\psi) &\equiv \tilde{\mathbf{Y}}(\neg\psi) & \neg(\tilde{\mathbf{Y}}\psi) &\equiv \mathbf{Y}(\neg\psi) \\
\neg(\mathbf{X}\psi) &\equiv \mathbf{X}(\neg\psi)
\end{aligned}
$$

3.3.2 Encoding the past

For constructing automata for pLTLs with the past-related operators (**Y**, **S** and **B**), we need to "remember" some history information, thus, we need to define the following three functions:

3.3.2.1 Past Rewriting

We can define a pair of functions $(-)_w$ and $(-)_s$ to represent the weakening and strengthening of **past** formulas:

$$\begin{array}{lll} (\mathbf{Y}\psi)_w := \tilde{\mathbf{Y}}\psi & (\psi\mathbf{S}\xi)_w := \psi\tilde{\mathbf{S}}\xi & (\psi\mathbf{B}\xi)_w := \psi\tilde{\mathbf{B}}\xi \\ (\tilde{\mathbf{Y}}\psi)_s := \mathbf{Y}\psi & (\psi\tilde{\mathbf{S}}\xi)_s := \psi\mathbf{S}\xi & (\psi\tilde{\mathbf{B}}\xi)_s := \psi\mathbf{B}\xi. \end{array}$$

For all other cases we have $\psi_w := \psi$ and $\psi_s := \psi$.

3.3.2.2 Rewriting under past formula sets

We can rewrite a formula recursively with a set of formulas.

$$\begin{aligned} a\langle C\rangle &:= a && (a \text{ is atomic}) \\ (op\ \psi)\langle C\rangle &:= \begin{cases} (op\ \psi\langle C\rangle)_w & (op\ \psi \in C) \\ (op\ \psi\langle C\rangle)_s & (\text{otherwise}) \end{cases} && (op \text{ is unary}) \\ (\psi\ op\ \xi)\langle C\rangle &:= \begin{cases} (\psi\langle C\rangle\ op\ \xi\langle C\rangle)_w & (\psi\ op\ \xi \in C) \\ (\psi\langle C\rangle\ op\ \xi\langle C\rangle)_s & (\text{otherwise}) \end{cases} && (op \text{ is binary}) \end{aligned}$$

Informally, the purpose of this rewrite function is to weaken the past subformulas in a formula, provided the subformula is an element of the set C, and strengthen the rest past subformulas.

3.3.2.3 Future Rewriting

For **future** operators we have also regarded **U** and **M** as strengthened froms of **W** and **R**. We denote subformulas of φ whose syntax trees are rooted with **W** or **R** as $\nu(\varphi)$ and subformulas of φ whose syntax trees are rooted with **U** or **M** as $\mu(\varphi)$.

We can also define rewrite functions for these future subformulas:

$$
\begin{aligned}
a[M]_\nu &:= a && (a \text{ atomic})\\
(op\,\psi)[M]_\nu &:= op\,(\psi[M]_\nu) && (op \text{ unary})\\
(\psi\, op\,\xi)[M]_\nu &:= (\psi[M]_\nu)\, op\, (\xi[M]_\nu) && (op \in \{\mathbf{W}, \mathbf{R}, \mathbf{S}, \tilde{\mathbf{S}}, \mathbf{B}, \tilde{\mathbf{B}}, \wedge, \vee\})\\
(\psi\,\mathbf{U}\,\xi)[M]_\nu &:= \begin{cases} (\psi[M]_\nu)\,\mathbf{W}\,(\xi[M]_\nu) & (\psi\,\mathbf{U}\,\xi \in M)\\ \bot & (\text{otherwise}) \end{cases}\\
(\psi\,\mathbf{M}\,\xi)[M]_\nu &:= \begin{cases} (\psi[M]_\nu)\,\mathbf{R}\,(\xi[M]_\nu) & (\psi\,\mathbf{M}\,\xi \in M)\\ \bot & (\text{otherwise}), \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
a[N]_\mu &:= a && (a \text{ atomic})\\
(op\,\psi)[N]_\mu &:= op\,(\psi[N]_\mu) && (op \text{ unary})\\
(\psi\, op\,\xi)[N]_\mu &:= (\psi[N]_\mu)\, op\, (\xi[N]_\mu) && (op \in \{\mathbf{U}, \mathbf{M}, \mathbf{S}, \tilde{\mathbf{S}}, \mathbf{B}, \tilde{\mathbf{B}}, \wedge, \vee\})\\
(\psi\,\mathbf{W}\,\xi)[N]_\mu &:= \begin{cases} \top & (\psi\,\mathbf{W}\,\xi \in N)\\ (\psi[N]_\mu)\,\mathbf{U}\,(\xi[N]_\mu) & (\text{otherwise}) \end{cases}\\
(\psi\,\mathbf{R}\,\xi)[N]_\mu &:= \begin{cases} \top & (\psi\,\mathbf{R}\,\xi \in N)\\ (\psi[N]_\mu)\,\mathbf{M}\,(\xi[N]_\mu) & (\text{otherwise}). \end{cases}
\end{aligned}
$$

3.3.2.4 Weakening conditions

Take $\psi\mathbf{S}\xi$ as an example. If ξ is satisfied at the current time point, then it is sufficient to check whether $\psi\tilde{\mathbf{S}}\xi$ holds starting from the next time point.

Conversely, if $\psi\tilde{\mathbf{S}}\xi$ is already satisfied, then if either ξ or ψ is satisfied, its weak form can continue to be checked at the next time point.

Similar situations exist for $\tilde{\mathbf{Y}}\psi$ and $\psi\tilde{\mathbf{B}}\xi$.

So we can define a function wc, which shows when we can weaken a formula:

$$
\begin{aligned}
wc(\mathbf{Y}\psi) &:= \psi & wc(\psi\mathbf{S}\xi) &:= \xi & wc(\psi\mathbf{B}\xi) &:= \psi \wedge \xi\\
wc(\tilde{\mathbf{Y}}\psi) &:= \psi & wc(\psi\tilde{\mathbf{S}}\xi) &:= \psi \vee \xi & wc(\psi\tilde{\mathbf{B}}\xi) &:= \xi.
\end{aligned}
$$

3.3.2.5 (local) after function

The local after function is defined as:

$$
\begin{aligned}
\mathrm{af}_l(\top, \sigma, C) &:= \top \\
\mathrm{af}_l(\bot, \sigma, C) &:= \bot \\
\mathrm{af}_l(p, \sigma, C) &:= \text{if } (p \in \sigma) \text{ then } \top \text{ else } \bot \\
\mathrm{af}_l(\neg p, \sigma, C) &:= \text{if } (p \in \sigma) \text{ then } \bot \text{ else } \top \\
\mathrm{af}_l(\mathbf{X}\psi, \sigma, C) &:= \mathrm{pu}_l(\psi, \sigma, C) \\
\mathrm{af}_l(\mathbf{Y}\psi, \sigma, C) &:= \bot \\
\mathrm{af}_l(\tilde{\mathbf{Y}}\psi, \sigma, C) &:= \top \\
\mathrm{af}_l(\psi \mathbin{op} \xi, \sigma, C) &:= \mathrm{af}_l(\psi, \sigma, C) \mathbin{op} \mathrm{af}_l(\xi, \sigma, C) && (op \in \{\wedge, \vee\}) \\
\mathrm{af}_l(\psi \mathbin{op} \xi, \sigma, C) &:= \mathrm{af}_l(\xi, \sigma, C) \vee \mathrm{af}_l(\psi, \sigma, C) \wedge \mathrm{pu}_l(\psi \mathbin{op} \xi, \sigma, C) && (op \in \{\mathbf{U}, \mathbf{W}\}) \\
\mathrm{af}_l(\psi \mathbin{op} \xi, \sigma, C) &:= \mathrm{af}_l(\xi, \sigma, C) \wedge (\mathrm{af}_l(\psi, \sigma, C) \vee \mathrm{pu}_l(\psi \mathbin{op} \xi, \sigma, C)) && (op \in \{\mathbf{R}, \mathbf{M}\}) \\
\mathrm{af}_l(\psi \mathbin{op} \xi, \sigma, C) &:= \mathrm{af}_l(wc(\psi \mathbin{op} \xi), \sigma, C) && (op \in \{\mathbf{S}, \tilde{\mathbf{S}}, \mathbf{B}, \tilde{\mathbf{B}}\}),
\end{aligned}
$$

where

$$
\mathrm{pu}_l(\varphi, \sigma, C) := \varphi\langle C \rangle \wedge \bigwedge_{\psi \in psf(\varphi) \cap C} \mathrm{af}_l(wc(\psi), \sigma, C)
$$

In the context of reading the initial letter σ of the word w, the local after function separates the formula into two parts: one can be fully evaluated to $\top$ or $\bot$ using σ, and parts that can only be partially evaluated using σ.

We also define the after function:

$$
\mathrm{af}(\varphi, \sigma) := \bigvee_{C \in 2^{\mathrm{psf}(\varphi)}} \mathrm{af}_l(\varphi, \sigma, C)
$$

This is simply the disjunction of all local after functions, with C iterating through every subset of past subformulas of φ (denoted as $\mathrm{psf}(\varphi)$).

3.3.3 Weakening Conditions automaton

Just as in the paper that introduced the algorithm [1], for the remainder of this section, we will consider a fixed formula φ that is to be translated into a Rabin automaton, and a fixed ordering $C_1, C_2, \ldots, C_k$ of the elements of $2^{\mathrm{psf}(\varphi)}$. For simplicity, we assume that $C_1 = \{\psi \in \mathrm{psf}(\varphi) \mid \psi = \psi_w\}$.

The first automaton constructed by the algorithm tracks the development of weakening conditions under rewrites of the local after function. Its states are k-tuples of formulas, denoted as $\psi^\times = \langle \psi_1, \psi_2, \ldots, \psi_k \rangle$, where k equals $2^{\mathrm{len}(\mathrm{psf}(\varphi))}$ with φ as the original pLTL formula to be transformed. Each element in this tuple describes the requirements that remain to be verified in order to justify a sequence of rewrites.

The transition function is defined as:

$$\psi'_i = \bigvee_{j \in J_i} \left(\mathrm{af}_l(\psi_j, \sigma, C_i\langle C_j \rangle) \wedge \bigwedge_{\xi \in C_i} \mathrm{af}_l(wc(\xi\langle C_j \rangle), \sigma, C_i\langle C_j \rangle) \right)$$

where

$$J_i := \{j \in [1..k] \mid \forall \xi, \xi' \in psf(\varphi).\xi\langle C_j \rangle = \xi'\langle C_j \rangle \Rightarrow \xi\langle C_i \rangle = \xi'\langle C_i \rangle\}$$

The initial state S_0 is the k-tuple $\langle \top, \bot, \cdots \bot \rangle$.

This automaton has no acceptance state, but other automata's transition function depend on this one.

3.3.4 Guarantee automata

This family of automata verifies the "guarantee" part of a pLTL formula.

Given a formula $\psi \in \mu(\varphi)$ and set $N \subseteq \nu(\varphi)$, we have the transition function:

$$\delta(\zeta, \xi^\times, \sigma) := \begin{cases} \bigvee_{i \in [1..k]} \mathbf{F}(\psi\langle C_i \rangle [N\langle C_i \rangle]_\mu) \wedge \xi_i[N\langle C_i \rangle]_\mu & (\zeta \sim \top) \\ \mathrm{af}(\zeta, \sigma) & (\text{otherwise}), \end{cases}$$

an initial state $Q_0 := \mathbf{F}(\psi[N]_\mu)$, and a Büchi acceptance condition $\alpha := \top$.

3.3.5 Safety automata

This family of automata verifies the "safety" part of a pLTL formula. It is dual to the guarantee automata.

Given a formula $\psi \in \nu(\varphi)$ and set $M \subseteq \mu(\varphi)$, we have the transition function:

$$\delta(\zeta, \xi^{\times}, \sigma) := \begin{cases} \bigvee_{i \in [1..k]} \mathbf{G}(\psi(C_i)[M(C_i)]_\nu) \wedge \xi_i[M(C_i)]_\nu & (\zeta \sim \bot) \\ \mathrm{af}(\zeta, \sigma) & (\text{otherwise}), \end{cases}$$

an initial state $Q_0 := \mathbf{G}(\psi[M]_\nu)$, and a co-Büchi acceptance condition $\alpha := \bot$.

3.3.6 Stable automata

This family of automata is to "guess" an index at which w is stable (all subformulas that should be eventually satisfied are infinitely often satisfied, and all subformulas that should be almost always satisfied would never fail to be satisfied) with respect to φ, starting with the guess that it is initially stable.

Given a set $M \subseteq \mu(\varphi)$, we have the transition function:

$$\delta(\langle \psi, \zeta \rangle, \xi^{\times}, \sigma) := \begin{cases} \langle \mathrm{af}(\psi, \sigma), \bigvee_{i \in [1..k]} \mathrm{af}(\psi, \sigma)[M\langle C_i \rangle]_\nu \wedge \xi_i[M\langle C_i \rangle]_\nu \rangle & (\zeta \sim \bot) \\ \langle \mathrm{af}(\psi, \sigma), \mathrm{af}(\zeta, \sigma) \rangle & (\text{otherwise}) \end{cases}$$

an initial state $Q_0 := \langle \varphi, \varphi[M]_\nu \rangle$, and a co-Büchi acceptance condition $\alpha := \langle -, \bot \rangle$.

3.3.7 Final Rabin automata

For each $M \subseteq \mu(\varphi)$ and $N \subseteq \nu(\varphi)$, we can construct a Rabin automaton with one Rabin acceptance pair like this:

$$\mathcal{B}_{M,N} := \bigcap_{\psi \in M} \mathcal{B}^{\psi}_{N} \quad \mathcal{C}_{M,N} := \bigcap_{\psi \in N} \mathcal{C}^{\psi}_{M} \tag{3.1}$$

$$\mathcal{R}_{\varphi,M,N} := \mathcal{C}_{\varphi,M} \cap \mathcal{B}_{M,N} \cap \mathcal{C}_{M,N} \tag{3.2}$$

And the final result is a union of them all.

$$\mathcal{A}_{DRA}(\varphi) := \bigcup_{\substack{M \subseteq \mu(\varphi) \\ N \subseteq \nu(\varphi)}} \mathcal{R}_{\varphi,M,N}.$$

4

Methods

This chapter provides a detailed description of the algorithm's implementation, subsequent optimizations, and its empirical benchmarking.

- **Section 4.1** introduces the implementation process and the optimizations performed. This includes modifications made to the algorithm for increased efficiency, along with a description of the data structures and specific algorithms utilized in the implementation.
- **Section 4.2** will outline the benchmarking methodology employed for the program. This encompasses how the pLTL formulas were collected or generated, and the intended purpose of the collected data.

4.1 Implementation and optimization

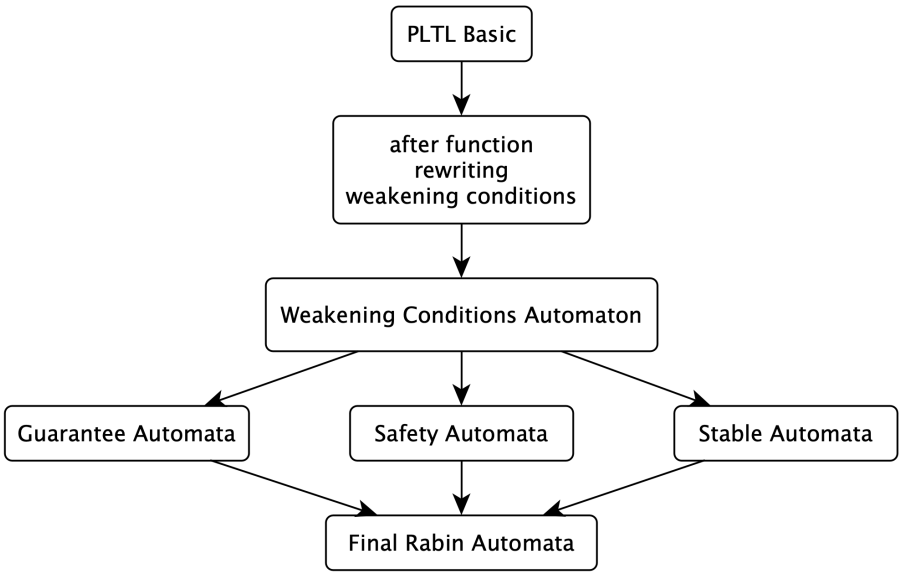
We implemented the automaton construction algorithm described in Chapter 3.

To achieve a balance between ease of implementation and performance, Rust was chosen for its development.

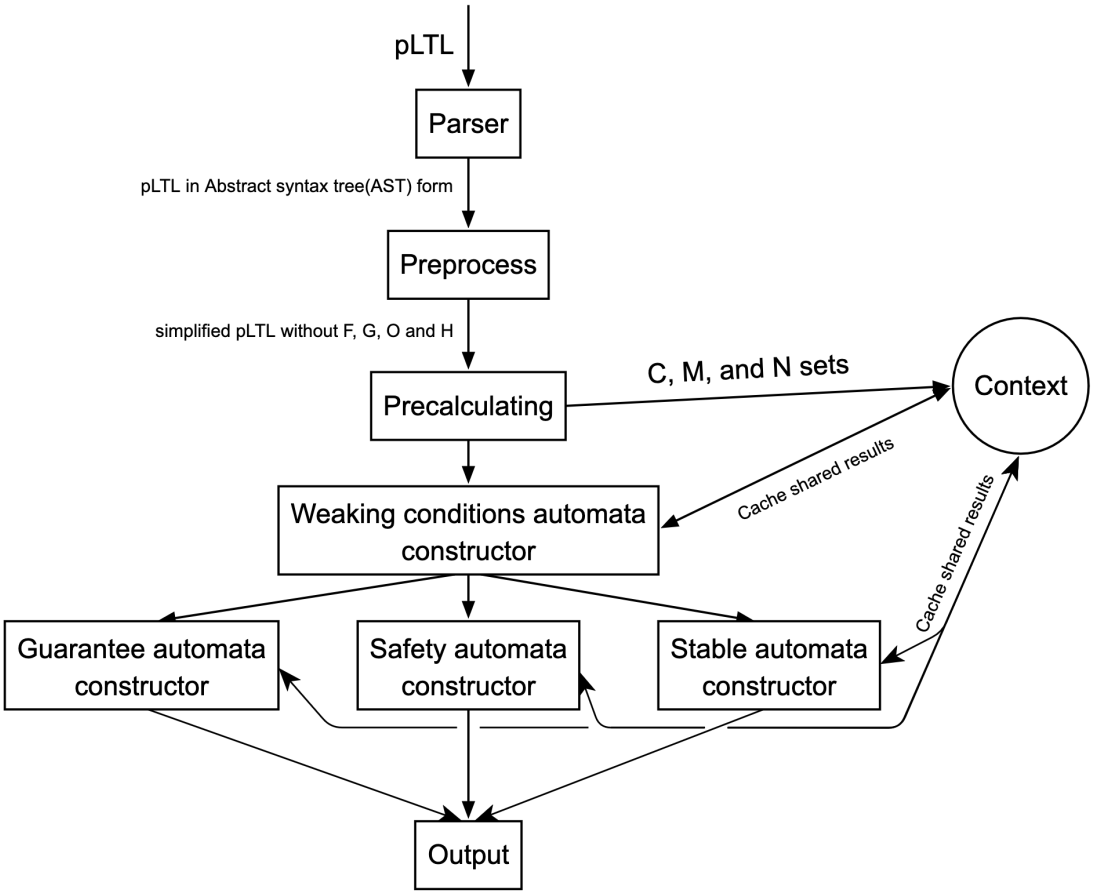
The resulting product is presented in three forms:

- Important functions and data structures is exported as a Rust library, enabling their use by other projects.
- A Command Line Interface (CLI) binary executable program `pltl2dra` was built, featuring an interface similar to tools found in Spot [8].
- An online version was developed by exporting the library into a WebAssembly format. This version is primarily intended for demonstration purposes.

The implementation process can be described by the following flowchart:



This directly corresponds to the implemented program's architecture:



4.1.1 pLTL Basics

This section describes the infrastructure required for parsing, manipulating, simplifying, and formatting pLTL formulas.

4.1.1.1 pLTL Parser

A parser combinator-based parser was implemented using `nom` [5]. To mitigate unnecessary backtracking, a cache for internal results was used.

4.1.1.2 pLTL Abstract Syntax Tree Representation

It is observed that the following operations need to be performed on pLTL formulas:

- Rewriting under past sets, specifically flipping operators between $\mathbf{Y}, \mathbf{S}, \mathbf{B}$ and $\tilde{\mathbf{Y}}, \tilde{\mathbf{S}}, \tilde{\mathbf{B}}$.
- Rewriting under μ and ν sets, specifically flipping operators between $\mathbf{U}, \mathbf{M}$ and $\mathbf{W}, \mathbf{R}$.
- Performing $\bigwedge$ and $\bigvee$ operations on numerous pLTL formulas.

4.1.1.3 Atomic propositions' representation

For atomic propositions, these are represented as integers, a practice common in many other tools [8]. The benefit of this approach is that an input set can be represented efficiently as a bit set.

For instance, if atomic propositions 'p' and 'q' are mapped to 0 and 1 respectively, then an input set $\{p\}$ can be represented as `0b01`, $\{p, q\}$ as `0b11`, and $\{\}$ as `0b00`.

4.1.1.4 Representation of subformulas rooted by past operators

The properties of subformulas rooted by past operators ($\mathbf{Y}, \mathbf{S}, \mathbf{B}$ and $\tilde{\mathbf{Y}}, \tilde{\mathbf{S}}, \tilde{\mathbf{B}}$) are the operator itself and its subformulas.

However, we have opted to assign each such subformula a unique integer ID.

The benefits of this approach include:

- C sets can be represented efficiently with a bit set. With the i-th bit stand for whether the past formula labeled by i exists in the set. Note that this representation causes the indexing of C sets used in the implementation to differ from that presented in the theoretical section. In the implementation the index of C sets starts from 0 instead of 1, and C_1 does not have to be $\{\psi \in \mathrm{psf}(\varphi) \mid \psi = \psi_w\}$.
- It allows us to "skip" some of the rewritings under past sets. The mechanism and rationale for achieving this will be explained subsequently.

Furthermore, instead of treating them as entirely distinct operators, the strong and weak versions of a past operator are distinguished by employing a dedicated bit (`1` for weak). This provides the advantage of being able to set or reset a bit when performing rewrites under sets, a process that will be covered later.

4.1.1.5 Representing of subformulas rooted by U/W or R/M

Similar to past subformulas, **W** is regarded as the weaker form of **U**, and **R** as the weaker form of **M**. A dedicated bit is used to distinguish between them. In other words, for each entry in the following table, the operators are represented as the same underlying operator, accompanied by a bit indicating its strength.

Weak	Strong
W	**U**
R	**M**

The benefit of this approach is the ability to set or reset a bit during μ and ν rewrites, a process that will be discussed later.

4.1.1.6 Representation of $\wedge$ and $\vee$

The naive approach of representing $\wedge$ and $\vee$ were to represent them in the same manner as other binary operators, such as **U** or **R**, specifically as a tree node with two sub-trees as its children. However, this approach soon presented several problems:

The naive approach for representing $\wedge$ and $\vee$ involved treating them similarly to other binary operators, such as **U** or **R**, specifically as a tree node with two sub-trees as its children. However, this approach soon presented several problems:

- Given that numerous formulas are typically connected by $\wedge$ and $\vee$, treating them as normal tree nodes tended to generate deeply nested data structures, leading to suboptimal performance.
- It proved challenging to efficiently determine the equivalence of two $\wedge$ or $\vee$ nodes when represented in this way. For example, efficiently deciding whether $p \wedge (q \wedge (r \wedge s))$, $(p \wedge q) \wedge (r \wedge s)$, and $((p \wedge q) \wedge r) \wedge s$ are equivalent would be troublesome.

To overcome these issues, a vector containing all sub-parts connected by these operators is utilized. When comparisons are necessary, all nested values are flattened, and these sub-parts are ordered to achieve a semi-normal form. This semi-normal form will be discussed further in the next paragraph.

4.1.1.7 Semi-normal Form

In the paper that introduced the algorithm [1], many constructs are defined under propositional equivalence classes to constrain the state space, but it is difficult

to judge whether two pLTL functions are propositional equivalent in an efficient manner.

Instead, we choose to convert the pLTL formulas to a semi-normal form to achieve a similar result: an approximation of semantic equivalence between pLTL formulas.

To obtain the semi-normal form of a pLTL formula, a rewriting system with the following rules is applied:

$$
\begin{array}{rrrrr}
\mathbf{Y}\bot \Rightarrow \bot & \tilde{\mathbf{Y}}\top \Rightarrow \top & & & \\
\mathbf{X}\bot \Rightarrow \bot & \mathbf{X}\top \Rightarrow \top & \mathbf{XY}\varphi \Rightarrow \varphi & & \\
\bot\mathbf{U}\varphi \Rightarrow \varphi & \varphi\mathbf{U}\top \Rightarrow \top & \varphi\mathbf{U}\varphi \Rightarrow \varphi & \varphi\mathbf{U}\bot \Rightarrow \bot & \varphi\mathbf{U}(\varphi\mathbf{U}\phi) \Rightarrow \varphi\mathbf{U}\phi \\
\bot\mathbf{W}\varphi \Rightarrow \varphi & \varphi\mathbf{W}\top \Rightarrow \top & \varphi\mathbf{W}\varphi \Rightarrow \varphi & \top\mathbf{W}\varphi \Rightarrow \top & (\varphi\mathbf{W}\phi)\mathbf{W}\phi \Rightarrow \varphi\mathbf{W}\phi \\
\bot\mathbf{M}\varphi \Rightarrow \bot & \top\mathbf{M}\varphi \Rightarrow \varphi & \varphi\mathbf{M}\bot \Rightarrow \bot & \varphi\mathbf{M}\varphi \Rightarrow \varphi & (\varphi\mathbf{M}\phi)\mathbf{M}\phi \Rightarrow \varphi\mathbf{M}\phi \\
\varphi\mathbf{R}\top \Rightarrow \top & \top\mathbf{R}\varphi \Rightarrow \varphi & \varphi\mathbf{R}\bot \Rightarrow \bot & \varphi\mathbf{R}\varphi \Rightarrow \varphi & \varphi\mathbf{R}(\varphi\mathbf{R}\phi) \Rightarrow \varphi\mathbf{R}\phi \\
\varphi\mathbf{S}\bot \Rightarrow \bot & \bot\mathbf{S}\varphi \Rightarrow \varphi & \varphi\mathbf{S}\varphi \Rightarrow \varphi & & \\
\varphi\tilde{\mathbf{S}}\top \Rightarrow \top & \bot\tilde{\mathbf{S}}\varphi \Rightarrow \varphi & \top\tilde{\mathbf{S}}\varphi \Rightarrow \top & \varphi\tilde{\mathbf{S}}\varphi \Rightarrow \varphi & \\
\varphi\mathbf{B}\bot \Rightarrow \bot & \top\mathbf{B}\varphi \Rightarrow \varphi & \bot\mathbf{B}\varphi \Rightarrow \bot & \varphi\mathbf{B}\varphi \Rightarrow \varphi & \\
\varphi\tilde{\mathbf{B}}\bot \Rightarrow \bot & \varphi\tilde{\mathbf{B}}\top \Rightarrow \top & \top\tilde{\mathbf{B}}\varphi \Rightarrow \varphi & \varphi\tilde{\mathbf{B}}\varphi \Rightarrow \varphi &
\end{array}
$$

For $\bigwedge$ and $\bigvee$ operations, an array of pLTL formulas is denoted by Σ, Γ, and Π. The order of these pLTL formulas can be arbitrarily defined.

In practice, the last two rules for $\bigwedge$ and $\bigvee$ are implemented by sorting and uniquify the content array.

$$
\begin{array}{ccc}
\bigwedge[\varphi] \Rightarrow \varphi & \bigwedge[\Sigma, \bot, \Gamma] \Rightarrow \bot & \bigwedge[\Sigma, \top, \Gamma] \Rightarrow \bigwedge[\Sigma, \Gamma] \\
\bigwedge[\Sigma, \bigwedge[\Gamma], \Pi] \Rightarrow \bigwedge[\Sigma, \Gamma, \Pi] & \bigwedge[\Sigma, \varphi, \varphi, \Pi] \Rightarrow \bigwedge[\Sigma, \varphi, \Pi] & \bigwedge[\Sigma, \phi, \varphi, \Pi] \overset{\varphi<\phi}{\Longrightarrow} \bigwedge[\Sigma, \varphi, \phi, \Pi] \\
\bigvee[\varphi] \Rightarrow \varphi & \bigvee[\Sigma, \top, \Gamma] \Rightarrow \top & \bigvee[\Sigma, \bot, \Gamma] \Rightarrow \bigvee[\Sigma, \Gamma] \\
\bigvee[\Sigma, \bigvee[\Gamma], \Pi] \Rightarrow \bigvee[\Sigma, \Gamma, \Pi] & \bigvee[\Sigma, \varphi, \varphi, \Pi] \Rightarrow \bigvee[\Sigma, \varphi, \Pi] & \bigvee[\Sigma, \phi, \varphi, \Pi] \overset{\varphi<\phi}{\Longrightarrow} \bigvee[\Sigma, \varphi, \phi, \Pi]
\end{array}
$$

Note: We do not convert the formulas to conjunctive normal form or disjunctive normal form here, as doing so may substantially increase the size of the formula.

4.1.2 Operations on pLTL formulas

4.1.2.1 Rewrite under sets

Since the semantics of rewriting under sets is setting the subformulas in the set to weak, and we have labeled past subformulas with ids, it is possible to represent past subformula sets with bit sets.

For example, say we have $\phi = \mathbf{Y}(p\mathbf{S}(q\mathbf{B}r))$, and we labeled $q\mathbf{B}r$ as 0, $p\mathbf{S}(q\mathbf{B}r)$ as 1 and $\mathbf{Y}(p\mathbf{S}(q\mathbf{B}r))$ as 2. For the past subformula set $\{q\mathbf{B}r, \mathbf{Y}(p\mathbf{S}(q\mathbf{B}r))\}$, we represent it as $0b101$, and for set $p\mathbf{S}(q\mathbf{B}r)$ it's $0b010$.

When performing rewriting, one can simply traverse the pLTL syntax tree and directly set the weak/strong state to the corresponding bit. Continuing with the example, suppose we want to rewrite $\mathbf{Y}(p\mathbf{S}(q\mathbf{B}r))$ with $\{q\mathbf{B}r, \mathbf{Y}(p\mathbf{S}(q\mathbf{B}r))\}$. We begin at the root, the first past subformula we found is the past subformula with label 2, and since it is in the set (0b101), it should be rewritten to $\tilde{\mathbf{Y}}$. Next, for $p\mathbf{S}(q\mathbf{B}r)$, whose label is 1, it is not present in the set, so it is strengthened; as it is already in its strengthened form, no change occurs. We then encounter $q\mathbf{B}r$, whose label 0 is in the set, and it is subsequently weakened. Finally, the result obtained is $\tilde{\mathbf{Y}}(p\mathbf{S}(q\tilde{\mathbf{B}}r))$.

4.1.2.2 M/N Rewrite

Similar to rewriting under sets, we simply traverse the pLTL syntax tree. If a syntax tree node is rooted with $\mathbf{U}, \mathbf{M}$ or $\mathbf{W}, \mathbf{R}$, it would be set to $\top$, $\bot$, or its strength-representing bit would be adjusted accordingly.

4.1.2.3 (local) after function

During automaton construction, it is common to calculate (local) after functions with varying input letters or past subformula set. Additionally, some subformulas' (local) after functions are utilized repeatedly in the construction of different automata.

An apparent solution involves implementing a cache that maps a tuple (pLTL formula, letter, weaken state set) to the corresponding local after function result. However, when the original pLTL formula is large, such a cache structure tends to consume a significant amount of memory and may require extended time for both building and querying. This limitation restricts the size of pLTL formulas that can be processed by the program.

Notice that when calculating the local after function of a pLTL formula, it may be necessary to compute the local after function for its subformulas. If this is done by strictly following the definition of the local after function as described in the theory section, even if certain letters are not used in a subformula, the corresponding result must still be calculated.

For example, when calculating the after function for:

$$\varphi = \neg(g_0 \wedge g_1) \wedge \mathbf{G}(\mathbf{F}(\neg r_0 \tilde{\mathbf{S}} g_0)) \wedge \mathbf{G}(\mathbf{F}(\neg r_1 \tilde{\mathbf{S}} g_1))$$

under all 16 possible input σ (from $0b0000$ to $0b1111$), we will have to disjunct $\mathrm{af}_l(\varphi, \sigma, C)$ for all 4 possible past subformula sets (which can be $\{\}, \{\neg r_0 \tilde{\mathbf{S}} g_0\}$, $\{\neg r_1 \tilde{\mathbf{S}} g_1\}, \{\neg r_0 \tilde{\mathbf{S}} g_0, \neg r_1 \tilde{\mathbf{S}} g_1\}$). Thus, we are required to compute the local after

function on φ under 64 distinct settings. However, recalling the definition of the local after function, one part of calculating $\mathrm{af}_l(\varphi, \sigma, C)$ involves computing $\mathrm{af}_l(\neg(g_0 \wedge g_1), \sigma, C)$. It may be noticed that:

- The local after function value of this subformula depends on only two variables.
- This subformula does not contain any past subformulas; therefore, the selection of C does not affect its local after function value.

This implies that there are actually only four meaningful settings for this particular subformula. It is not necessary to perform calculations for all 64 settings.

To optimize the performance of (local) after function calculations in such cases, a specialized cache data structure was designed.

The fundamental idea is to build a two-level hash map. The first level maps each formula to a `CacheItem` entry, which contains:

- A mask indicating which variables are present in the formula.
- A mask indicating which past subformulas are present in the formula.
- A hash map from `(variable IDs, past subformula IDs)` to the af_l result.

Building the cache for a past formula follows the algorithm described in the pseudocode in A.2.1. This process involves calculating af_l for every subformula of φ in a bottom-up manner.

When calculating $\mathrm{af}_l(\varphi, \sigma, C)$, the `build_local_past_function_cache` function is first applied to φ to construct a cache for all its subformulas.

Then, by querying for (σ&`atom_mask`, C&`past_st_mask`) (accessed via the `CacheItem`'s get method), the desired result is directly retrieved. Furthermore, even if $\mathrm{af}_l(\varphi, \sigma, C)$ must be computed for every possible σ and C pair, the cache needs to be built only once.

For the previous example, let's say we assign IDs for variables as follows:

Variable	**id**
g_0	0
g_1	1
r_0	2
r_1	3

And past subformulas as follows:

Past subformula	id
$\{\neg r_0 \tilde{\mathbf{S}} g_0\}$	0
$\{\neg r_1 \tilde{\mathbf{S}} g_1\}$	1

Then, in our solution, the cache structure might appear as follows [1]:

Formula	Variable Mask	Past subformula Mask	Variable Values	C set	afl Result
$\neg(g_0 \wedge g_1)$	0b0011	0b00	0b0000	0b00	$\top$
			0b0001	0b00	$\top$
			0b0010	0b00	$\top$
			0b0011	0b00	$\bot$
$\mathbf{G}(\mathbf{F}(\neg r_0 \tilde{\mathbf{S}} g_0))$	0b0101	0b01	0b0000	0b00	$\mathbf{G}(\mathbf{F}(\neg r_0 \mathbf{S} g_0)))$
			0b0001	0b00	$\mathbf{G}(\mathbf{F}(\neg r_0 \mathbf{S} g_0)))$
			0b0100	0b00	$\mathbf{F}(\neg r_0 \mathbf{S} g_0) \wedge \mathbf{G}(\mathbf{F}(\neg r_0 \mathbf{S} g_0)))$
			0b0101	0b00	$\mathbf{G}(\mathbf{F}(\neg r_0 \tilde{\mathbf{S}} g_0)))$
			0b0000	0b01	$\mathbf{G}(\mathbf{F}(\neg r_0 \tilde{\mathbf{S}} g_0)))$
			0b0001	0b01	$\mathbf{G}(\mathbf{F}(\neg r_0 \tilde{\mathbf{S}} g_0)))$
			0b0100	0b01	$\bot$
			0b0101	0b01	$\mathbf{G}(\mathbf{F}(\neg r_0 \tilde{\mathbf{S}} g_0)))$
...					

Say if we are calculating $\mathrm{af}_l(\mathbf{G}(\mathbf{F}(\neg r_0 \tilde{\mathbf{S}} g_0)), \{g_0, g_1, r_1\}, \{\neg r_0 \tilde{\mathbf{S}} g_0, \neg r_1 \tilde{\mathbf{S}} g_1\})$ (which $\sigma = 0b1011$, $C = 0b11$), we just query the entry $(0b1011 \& 0b0101, 0b11 \& 0b01) = (0b0001, 0b01)$, and the result $\mathbf{G}(\mathbf{F}(\neg r_0 \mathbf{S} g_0)$ is just what we want.

4.1.3 $M\langle C_i \rangle$

A optimization similar to the one applied in calculating the local after function, is also utilized in the pre-calculation of $M\langle C_i \rangle$, which is employed during the construction of stable automata.

It is observed that, for a given M, the values of $M\langle C_i \rangle$ sets depend solely on those C_i sets whose elements are all part of one of the subformulas within M.

Therefore, instead of rewriting all M with all C_i:

- For any μ subformula ψ, its past subformula set C_j is calculated. All such j's can be stored in a bitset mask, referred to as J. Note that for any ψ,

[1] Note that in the actual implementation, **G** and **F** operators in formulas are represented using **U** and **W** with $\top$ or $\bot$ as one of their operands; for example, $\mathbf{G}(\mathbf{F}(\neg r_0 \mathbf{S} g_0)))$ is actually $(\top \mathbf{U}(\neg r_0 \mathbf{S} g_0))\mathbf{W}\bot$. We use **G** and **F** here purely for simplicity.

$\psi\langle C_i\rangle = \psi\langle C_i \& J\rangle$, which allows us to cache only the $\psi\langle C_j\rangle$ results.

- We can construct $M\langle C_i\rangle$s with all $\psi\langle C_i\rangle$s. Consider a scenario where $M_0\langle C_i\rangle$ is already known. We can then get $(M_0 \cup \{\psi\})\langle C_i\rangle$, where $\psi \notin M_0$, by:
  - If $M_0\langle C_i\rangle$'s bit mask is J_{C_0} and ψ's bit mask is J_ψ, then $(M_0 \cup \{\psi\})\langle C_i\rangle$'s bit mask becomes $J_{C_0}|J_\psi$. This indicates that the C items "in" ψ are taken into consideration.
  - $(M_0 \cup \{\psi\})\langle C_i\rangle = M_0\langle C_i\rangle \cup \{\psi\}\langle C_i\rangle = M_0\langle J_{C_0} \& C_i\rangle \cup \{\psi\}\langle J_\psi \& C_i\rangle$. Consequently, we can cache only those results where C_i is a subset of $J_{C_0}|J_\psi$, and these results can be readily computed using the known $M_0\langle C_i\rangle$ and $\psi\langle C_i\rangle$.

For example, when transforming the pLTL formula $\mathbf{F}(p\mathbf{S}q) \vee \mathbf{F}(r\mathbf{B}s)$, past subformulas are labeled as follows:

Past subformula	**id**
$p\mathbf{S}q$	0
$r\mathbf{B}s$	1

And μ subformulas are:

Past subformula	**id**
$\mathbf{F}(p\mathbf{S}q)$	0
$\mathbf{F}(r\mathbf{B}s)$	1

First, we calculate the $\psi\langle C_j\rangle$s, which are also the results for single-element M sets:

μ subformula (only element in M set)	Past subformula mask	C set	$\psi\langle C_i\rangle$ result (only element in $M\langle C_i\rangle$ result)
$\mathbf{F}$(p **S** q)	0b01	0b00	$\mathbf{F}$(p **S** q)
		0b01	$\mathbf{F}(p\tilde{\mathbf{S}}q)$
$\mathbf{F}$(r **B** s)	0b10	0b00	$\mathbf{F}$(r **B** s)
		0b10	$\mathbf{F}(r\tilde{\mathbf{B}}s)$

Then combine them to form the results for M sets containing more than one element.

M set	Past subformula mask	C set	$M\langle C_i\rangle$ result
$\{\mathbf{F}(p\mathbf{S}q), \mathbf{F}(p\mathbf{B}q)\}$	0b11	0b00	$\{\mathbf{F}(p\mathbf{S}q), \mathbf{F}(r\mathbf{B}s)\}$
		0b01	$\{\mathbf{F}(p\tilde{\mathbf{S}}q), \mathbf{F}(r\mathbf{B}s)\}$
		0b10	$\{\mathbf{F}(p\mathbf{S}q), \mathbf{F}(r\tilde{\mathbf{B}}s)\}$
		0b11	$\{\mathbf{F}(p\tilde{\mathbf{S}}q), \mathbf{F}(r\tilde{\mathbf{B}}s)\}$

Here, for example, the result for $C = \texttt{0b01}$ is calculated using the formula:

$$\begin{aligned}
&(\{\mathbf{F}(p\mathbf{S}q)\} \cup \{\mathbf{F}(r\mathbf{B}s)\})\langle C_{\texttt{0b01}}\rangle \\
=&\{\mathbf{F}(p\mathbf{S}q)\}\langle C_{\texttt{0b01}}\rangle \cup \{\mathbf{F}(r\mathbf{B}s)\}\langle C_{\texttt{0b01}}\rangle \\
=&\{\mathbf{F}(p\mathbf{S}q)\}\langle \texttt{0b01\&0b01}\rangle \cup \{\mathbf{F}(r\mathbf{B}s)\}\langle \texttt{0b01\&0b10}\rangle \\
=&\{\mathbf{F}(p\mathbf{S}q)\}\langle \texttt{0b01}\rangle \cup \{\mathbf{F}(r\mathbf{B}s)\}\langle \texttt{0b00}\rangle \\
=&\{\mathbf{F}(p\tilde{\mathbf{S}}q), \mathbf{F}(r\mathbf{B}s)\}
\end{aligned}$$

4.1.4 Saturated C sets

Recall that in the construction of the weakening conditions automaton, it is required to compute saturated C sets:

$$J_i := \{j \in [1..k] \mid \forall \xi, \xi' \in \mathrm{psf}(\varphi).\xi\langle C_j\rangle = \xi'\langle C_j\rangle \Rightarrow \xi\langle C_i\rangle = \xi'\langle C_i\rangle\}$$

Calculating J_i by definition can be slow, especially when there are many past subformulas.

It has been observed that if two distinct past subformulas can become equivalent under certain rewrites, it implies that they share the same "shape," differing only in the strength of their operators.

For instance, the past subformulas that might become equivalent under some rewrites for $\mathbf{Y}(p\mathbf{S}q)$ include $\mathbf{Y}(p\tilde{\mathbf{S}}q)$, $\tilde{\mathbf{Y}}(p\mathbf{S}q)$, and $\tilde{\mathbf{Y}}(p\tilde{\mathbf{S}}q)$.

If no past subformula of the same "shape" as ξ exists within $\mathrm{psf}(\varphi)$, then there would also never exist a ξ' that could be equal to ξ under any C, and this kind of check can be safely skipped.

To further reduce the computational workload, a bitmask (denoted as $mask_k$ for the k-th pair) is used to indicate which pairs of past subformulas share the same "shape." For each such pair, a bitmask can be constructed. If, for any $mask_k$, the expression $C_j \& mask_k$ is neither equal to the mask (meaning both subformulas in the pair are present) nor equal to 0 (meaning neither is present), then the condition $\xi\langle C_j\rangle = \xi'\langle C_j\rangle$ would be false, and the actual rewrite and comparison can be safely skipped[2].

[2]However, if $C_j \& mask_k$ equals the mask or 0, it does not necessarily mean that $\xi\langle C_j\rangle = \xi'\langle C_j\rangle$, since there may be subformulas within ξ.

For example, consider $\varphi = \mathbf{Y}(p\mathbf{S}q) \wedge \tilde{\mathbf{Y}}(p\tilde{\mathbf{S}}q) \wedge (p\mathbf{B}q)$, and assume the past subformulas are labeled as follows:

Past subformula	**id**
$p\mathbf{S}q$	0
$\mathbf{Y}(p\mathbf{S}q)$	1
$p\tilde{\mathbf{S}}q$	2
$\tilde{\mathbf{Y}}(p\tilde{\mathbf{S}}q)$	3
$p\tilde{\mathbf{B}}q$	4

Then, the bitmask for the pair $(p\mathbf{S}q, p\tilde{\mathbf{S}}q)$ is $mask_0 = \texttt{0b00101}$, and for $(\mathbf{Y}(p\mathbf{S}q), \tilde{\mathbf{Y}}(p\tilde{\mathbf{S}}q))$ it is $mask_1 = \texttt{0b01010}$.

To check whether $C_i = \{pSq\}$ (`0b00001`) is saturated with respect to $C_j = \{\tilde{\mathbf{Y}}(p\tilde{\mathbf{S}}q)\}$ (`0b01000`):

- For $mask_1$, since $C_j \& mask_1$ is neither equal to the mask nor 0, this check can be skipped.
- For $mask_0$, note that $p\mathbf{S}q\langle C_j\rangle$ is equal to $p\tilde{\mathbf{S}}q\langle C_j\rangle$, but $p\mathbf{S}q\langle C_i\rangle$ is not equal to $p\tilde{\mathbf{S}}q\langle C_i\rangle$, so j should not be included in J_i.

In the other direction, to check whether $C_i = \{\tilde{\mathbf{Y}}(p\tilde{\mathbf{S}}q)\}$ (`0b001000`) is saturated with respect to $C_j = \{pSq\}$ (`0b000001`):

- For $mask_0$, since $C_j \& mask_0$ is neither equal to the mask nor 0, the check can be skipped.
- For $mask_1$, since $Y(pSq)\langle C_j\rangle$ is not equal to $\tilde{Y}(p\tilde{S}q)\langle C_j\rangle$, the check can also be skipped.

As there are no violations, j is included in J_i in this case.

4.1.5 Automata representation & construction

All guarantee, safety, and stable automata, along with the weakening conditions automata, are internally represented as a struct containing the following:

- A **transition map** from the current state to a vector of states. The index, which can be interpreted as a bit set, represents an input word.
- An **initial state**.
- A **prediction function**, which accepts a state and returns whether it is an accepting state. For weakening conditions automata, which do not have an accepting state, this field can always return false.
- An **optional accepting type**, indicating whether it is a Büchi or co-Büchi automaton. For weakening conditions automata, this field can be `None`.

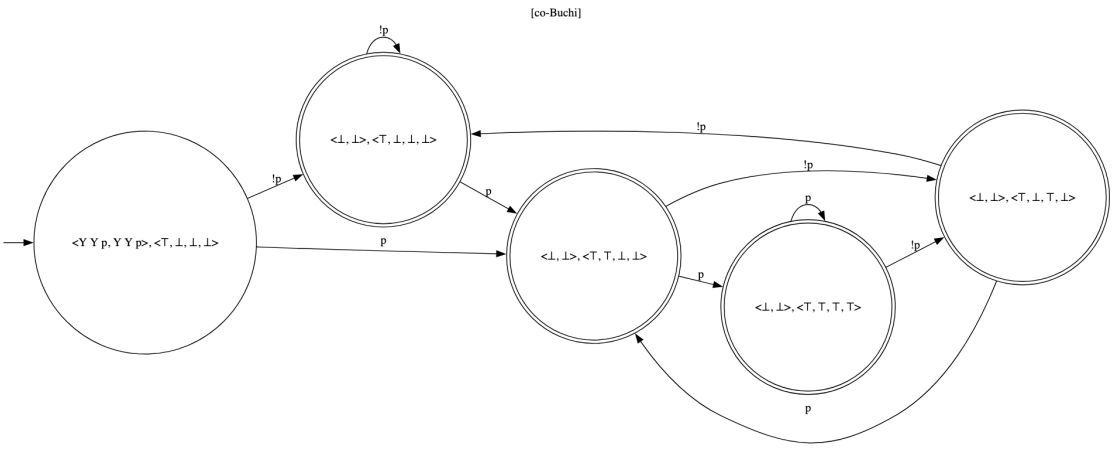
After the program generates these automata, they will be exported in HOA [2] format with a Makefile. Subsequently, running `make` will utilize Spot [8] to merge them (by calculating the product and co-product of these automata) to generate the resulting automaton.

4.1.5.1 Merging sink states in stable automata

Recall the transition function of the stable automata:

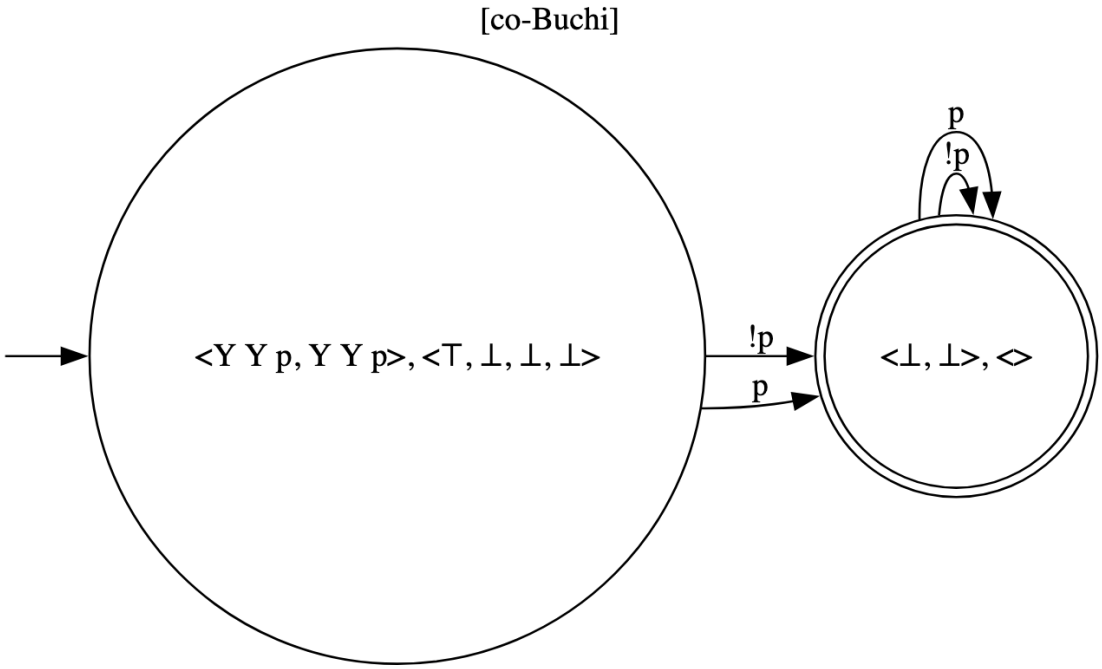
$$\delta(\langle\psi,\zeta\rangle,\xi^{\times},\sigma) := \begin{cases} \langle \mathrm{af}(\psi,\sigma), \bigvee_{i\in[1..k]} \mathrm{af}(\psi,\sigma)[M\langle C_i\rangle]_\nu \wedge \xi_i[M\langle C_i\rangle]_\nu\rangle & (\zeta \sim \bot) \\ \langle \mathrm{af}(\psi,\sigma), \mathrm{af}(\zeta,\sigma)\rangle & (\text{otherwise}) \end{cases}$$

According to this definition, many states may exist in which the $\langle\psi,\zeta\rangle$ component is identical, while the corresponding $\xi^{\times}$ values differ. For example, in the case of $\mathbf{YY}p$:



In this example, four states have $\langle \bot, \bot \rangle$ as their $\langle \psi, \zeta \rangle$ component. As $\mathrm{af}(\bot, \sigma)$ consistently evaluates to $\bot$, and $\bot[M]_\nu \wedge \cdots$ also evaluates to $\bot$, these states invariably transition to another state which first part is $\langle \bot, \bot \rangle$. In other words, such states cannot be exited to a state with a different $\langle \psi, \zeta \rangle$ structure, regardless of the values in $\xi^\times$. Therefore, all states for which the $\langle \psi, \zeta \rangle$ component is $\langle \bot, \bot \rangle$ can be safely merged into a single sink state.

For the given example, this optimization yields:



As you may have seen, this optimization greatly reduces the size of the automata.

Similarly, states where the $\langle \psi, \zeta \rangle$ component is $\langle \top, \top \rangle$ have similar behavior. Such states will always transition to other states where the $\langle \psi, \zeta \rangle$ component is also $\langle \top, \top \rangle$, and therefore can also be merged into a single state.

4.1.5.2 Reusing the Same Guarantee and Safety Automata

The algorithm described in [1] requires the construction of one safety automaton for each pair of $\psi \in \nu(\varphi)$ and $M \subseteq \mu(\varphi)$.

Recall the definition of the transition function of these automata:

$$\delta(\zeta, \xi^\times, \sigma) := \begin{cases} \bigvee_{i \in [1..k]} \mathbf{G}(\psi \langle C_i \rangle [M \langle C_i \rangle]_\nu) \wedge \xi_i [M \langle C_i \rangle]_\nu & (\zeta \sim \bot) \\ \mathrm{af}(\zeta, \sigma) & (\text{otherwise}) \end{cases}$$

,

and their initial state $Q_0 := \mathbf{G}(\psi[M]_\nu)$.

It is clear that, among all elements in M, only those that are also subformulas of ψ affect the resulting automaton.

Each item in $\mu(\varphi)$ can be assigned an id, and the set M can be represented as a bitset. A mask-based method, similar to the one used when caching the local after function, can then be employed to compute only what is necessary.

For example, given $\varphi = \mathbf{G}(\mathbf{F}(p)) \vee \mathbf{G}(\mathbf{F}(q))$ and $\psi = \mathbf{G}(\mathbf{F}(p))$, the resulting automaton is the same for both $M = \{\}$ and $M = \{\mathbf{F}(q)\}$.

If the items in $\mu(\varphi)$ are labeled as follows:

$\mu(\varphi)$	**id**
$\mathbf{F}(p)$	0
$\mathbf{F}(q)$	1

Then, a cache structure can be constructed as follows:

$\nu(\varphi)$	**id**	Mask (Contains $\mu(\varphi)$ set)	M set	Automata
$\mathbf{G}(\mathbf{F}(p))$	0	0b01	0b00 ($\{\}$)	A_0
			0b01 ($\{\mathbf{F}(p)\}$)	A_1
$\mathbf{G}(\mathbf{F}(q))$	1	0b10	0b00 ($\{\}$)	A_2
			0b10 ($\{\mathbf{F}(q)\}$)	A_3

To obtain the automaton corresponding to $\psi = \mathbf{G}(\mathbf{F}(p))$ and $M = \{\mathbf{F}(q)\}$ (represented as `0b10`), we first compute the bitwise AND with its mask: `0b10 & 0b01 = 0b00`. Then, the result at the corresponding position, which is A_0, is used. This is the same as the case where $M = \{\}$, consistent with the result obtained previously.

A similar optimization can also be applied to the guarantee automata.

4.1.6 Parallelization

We utilize rayon [4], a data-parallelism library, to parallelize several calculations. Parallelizable tasks are created for situations including, but not limited to, the following:

- Each pair of ψ and M sets is assigned to a separate task during the construction of the safety automaton.
- Each pair of ψ and N sets is assigned to a separate task during the construction of the guarantee automaton.
- When evaluating transition functions, the calculation for each letter is performed in its own task.

- During the construction of the local after function cache, upon encountering an $\wedge$ or $\vee$ operator, each child node is forked into a separate task.

Although these partitions are based on rough estimates, aiming to ensure tasks are neither too small nor too large, the CPU utilization achieved in practice is quite satisfactory.

For cache structures, such as the cache for the local after function, concurrent hashmaps and read-write locks are employed to prevent data races. The type system in Rust significantly aids in ensuring this correctness.

Besides, the `make` process can also be parallelized by passing `-j<thread count>` argument.

4.1.7 Testing

To ensure the tool produces correct results, it is tested against `pLTL2NGBA`. This tool constructs a Büchi automaton directly from a pLTL formula using an algorithm similar to the one implemented in our work. `pLTL2NGBA` is provided by its author as Haskell source code, and its construction algorithm is detailed by David Lidell et al. [7], a subsequent work to [1] that reuses many of its constructions.

For testing, we utilize both all benchmark cases (provided conversion time does not exceed 30 minutes) and randomly generated pLTL formulas. Once both tools complete the conversion and save the results as HOA files, the command `autfilt -equivalent-to` from Spot [8] is used to verify the equivalence of the resulting automata.

4.1.8 Design Choices and Alternatives

In this section we will list out some of our most important the design choices we have made, possible alternatives and the reason we choose current option.

Although these choices were made by the authors based on encountered situations and personal experience, they often align with existing industry practices due to the similarity of problems encountered when dealing with (p)LTL and ω-automata.

4.1.8.1 Atom propositions and atom proposition set representation

We represent atoms as integers and atom sets as bitsets. The approach of using integers for atom propositions aligns with many other LTL and ω-automata tools, such as Spot [8].

We used a fixed maximum bitset size of 32 for two key reasons:

- This allows us to represent the bitset as a plain 32-bit unsigned integer, which significantly accelerates operations like calculating intersections, unions, and subsets, checking element membership, and adding/removing items.

- We can use a simple array of type `T` to represent a map from an atom proposition set to `T`, directly using the bitset itself as an index.

The main limitation of this design is that it is not able to handle pLTL formulas with more than 32 atom propositions. However, given the algorithm's high time and space complexity, the current implementation is unlikely to process pLTL formulas with such a large number of atom propositions efficiently anyway.

In the future, we might consider using Binary Decision Diagrams (BDDs) to represent the atom proposition set. This would, however, leads to a symbolic conversion algorithm, which is out of the scope of this thesis.

4.1.8.2 ID for temporal subformulas

We assign a unique ID to each temporal subformula. For past subformulas, this ID is embedded within the Abstract Syntax Tree (AST), whereas for future subformulas, the ID is kept external. Here's why:

When constructing the automata, past subformulas are frequently rewritten. When checking if a past subformula exists within a set of other past subformulas, they are often rewritten into identical weak/strong states. By keeping their IDs within the AST, we can represent the past subformula sets here as bitsets, which significantly aids in efficient checks.

Conversely, all rewriting of future subformulas can be completed and cached before the automata construction even begins. This pre-calculation means there's no need to keep their IDs within the AST; we can simply perform the necessary computations and then forget the IDs.

4.1.8.3 Semi-normal form

We compare pLTL formulas using their semi-normal form to approximate semantic equivalence, rather than the BDD method proposed by Azzopardi et al. [1]. Our primary reason for this approach is that rewriting into semi-normal form is simpler to implement and avoids adding extra complexity, such as the need for a BDD operating library, data structures, or algorithms for converting pLTL formulas to BDDs.

Whether this was the best decision is still unclear, and it is an area we've identified for future investigation.

Existing tools like Spot [8] have also proposed methods for simplifying LTL formulas, some of which overlap with the rules we've presented here. Implementing all of these methods (or determining their necessity) is also a potential area for future work.

4.2 Evaluating and Benchmarking

4.2.1 Data Collection

A set of pLTL formulas has been designed to benchmark the performance of our implementation. Some of these formulas are collected or summarized from real-world problems, while others are synthetic formulas specifically generated to evaluate the implementation under certain conditions.

4.2.1.1 Generated Data

As part of our implementation, a pLTL formula generator has also been developed. This enables the generation of pLTL formulas with varying numbers of atomic propositions and operators, providing a controlled method to test the algorithm's performance across different complexity levels.

4.2.1.2 Real-world Data

The complete list of real-world pLTL formulas can be found in the appendix.

In these formulas:

- A.1 to A.8 are from [15], which has also been utilized in [24]. These pLTLs are small to medium-sized (0 to 10 unique temporal subformulas).
- A.9 to A.20 were summarized by the authors themselves. These pLTLs are small-sized (fewer than 5 temporal subformulas).
- A.21 to A.23 are real-world pLTL formulas that model an arbiter; the resulting automata can be directly used for model checking and controller synthesis. They range from medium-sized (6 temporal subformulas) to large-sized (18 temporal subformulas).

4.2.2 Benchmark and Data Collection

For each pLTL formula, the following performance data has been collected:

- Time consumed for automaton construction.
- Memory consumed for automaton construction.

4.2.3 Data Analysis

4.2.3.1 Scaling Comparison

To assess the algorithm's scalability, its performance (execution time and memory usage) hasbeen measured across pLTL formulas of varying sizes. The performance curve has been plotted.

4.2.3.2 Benchmarking Against Existing Tools

To compare the proposed algorithm with alternative approaches, their performance will be evaluated on a fixed dataset. The performance data will be compared across these three methods:

- Our algorithm and its implementation.
- Constructing a (non-deterministic) Büchi automaton using the `pLTL2BA` tool in GOAL [28], followed by determinization with GOAL's determinization tool.
- Constructing a Büchi automaton directly using `pLTL2NGBA`, followed by determinization using the determinization tool in Spot [8].

5

Results

We have completed a working implementation of the tool, named `pltl2dra`, which is published on GitHub at https://github.com/longfangsong/pltl/. An online demo is also available for viewing here: https://longfangsong.github.io/pltl/.

5.1 Performance Evaluation

5.1.1 Scalability Analysis

We use several specific sets of pLTL formulas to evaluate how the implementation scales with the size of these formulas.

These sets include:

- $p_0 \mathbf{U} p_1 \wedge \cdots \wedge p_m \mathbf{U} p_{m+1}$, where m ranges from 0 to 7.
  This set is used to test the scalability of the algorithm when it contains only future operators.

- $q_0 \mathbf{S} q_1 \wedge \cdots \wedge q_m \mathbf{S} q_{m+1}$, where m ranges from 0 to 7.
  This set is used to test the scalability of the algorithm when it contains only past operators.

- $p_0 \mathbf{U} p_1 \wedge \cdots \wedge p_m \mathbf{U} p_{m+1} \wedge q_0 \mathbf{S} q_1 \wedge \cdots \wedge q_n \mathbf{S} q_{n+1}$, where m and n range from 0 to 4.
  This set combines the characteristics of the previous two. We will analyze these results together to provide a two-dimensional evaluation of the implementation's scalability.

- $\mathbf{G}(\mathbf{Y}(p_0)) \wedge \cdots \wedge \mathbf{G}(\mathbf{Y}(p_m))$, where m ranges from 0 to 4.
  This set is used to evaluate how the implementation scales with subformulas that contain both past and future operators.

We also measure the time consumption for a specific pLTL formula (A. 23) using an increasing number of threads. This demonstrates how our program scales with the utilization of additional CPU resources.

We will measure the execution time and memory consumption of `pltl2dra` and `make` (which calls tools in Spot[8] to combine the sub-automata generated by `pltl2dra` into a single deterministic Rabin automaton)[1].

We will present the Benchmark resultss first, followed by a description of the findings from these results.

The following results were obtained from benchmarking on an Apple M2 Max, featuring 12 CPU cores (8 performance and 4 efficiency) and 64GB of memory.

The program is compiled using the Rust toolchain version `nightly-2025-04-22-aarch64-apple-darwin`, with all default optimizations enabled. The specific version of the code utilized is commit `f2afabb`.

The Spot version used for merging is 2.13.

Note that there is some missing data in the tables below, denoted by `-`. This means that the tool failed to accomplish the task within a reasonable time (exceeding 30 minutes).

We ran the benchmarks five times but only kept one value in the following result table because the variation between different runs was fairly small (less than 2%).

[1]The sub-automata generated by `pltl2dra` are still useful without combination, as one may check whether an ω-word satisfies all of the automata's acceptance conditions and merge the results logically.

5.1.1.1 Future only

Table 5.1: Benchmark results for $p_0 \mathbf{U} p_1 \wedge \cdots \wedge p_m \mathbf{U} p_{m+1}$

m		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
pltl2dra	Time(s)	0.003	0.003	0.005	0.011	0.14	0.858	5.587	35.234
	Mem(MB)	3.888	4.496	5.328	8.608	46.928	243.024	2253.256	13688.096
make	Time(s)	0.022	0.043	0.303	19.633	-	-	-	-
	Mem(MB)	18.432	34.392	166.352	7518.864	-	-	-	-

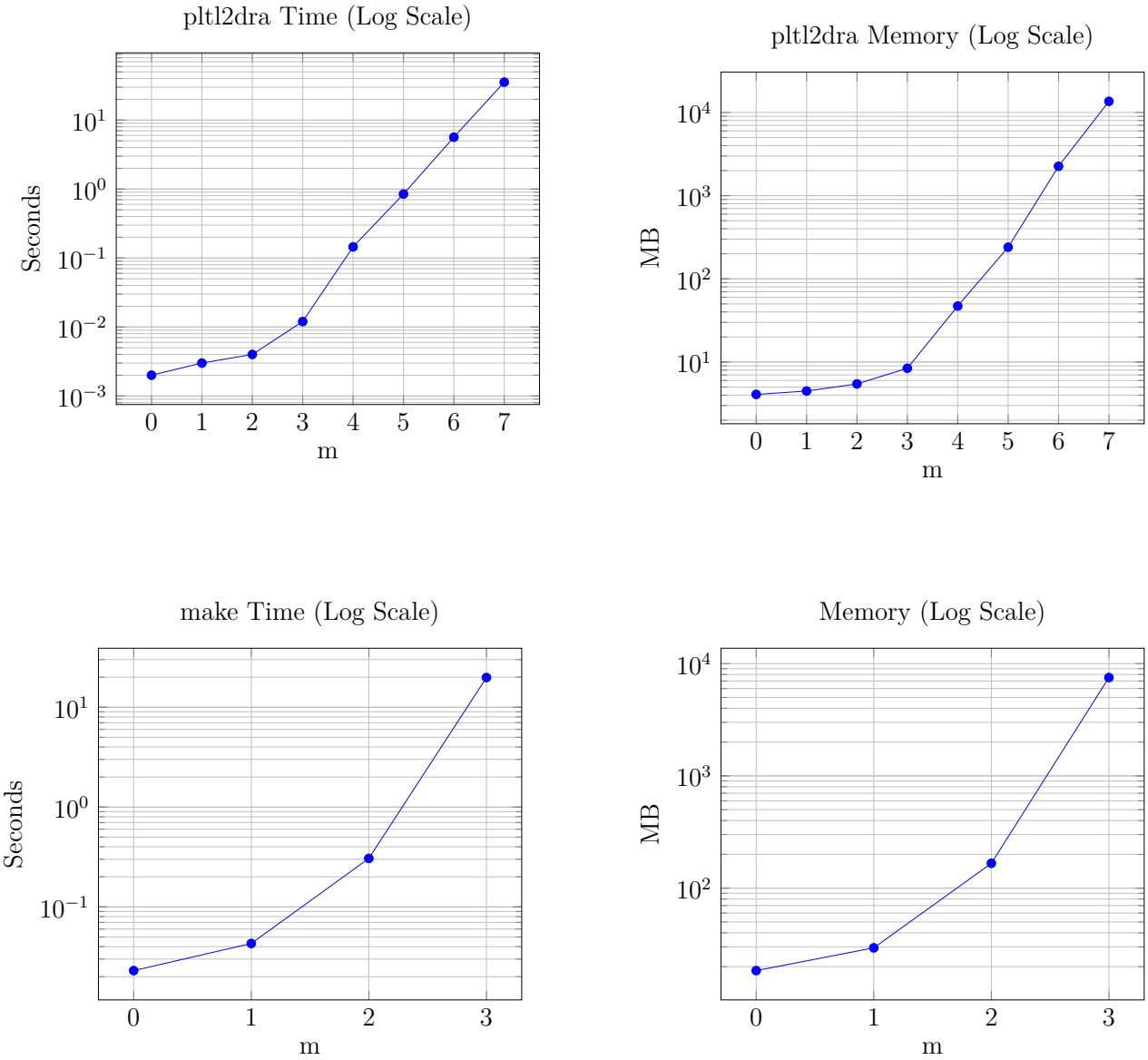


Figure 5.1: Benchmark results for $p_0 \mathbf{U} p_1 \wedge \cdots \wedge p_m \mathbf{U} p_{m+1}$

5.1.1.2 Past only

Table 5.2: Benchmark results for $p_0 \mathbf{S} p_1 \wedge \cdots \wedge p_m \mathbf{S} p_{m+1}$

m		**0**	**1**	**2**	**3**	**4**	**5**	**6**
pltl2dra	Time(s)	0.002	0.003	0.007	0.007	0.078	1.142	15.223
	Mem(MB)	3.792	4.336	4.912	4.944	5.872	10.928	40.272
make	Time(s)	0.016	0.017	0.022	0.018	0.025	0.023	0.028
	Mem(MB)	17.6	17.824	18.704	19.552	19.184	21.152	28.272

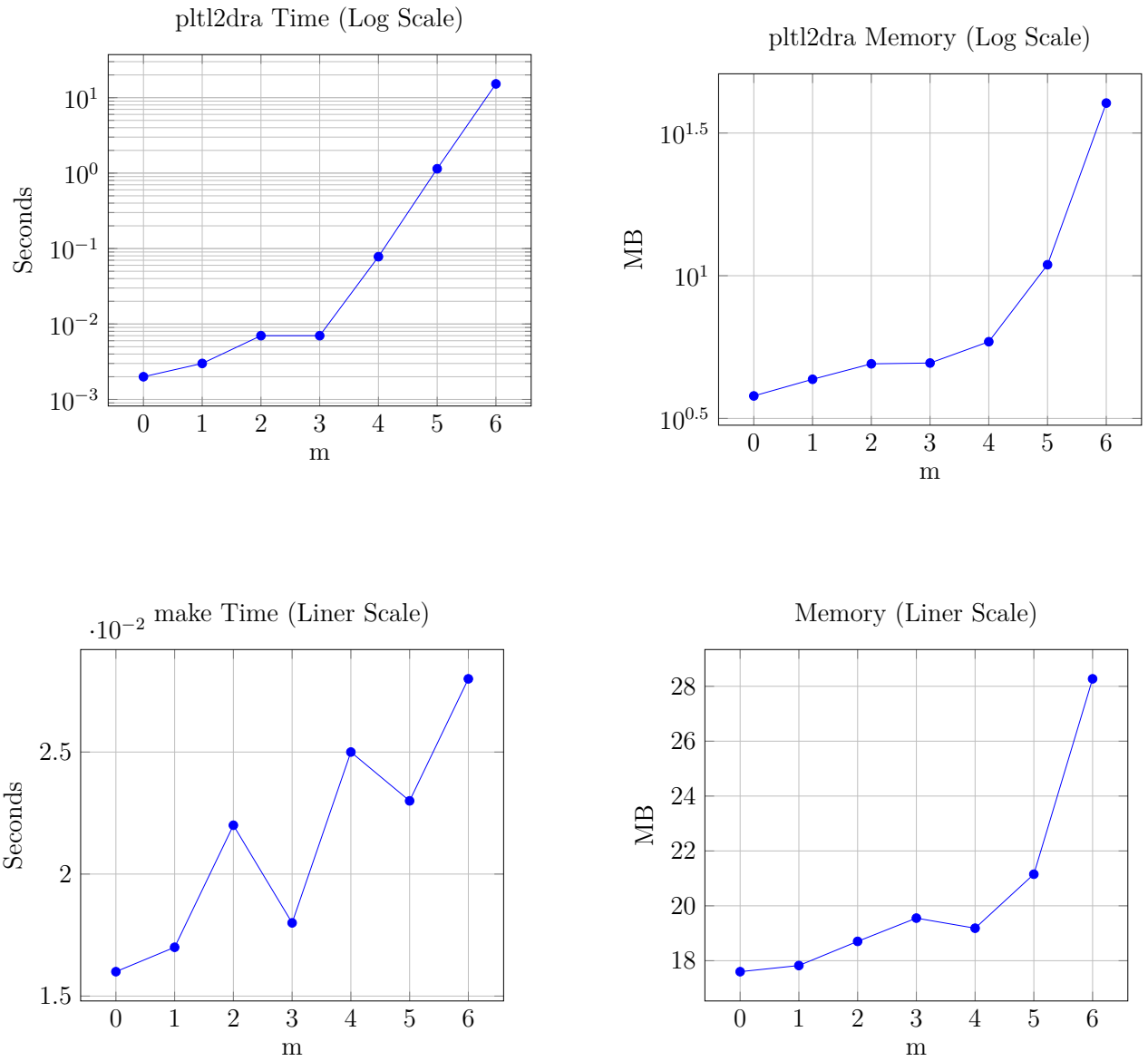


Figure 5.2: Benchmark results for $p_0 \mathbf{S} p_1 \wedge \cdots \wedge p_m \mathbf{S} p_{m+1}$

5.1.1.3 Past and Future, separated

Table 5.3: Benchmark results for $p_0\mathbf{U}p_1 \wedge \cdots \wedge p_m\mathbf{U}p_{m+1} \wedge q_0\mathbf{S}q_1 \wedge \cdots \wedge q_n\mathbf{S}q_{n+1}$, pltl2dra Execution time (seconds)

Past \ Future	0	1	2	3	4
0	0.002	0.002	0.003	0.004	0.012
1	0.002	0.003	0.005	0.014	0.075
2	0.003	0.005	0.011	0.053	0.295
3	0.007	0.03	0.074	0.269	1.283
4	0.007	0.366	0.835	2.151	6.878

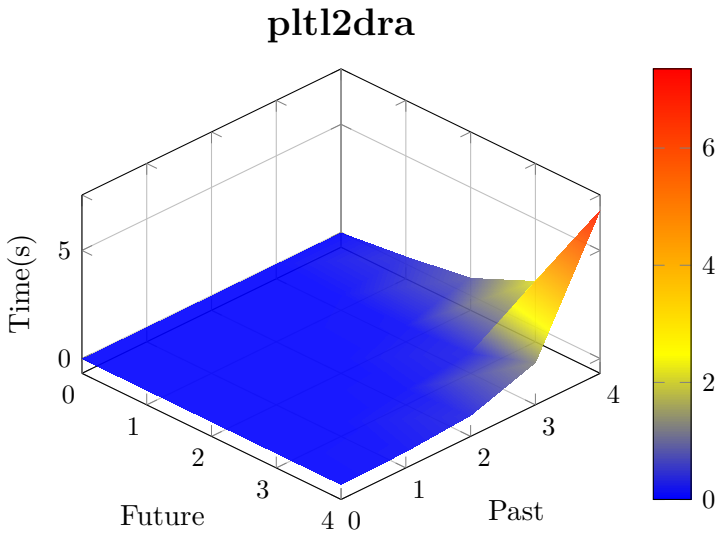


Figure 5.3: Benchmark results for $p_0\mathbf{U}p_1 \wedge \cdots \wedge p_m\mathbf{U}p_{m+1} \wedge q_0\mathbf{S}q_1 \wedge \cdots \wedge q_n\mathbf{S}q_{n+1}$, pltl2dra Execution time (seconds)

Table 5.4: Benchmark results for $p_0\mathbf{U}p_1 \wedge \cdots \wedge p_m\mathbf{U}p_{m+1} \wedge q_0\mathbf{S}q_1 \wedge \cdots \wedge q_n\mathbf{S}q_{n+1}$, Memory consumption (MB)

Past \ Future	0	1	2	3	4
0	3.2	4.08	4.48	5.456	8.416
1	3.792	4.56	5.648	9.248	27.248
2	4.336	5.376	8.064	20.064	87.84
3	4.912	7.712	17.664	65.008	486.192
4	4.944	20.16	66.592	337.168	1817.424

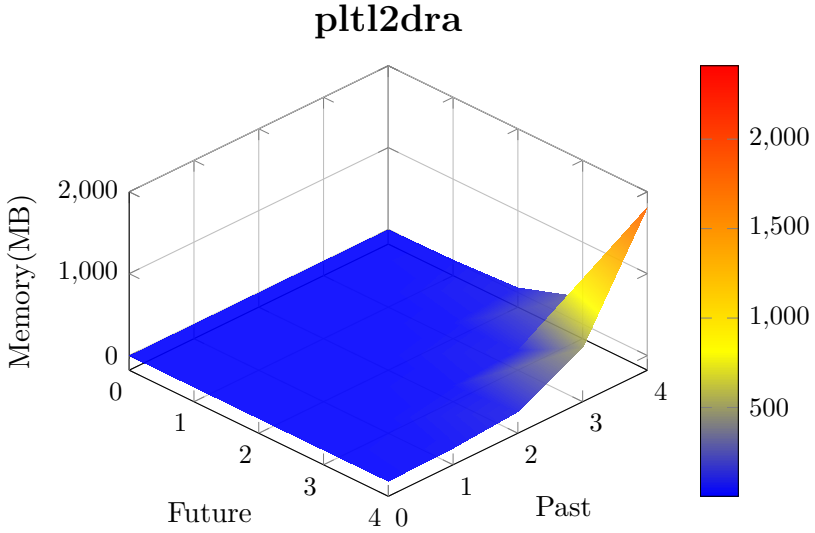


Figure 5.4: Benchmark results for $p_0\mathbf{U}p_1 \wedge \cdots \wedge p_m\mathbf{U}p_{m+1} \wedge q_0\mathbf{S}q_1 \wedge \cdots \wedge q_n\mathbf{S}q_{n+1}$, Memory consumption (MB)

Table 5.5: Benchmark results for $p_0\mathbf{U}p_1 \wedge \cdots \wedge p_m\mathbf{U}p_{m+1} \wedge q_0\mathbf{S}q_1 \wedge \cdots \wedge q_n\mathbf{S}q_{n+1}$, make execution Time (s)

Past \ Future	0	1	2	3	4
0	0.015	0.023	0.043	0.306	19.821
1	0.016	0.032	0.164	2.284	167.53
2	0.017	0.067	0.62	9.75	-
3	0.022	0.209	2.433	42.973	-
4	0.018	0.784	10.804	200.31	-

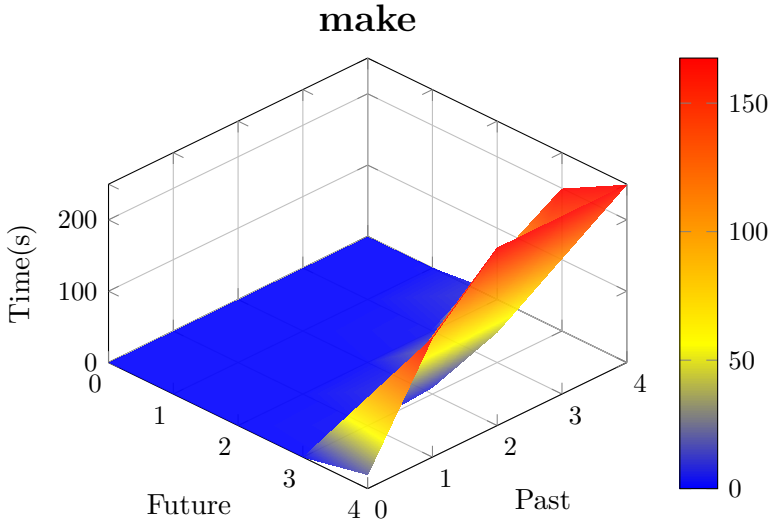


Figure 5.5: Benchmark results for $p_0\mathbf{U}p_1 \wedge \cdots \wedge p_m\mathbf{U}p_{m+1} \wedge q_0\mathbf{S}q_1 \wedge \cdots \wedge q_n\mathbf{S}q_{n+1}$, make execution Time (s)

Table 5.6: Benchmark results for $p_0\mathbf{U}p_1 \wedge \cdots \wedge p_m\mathbf{U}p_{m+1} \wedge q_0\mathbf{S}q_1 \wedge \cdots \wedge q_n\mathbf{S}q_{n+1}$, make Memory consumption (MB)

Past \ Future	0	1	2	3	4
0	16.864	18.432	29.392	166.352	7512.864
1	17.6	28.208	121.968	1290.912	37318.992
2	17.824	59.824	472.48	5084.88	-
3	18.704	190.752	1759.232	12374.208	-
4	19.552	645.84	6374.32	41049.52	-

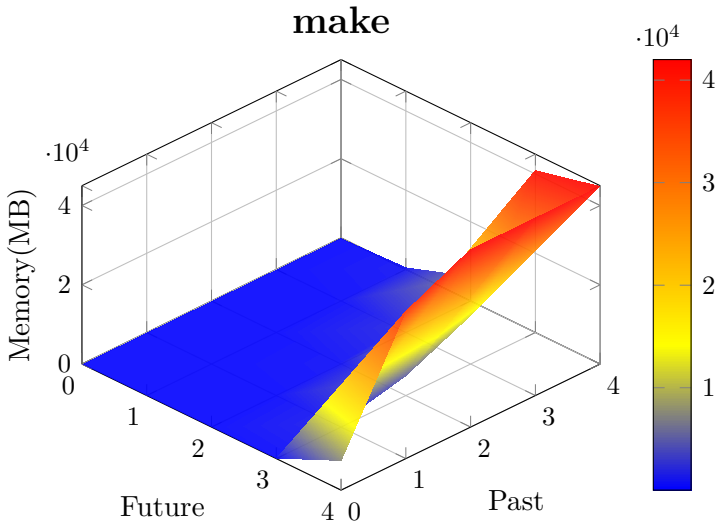


Figure 5.6: Benchmark results for $p_0\mathbf{U}p_1 \wedge \cdots \wedge p_m\mathbf{U}p_{m+1} \wedge q_0\mathbf{S}q_1 \wedge \cdots \wedge q_n\mathbf{S}q_{n+1}$, make Memory consumption (MB)

5.1.1.4 Past and Future, combined

Table 5.7: Benchmark results for $\mathbf{G}(\mathbf{Y}(p_0)) \wedge \cdots \wedge \mathbf{G}(\mathbf{Y}(p_m))$

		1	**2**	**3**	**4**	**5**	**6**	**7**
Time(s)	pltl2dra	0.003	0.003	0.006	0.039	0.507	6.486	86
	make	0.019	0.027	0.095	0.796	10.634	147.93	-
Memory(MB)	pltl2dra	3.936	4.576	5.456	8.672	27.760	215.392	1782.416
	make	17.344	20.464	46.624	371.984	2818.688	14001.504	-

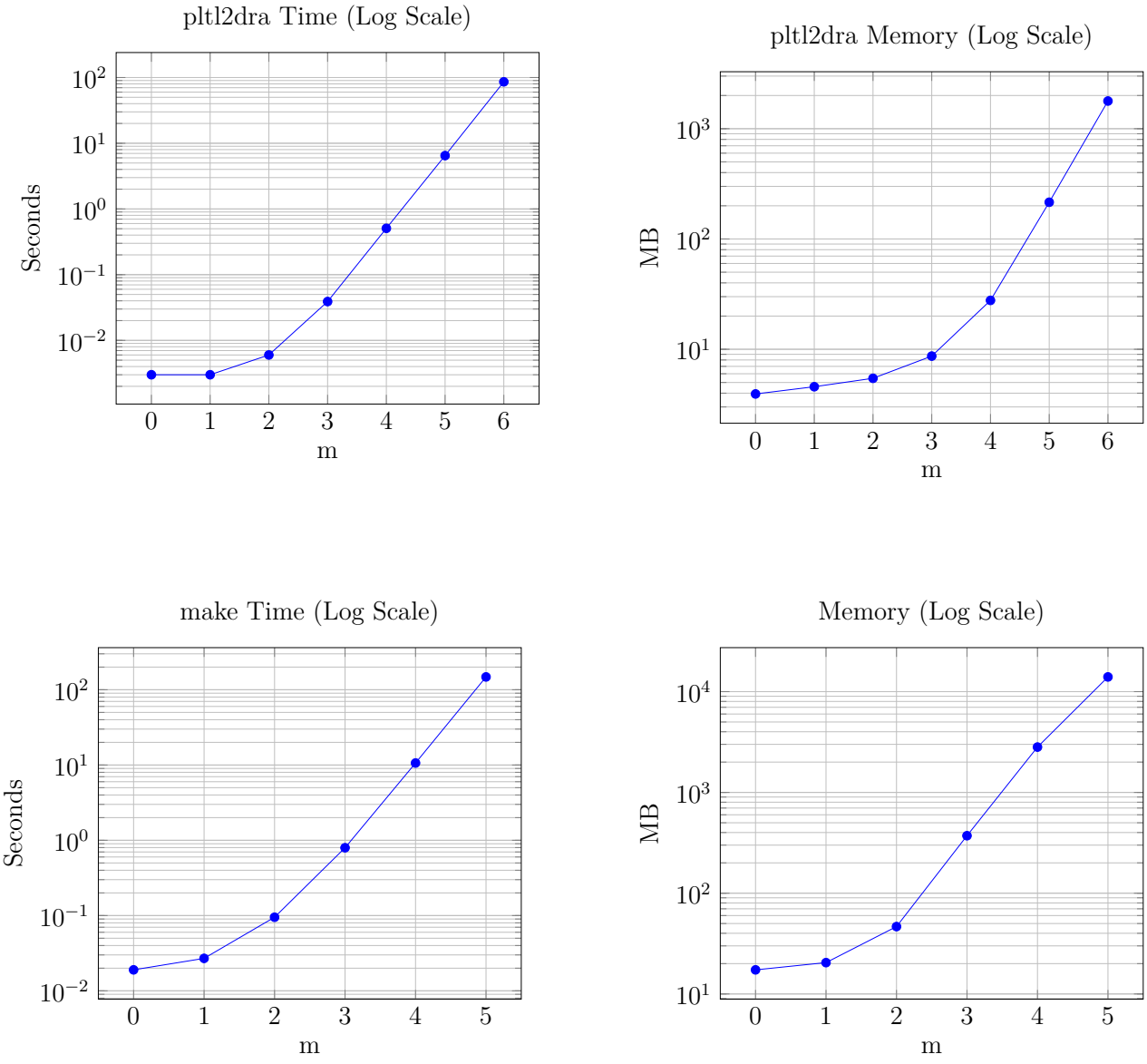


Figure 5.7: Benchmark results for $\mathbf{G}(\mathbf{Y}(p_0)) \wedge \cdots \wedge \mathbf{G}(\mathbf{Y}(p_m))$

5.1.1.5 Increasing number of threads

Table 5.8: Benchmark results for pLTL formula A.23, with increasing threads

Threads/CPUs	1	2	3	4	5	6	7	8
Time(s)	153.46	94.11	69.1	52.084	46.28	44.804	42.698	36.669

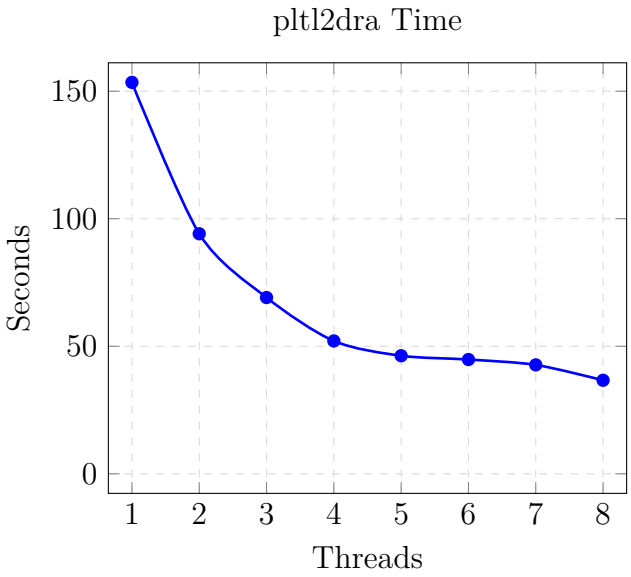


Figure 5.8: Benchmark results for pLTL formula A.23, with increasing threads

5.1.1.6 Findings

From the result of benchmarking the program, we can deduce that:

- The program's scalability largely aligns with the time and space complexity predictions of its underlying algorithm, as described in [1]. This indicates that the program's scalability increases double exponentially with respect to both past and future operator counts.

- Consider the evaluation result of past and future only pLTLs. Though the program itself can handle future formula more efficient than past formula, it takes much more time and space to merge the generated sub-automata into the final deteministic rabin automata. It is because of the count of sub-automata for merging together is directly related to the future subformula count, but not related to the past subformula count directly. You can also draw a similar conclusion by looking at the diagrams generated by plotting the make commands' time and space consumption of seperated past and future formula. They are steeper on the future-axis than the past-axis.

- Merging sub-automata generated by `pltl2dra` typically consumes considerably more time and memory. The only exception is when the pLTL contains solely past operators, which generate fewer automata.

- Increasing the number of threads generally reduces execution time; however, performance gains become less substantial beyond four threads. This could be attributed to memory access speed, suggesting an area for future benchmarking and optimizations.

These conclusions answer **RQ2**: Our implementation's complexity largely follows that of the algorithm itself, exhibiting double exponential time and space complexity relative to the count of temporal operators.

It's worth noting that the time and space consumed by merging sub-automata with make and Spot typically exceeds that spent on their construction. This observation provides a clear direction for future optimization.

5.1.2 Comparative Benchmarking

The hardware settings are the same as the scalability analysis.

We used the same version of our implementation and Spot as in the scalability analysis.

For `goal`, we use version 2020-05-06.

The pLTL formulas we used can be found in Appendix A; the results are presented on the next page.

From the benchmarking results, we can observe the following:

- For smaller formulas, the tool itself runs quickly and is memory-efficient in most cases. However, the `make` step may consume slightly more time and memory.
- In some special cases (e.g., for pLTL case 3 and 4), the tool is slower and less memory-efficient than `pLTL2NGBA`. These pLTL formulas often contain numerous conjuncted subformulas, which can trigger the algorithm's exponential time and space complexity.
- For larger pLTL formulas, this tool tends to be more efficient compared to other tools. In certain instances (e.g., pLTL case 23), it is even the only tool capable of accomplishing the task within a reasonable timeframe.

These conclusions provide an answer to **RQ1**: this implementation's performance is comparable to other tools for small pLTL formulas, and it generally performs much better for larger ones. This also implies better scalability for this implementation compared to other tools.

Table 5.9: Automata building time in seconds

-	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
pltl2dra	0.004	0.003	0.335	0.76	0.004	0.002	0.003	0.002	0.002	0.005	0.006	0.002	0.003	0.003	0.003	0.002	0.002	0.003	0.003	0.002	0.004	0.244	43.68
make	0.017	0.015	0.014	0.015	0.053	0.015	0.014	0.014	0.02	0.068	0.121	0.02	0.021	0.02	0.029	0.024	0.019	0.026	0.021	0.019	0.039	0.611	44.891
sum	0.021	0.018	0.349	0.775	0.057	0.017	0.017	0.016	0.022	0.073	0.127	0.022	0.024	0.023	0.032	0.026	0.021	0.029	0.024	0.021	0.043	0.855	88.571
pLTL2TNGBA	0.017	0.017	0.016	0.029	0.014	0.017	0.015	0.017	0.017	0.015	0.016	0.014	0.017	0.017	0.017	0.017	0.017	0.017	0.017	0.017	0.017	5.065	-
autfilt	0.006	0.005	0.004	0.005	0.005	0.004	0.004	0.004	0.005	0.008	0.012	0.004	0.007	0.005	0.005	0.005	0.005	0.007	0.005	0.005	0.006	0.049	-
sum	0.023	0.022	0.02	0.034	0.019	0.021	0.019	0.021	0.022	0.023	0.028	0.018	0.024	0.022	0.022	0.022	0.022	0.024	0.022	0.022	0.023	5.114	-
goal	3.34	1.03	1.081	0.945	12.512	0.791	0.815	0.784	0.811	1.092	0.975	0.811	0.884	0.83	0.891	0.845	0.814	0.889	0.841	0.813	1.09	-	-
determinize	1.044	0.858	1.02	0.894	6.567	0.781	0.82	0.786	0.805	0.957	0.909	0.795	0.833	0.828	0.844	0.858	0.8	0.862	0.828	0.789	0.791	-	-
sum	4.384	1.888	2.101	1.839	19.079	1.572	1.635	1.57	1.616	2.049	1.884	1.606	1.717	1.658	1.735	1.703	1.614	1.751	1.669	1.602	1.881	-	-

Table 5.10: Automata building's memory consumption, in MB

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
pltl2dra	4.3	4.1	7.1	12.5	5.1	3.8	4.5	3.7	4.3	5.7	6.2	4.4	4.7	4.6	5.3	4.2	4.2	4.9	4.6	4.1	5.8	80.0	5417.0
make	17.9	18.1	17.6	17.8	40.0	17.6	17.6	17.6	17.9	50.9	61.3	18.2	19.1	19.0	21.1	21.8	17.8	19.2	19.1	17.9	21.3	75.4	2257.6
pLTL2TNGBA	12.7	12.6	12.6	12.8	12.6	12.6	12.6	12.6	12.6	12.8	12.9	12.6	12.7	12.6	12.6	12.6	12.6	12.6	12.6	12.6	12.9	19.1	-
autfilt	17.4	20.1	19.4	17.1	16.9	19.4	17.1	17.1	19.4	17.5	19.7	17.1	17.1	17.1	17.4	19.6	19.6	17.6	19.6	17.1	17.4	21.3	-
goal	1311.0	306.5	328.8	215.9	1321.1	163.4	167.6	162.2	166.7	265.5	211.6	165.9	200.9	166.7	195.1	169.6	167.0	195.2	167.1	166.5	420.6	-	-
determinization	223.8	166.7	234.5	198.0	2171.6	167.3	173.0	161.1	167.3	194.7	196.6	169.7	164.8	167.1	171.5	182.7	165.0	172.1	165.8	167.1	163.7	-	-

6

Discussion And Conclusion

This thesis implemented the algorithm described in [1] and investigate whether it could improve the performance of translating pLTL to a deterministic Rabin automaton.

Following the implementation and evaluation, we are now able to answer the questions posed earlier:

RQ1: How efficient is the implementation of Azzopardi et al.'s algorithm [1] compared to existing tools?

For most small-sized real-world pLTL formulas, the implementation performs at least as well as existing tools. Furthermore, for some larger pLTL formulas, this tool is the only known one capable of performing the conversion.

However, due to the algorithm's double exponential complexity concerning the total number of temporal operators, and since the generated subformulas for merging depend on the future operator count, this tool can become less time and memory efficient with too many future operators. The bottleneck lies in merging the generated sub-automata.

RQ2: Does the implementation scale well with the size of the pLTL formulas?

Although the algorithm itself has double-exponential complexity, by applying significant optimizations and leveraging the algorithm's inherent parallelizability, the growth of the actual time and space used by the implementation are controlled to a reasonable level.

For past-only pLTL formulas, this implementation scales significantly better than for other types of pLTL formulas. This is likely because increasing the count of past subformulas does not increase the number of subautomata generated. Since the bottleneck in our conversion process is typically the subautomata combining stage, a lower increase in subautomata count leads to less overall performance decrement.

For the remaining cases, generally speaking, both the time and space used in conversion follow a double-exponential complexity. We can also observe that the subautomata combining stage scales less efficiently. This indicates a clear direction for

future optimization.

In a nutshell, this implementation is ideal when a pLTL specification requires a deterministic ω-automaton. It's particularly more performant and memory-efficient for large pLTL formulas with over 10 temporal operators. We also expect further performance gains after planned future optimizations.

6.1 Limitations and Delimitations

- The experimental evaluation was performed using a predefined set of pLTL formulas and was conducted under controlled laboratory conditions. The selection of these formulas, while intended to be representative, may not fully reflect the diversity of real-world applications, impacting the generalizability of the study's conclusions.

  This has been mitigated by ensuring that the formula set is as diverse as possible, both structurally (for example, diverse in operator and atom count) and semantically (for example, by collecting pLTL coming from different fields).

- The comparison with existing tools is limited to a subset of widely used temporal logic tools. The choice of tools and benchmarks may not fully encompass all use cases, potentially limiting the scope of the performance comparison.

  This has been mitigated by searching a very broad range of existing temporal logic tools. Even if some comparable tools are overlooked, the findings regarding the scalability of this specific algorithm/tool remain relevant.

6.2 Future Work

Currently, there remain several possible avenues for improving the program:

- The final sub-automata combining stage often consumes more time than the conversion stage itself.

  Multiple optimization strategies could be explored, such as

  - Pruning the merge tree by identifying obviously never-accepting or always-accepting automata
  - Calling the Spot[8] library functions for merging via a foreign function interface instead of relying on `make` and command line interface tools
  - Implementing our own high-efficiency algorithm for calculating the product and sum of automata.

- The implementation could benefit from further optimization. Some design choices made during implementation were adopted without definitive knowledge of their optimality.

  For instance,

  - It might be possible to enhance (but also possible to degrade) the performance of pLTL equivalence checking by utilizing the BDD-based algorithm introduced in [1] instead of the current semi-normal form based approach.
  - It can also be beneficial to introduce a symbolic algorithm into the construction process. For example, many transitions can be merged if their requirements are symbolically resolved, rather than being evaluated for every possible letter combination. This approach is expected to significantly improve performance, especially in cases involving a large number of letters.
  - If the formula is in shape $\varphi \wedge \phi$ or $\varphi \vee \phi$, and φ contains future formula only, it could be faster to calculate this part with traditional LTL-to-automata methods (by executing Spot functions directly for example), and merge the result with applying the current implemented on ϕ.
  - Further engineering optimizations, such as reducing shared resources to enhance parallelism performance gains.

- To address the limitations of current evaluation, we can benchmark with more pLTL formulas found in industry.

Bibliography

[1] Shaun Azzopardi, David Lidell, and Nir Piterman. A Direct Translation from LTL with Past to Deterministic Rabin Automata. In Rastislav Královič and Antonín Kučera, editors, *49th International Symposium on Mathematical Foundations of Computer Science (MFCS 2024)*, volume 306 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:15, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[2] Tomáš Babiak, František Blahoudek, Alexandre Duret-Lutz, Joachim Klein, Jan Křetínskỳ, David Müller, David Parker, and Jan Strejček. The hanoi omega-automata format. In *International Conference on Computer Aided Verification*, pages 479–486. Springer, 2015.

[3] J Richard Buchi and Lawrence H Landweber. Solving sequential conditions by finite-state strategies. In *The Collected Works of J. Richard Büchi*, pages 525–541. Springer, 1990.

[4] P.W.D. Charles. Rayon: A data parallelism library for rust. `https://github.com/rayon-rs/rayon`.

[5] Geoffroy Couprie. Nom, a byte oriented, streaming, zero copy, parser combinators library in rust. In *2015 IEEE Security and Privacy Workshops*, pages 142–148. IEEE, 2015.

[6] Costas Courcoubetis and Mihalis Yannakakis. Verifying temporal properties of finite-state probabilistic programs. In *[Proceedings 1988] 29th Annual Symposium on Foundations of Computer Science*, pages 338–345. IEEE Computer Society, 1988.

[7] N. Piterman D. Lidell. Translations from ltl with past to büchi automata. 2025.

[8] Alexandre Duret-Lutz, Etienne Renault, Maximilien Colange, Florian Renkin, Alexandre Gbaguidi Aisse, Philipp Schlehuber-Caissier, Thomas Medioni, Antoine Martin, Jérôme Dubois, Clément Gillard, and Henrich Lauko. From Spot 2.0 to Spot 2.10: What's new? In *Proceedings of the 34th International Conference on Computer Aided Verification (CAV'22)*, volume 13372 of *Lecture Notes in Computer Science*, pages 174–187. Springer, August 2022.

[9] Javier Esparza and Jan Křetínskỳ. From ltl to deterministic automata: A safraless compositional approach. In *International Conference on Computer Aided Verification*, pages 192–208. Springer, 2014.

[10] Berndt Farwer. ω-automata. In *Automata logics, and infinite games: a guide to current research*, pages 3–21. Springer, 2002.

[11] LTL Fast. Fast ltl to büchi automata translation. 2013.

[12] Paul Gastin and Denis Oddoux. Fast ltl to büchi automata translation. In *Computer Aided Verification: 13th International Conference, CAV 2001 Paris, France, July 18–22, 2001 Proceedings 13*, pages 53–65. Springer, 2001.

[13] Paul Gastin and Denis Oddoux. Ltl with past and two-way very-weak alternating automata. volume 2747, pages 439–448, 08 2003.

[14] Paul Gastin and Denis Oddoux. Ltl with past and two-way very-weak alternating automata. In *International Symposium on Mathematical Foundations of Computer Science*, pages 439–448. Springer, 2003.

[15] Constance Heitmeyer, Dino Mandrioli, et al. *Formal methods for real-time computing*. John Wiley & Sons, 1996.

[16] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[17] Gerard J Holzmann and Doron Peled. The state of spin. In *Computer Aided Verification: 8th International Conference, CAV'96 New Brunswick, NJ, USA, July 31–August 3, 1996 Proceedings 8*, pages 383–389. Springer, 1996.

[18] Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 97–107, 1985.

[19] Orna Lichtenstein, Amir Pnueli, and Lenore Zuck. *The glory of the past*. Springer, 1985.

[20] Nicolas Markey. Temporal logic with past is exponentially more succinct. *Bulletin-European Association for Theoretical Computer Science*, 79:122–128, 2003.

[21] Robert McNaughton. J. richard büchi. weak second-order arithmetic and finite automata. zeitschrift für mathematische logik und grundlagen der mathematik, vol. 6 (1960), pp. 66–92.-j. richard büchi. on a decision method in restricted second order arithmetic. logic, methodology and philosophy of science, proceedings of the 1960 international congress, edited by ernest nagel, patrick suppes, and alfred tarski, stanford university press, stanford, calif., 1962, pp. 1–11. *The*

Journal of Symbolic Logic, 28(1):100–102, 1963.

[22] Thibaud Michaud and Maximilien Colange. Reactive synthesis from ltl specification with spot. In *Proceedings of the 7th Workshop on Synthesis, SYNT@ CAV 2018*, 2018.

[23] Amir Pnueli. The temporal logic of programs. In *18th annual symposium on foundations of computer science (sfcs 1977)*, pages 46–57. ieee, 1977.

[24] Matteo Pradella, Pierluigi San Pietro, Paola Spoletini, Angelo Morzenti, et al. Practical model checking of ltl with past. In *ATVA03: 1st Workshop on Automated Technology for Verification and Analysis*, 2003.

[25] Roni Rosner. *Modular synthesis of reactive systems*. The Weizmann Institute of Science (Israel), 1991.

[26] Shmuel Safra. On the complexity of!-automata. In *Proc. 29th IEEE Symp. Found. of Comp. Sci*, pages 319–327, 1988.

[27] Wolfgang Thomas. *Automata, logics, and infinite games: A guide to current research*, volume 2500. Springer Science & Business Media, 2002.

[28] Yih-Kuen Tsay, Yu-Fang Chen, Ming-Hsien Tsai, Wen-Chin Chan, and Chi-Jian Luo. Goal extended: Towards a research tool for omega automata and temporal logic. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 346–350. Springer, 2008.

[29] Moshe Y Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *1st Symposium in Logic in Computer Science (LICS)*. IEEE Computer Society, 1986.

A

Appendix

A.1 pLTL collection in real world

A.1.1 Standard railroad crossing problem

This part of pLTLs are from [15], which has also been used in [24].

For pLTLs which only difference is the atom names, we keep only first of them.

Note we continue to use these abbreviation introduced in [24].

- $\mathbf{X}^1\phi \leftrightarrow \mathbf{X}\phi, \mathbf{X}^k\phi \leftrightarrow \mathbf{X}\mathbf{X}^{k-1}\phi$ ($\mathrm{Y}^k$is its past analogous);
- $\mathbf{G}_{<k}\phi \leftrightarrow \mathbf{X}\phi \wedge \mathbf{X}^2\phi \wedge \cdots \wedge \mathbf{X}^{k-1}\phi$ ($\mathbf{H}_{<k}$ is its past analogous);
- $\mathbf{F}_{<k}\phi \leftrightarrow \neg\mathbf{G}_{<k}\neg\phi$ ($\mathbf{O}_{<k}$is its past analogous).

There are the 8 formulars:

$$\text{enterR} \rightarrow \mathbf{G}_{<\mu}\text{enterR} \tag{A.1}$$

$$\text{enterR} \rightarrow \mathbf{X}^{\text{dm}}(\text{enterI} \vee \mathbf{F}_{<\gamma}\text{enterI}) \tag{A.2}$$

$$\text{enterI} \rightarrow \mathbf{Y}^{\text{dm}}(\text{enterR} \wedge \mathbf{O}_{<\gamma}\text{enterR}) \tag{A.3}$$

$$\text{inR} \leftrightarrow (\text{enterR} \wedge \mathbf{O}_{<1+dM}\text{enterR}) \wedge (\neg\text{enterI}\ \mathbf{S}\ \text{enterR}) \tag{A.4}$$

$$\mathbf{Y}\text{closed} \wedge \text{go(up)} \rightarrow \text{mvUp} \wedge \mathbf{G}_{<\gamma}\text{mvUp} \wedge \mathbf{X}^{\gamma}((\text{open}\ \mathbf{U}\ \text{go(down)}) \vee \mathbf{G}\text{open}) \tag{A.5}$$

$$\mathbf{H}\neg\text{go(down)} \wedge \neg\text{go(down)} \rightarrow \text{open} \tag{A.6}$$

$$\text{go(down)} \leftrightarrow \mathbf{Y}^{\text{dm}-\gamma}\text{enterR} \tag{A.7}$$

$$\text{go(up)} \leftrightarrow \text{exitI} \tag{A.8}$$

We take $\mu = 10$, $d_M = 5$, $d_m = 4$ and $\gamma = 2$.

A.1.2 Summarized by us

These pLTLs are summarized by the authors themselves, from the fields like database, parallel programming, etc.

There are the 12 formulas:

$$\mathbf{G}(\text{commit} \rightarrow \neg\mathbf{O}(\text{abort})) \tag{A.9}$$

$$\mathbf{G}(\text{start} \rightarrow (\mathbf{F}(\text{commit} \ \vee \text{abort}) \wedge (\text{abort} \rightarrow \mathbf{O}\ \text{conflict}))) \tag{A.10}$$

Means: every transaction must be committed or aborted at last, and if it got aborted, there should be a conflict in the past.

$$\mathbf{G}(\text{acquire} \rightarrow (\mathbf{F}(\text{inCS} \wedge \mathbf{F}\text{release}) \wedge (\text{inCS} \rightarrow \neg\text{release}\ \mathbf{S}\ \text{acquire}))) \tag{A.11}$$

Means: every transaction must be committed or aborted at last, and if it got aborted, there should be a conflict in the past

$$\mathbf{G}(\text{enterCS} \rightarrow \mathbf{Y}(\neg\text{mutex})) \tag{A.12}$$

If enter critical section at this time, then there should be nobody holding the mutex at last time stamp.

$$\mathbf{G}(\text{grant} \rightarrow \mathbf{O}(\text{request} \wedge (\neg\text{cancel}\ \mathbf{S}\ \text{request}))) \tag{A.13}$$

Grant a resource to a user only when it is once requested and not canceled.

$$\mathbf{G}((\text{pressureHigh}\ \mathbf{S}\ \text{sysCheck}) \rightarrow \text{alert}) \tag{A.14}$$

If pressure is high since system check last time, then trigger an alert.

$$\mathbf{G}(\mathbf{O}(\text{sendAttempt} \ \wedge \ \neg\text{ack}) \rightarrow \mathbf{F}(\text{retransmit})) \tag{A.15}$$

If a message is sent and don't have an ack, then we should eventually retransmit it.

$$(\text{alert} \rightarrow \mathbf{F}(\text{writeLog})) \wedge (\text{writeLog} \rightarrow (\text{alert}\ \mathbf{S}\ \text{alertEvent})) \tag{A.16}$$

If an alert is triggered, we'll eventually write it into log, and if we are writing a log, an alert should be holding since an event triggered

$$\mathbf{G}(\text{decide}(P, d) \rightarrow \mathbf{O}(\text{propose}(\mathit{any}, d))) \tag{A.17}$$

node P decides the value of key if only if any node proposed it.

$$\mathbf{FG}(\text{read} \rightarrow ((\neg\text{writeConflicting})\ \mathbf{S}\ \text{writeStable})) \tag{A.18}$$

Eventually all reads are latest writes (Eventual Consistency), once we reach "steady state," every read must be backed by a past stable write with no conflicting writes in the interval.

$$\mathbf{G}(\text{read}(v) \wedge \mathbf{Y}(\text{read}(v')) \rightarrow (v \geq v')) \tag{A.19}$$

v is a monotonic counter, it won't decrease between readings

$$\mathbf{G}(\text{moveForward} \rightarrow \mathbf{H}(\neg\text{collisionDetected})) \tag{A.20}$$

if the robot tries to move forward now, it must never have detected a collision at any past time.

A.1.3 Arbiter Specification

Say that we have n clients that can request access to a resource.

Client i requests access at a given point in time is corresponds to the variable r_i being true at that time.

The arbiter grants access to client i corresponds to the variable g_i being true.

The system requires:

- Only one client should be given access at any given point in time.
- Every request made should eventually be followed by a grant.
- No grants should be given unless there's an open request.

We can express these three requirements in pLTL by the formula:

$$\bigwedge_{\substack{i\in[1..n]\\ j\in[1..n]\\ i\neq j}} \neg(g_i \wedge g_j) \wedge \bigwedge_{i\in[1..n]} \mathbf{GF}(\neg r_i \tilde{\mathbf{S}} g_i) \wedge \bigwedge_{i\in[1..n]} \mathbf{G}(g_i \Rightarrow r_i \vee \mathbf{Y}(r_i \mathbf{B} \neg g_i))$$

We will take n=1, 2 and 3 for our benchmarking[1]:

$$\mathbf{G}(\mathbf{F}(\neg r_0 \tilde{\mathbf{S}} g_0)) \wedge \mathbf{G}(g_0 \rightarrow (r_0 \vee \mathbf{Y}(r_0 \mathbf{B} \neg g_0))) \tag{A.21}$$

$$\begin{aligned} &\neg(g_0 \wedge g_1) \wedge \neg(g_1 \wedge g_0) \wedge \\ &\mathbf{G}(\mathbf{F}(\neg r_0 \tilde{\mathbf{S}} g_0)) \wedge \mathbf{G}(\mathbf{F}(\neg r_1 \tilde{\mathbf{S}} g_1)) \wedge \\ &\mathbf{G}(g_0 \rightarrow (r_0 \vee \mathbf{Y}(r_0 \mathbf{B} \neg g_0))) \wedge \mathbf{G}(g_1 \rightarrow (r_1 \vee \mathbf{Y}(r_1 \mathbf{B} \neg g_1))) \end{aligned} \tag{A.22}$$

[1]The n=1 case does not make much sense in the real world, but is useful to show how the programs scales

$$\neg(g_0 \wedge g_1) \wedge \neg(g_0 \wedge g_2) \wedge \neg(g_1 \wedge g_0) \wedge \neg(g_1 \wedge g_2) \wedge \neg(g_2 \wedge g_0) \wedge \neg(g_2 \wedge g_1) \wedge$$
$$\mathbf{G}(\mathbf{F}(\neg r_0 \tilde{\mathbf{S}} g_0)) \wedge \mathbf{G}(\mathbf{F}(\neg r_1 \tilde{\mathbf{S}} g_1)) \wedge \mathbf{G}(\mathbf{F}(\neg r_2 \tilde{\mathbf{S}} g_2)) \wedge$$
$$\mathbf{G}(g_0 \rightarrow (r_0 \vee \mathbf{Y}(r_0 \mathbf{B} \neg g_0))) \wedge \mathbf{G}(g_1 \rightarrow (r_1 \vee \mathbf{Y}(r_1 \mathbf{B} \neg g_1))) \wedge \mathbf{G}(g_2 \rightarrow (r_2 \vee \mathbf{Y}(r_2 \mathbf{B} \neg g_2))) \tag{A.23}$$

A.2 Related pseudocode

A.2.1 Build local past function cache

```
fn build_local_past_function_cache(ltl: pLTL, result: &mut Map<pLTL, CacheItem>) {
    if result.contains(ltl) {
        return;
    }
    match ltl {
        T | F => {
            result[pLTL] = CacheItem {
                atom_mask: 0b0,
                past_st_mask: 0b0,
                cache: {(0b0, 0b0) => ltl},
            };
        }
        p => {
            let atom_mask = 1 << atom;
            result[pLTL] = CacheItem {
                atom_mask,
                past_st_mask: 0b0,
                cache: {(0b0, 0b0) => F, (atom_mask, 0b0) => T},
            };
        }
        !p => {
            let atom_mask = 1 << atom;
            result[pLTL] = CacheItem {
                atom_mask,
                past_st_mask: 0b0,
                cache: {(0b0, 0b0) => T, (atom_mask, 0b0) => F},
            };
        }
        Y(content) => {
            build_local_past_function_cache(content, result);
            let subresult = result[content];
            let mut cache = {};
            let result_value = F;
            for ((letter, past_st), content_result) in content_entry.cache {
                cache[(letter, past_st)] = result_value;
```

```
            cache[(letter, past_st | (1 << id))] = result_value;
        }
        let item = CacheItem {
            atom_mask: subresult.atom_mask,
            past_st_mask: subresult.past_st_mask | (1 << id),
            cache,
        };
        result[pLTL] = item;
    }
    // Weak Y is similar to Y, just change result_value from F to T
    X(content) => {
        build_local_past_function_cache(content, result);
        let content_entry = result[content];
        let mut cache = {};
        for ((letter, past_st), content_result) in content_entry.cache {
            cache[(letter, past_st)] = pu_1(
                content_result,
                letter,
                past_st,
                result
            );
        }
        result[pLTL] = CacheItem { atom_mask, past_st_mask, cache};
    }
    And(contents) => {
        let mut result_atom_mask = 0b0;
        let mut result_past_st_mask = 0b0;
        for content in contents {
            build_local_past_function_cache(content, result);
            result_atom_mask |= result[content].atom_mask;
            result_past_st_mask |= result[content].past_st_mask;
        }
        let mut cache = {};
        for result_atom in result_atom_mask.sub_sets() {
            for result_past_st in result_past_st_mask.sub_sets() {
                let mut result_pltls = [];
                for sub_result in sub_results.iter() {
                    result_pltls.push(
                        result[sub_result]
                            .get(result_atom, result_past_st)
                    );
                }
                cache[(result_atom, result_past_st)] = And(result_pltls);
            }
        }
        result[pLTL] = CacheItem {
```

```
                atom_mask: result_atom_mask,
                past_st_mask: result_past_st_mask,
                cache,
            };
        }
        // Or is similar to And
        lhs U rhs => {
            build_local_past_function_cache(lhs, result);
            build_local_past_function_cache(rhs, result);
            let lhs_entry = result[lhs];
            let rhs_entry = result[rhs];
            let atom_mask = lhs_entry.atom_mask | rhs_entry.atom_mask;
            let past_st_mask = lhs_entry.past_st_mask | rhs_entry.past_st_mask;
            let mut cache = {};
            for result_atom in atom_mask.sub_sets() {
                for result_past_st in past_st_mask.sub_sets() {
                    let lhs_result = lhs_entry.get(result_atom, result_past_st);
                    let rhs_result = rhs_entry.get(result_atom, result_past_st);
                    let pu = pu_l(ltl, result_atom, result_past_st, result);
                    cache[(result_atom, result_past_st)] =
                        rhs_result & (lhs_result | pu);
                }
            }
            result[pLTL] = CacheItem { atom_mask, past_st_mask, cache};
        }
        // R is dual to U
        lhs S rhs => {
            build_local_past_function_cache(lhs, result);
            build_local_past_function_cache(rhs, result);
            let wc = ltl.weaken_condition();
            build_local_past_function_cache(wc, result);
            let wc_entry = result[wc];
            let mut cache = wc_entry.cache;
            for ((letter, past_st), content_result) in wc_entry.cache {
                cache[(letter, past_st | (1 << id))] = content_result;
            }
            result[pLTL] = CacheItem {
                atom_mask: wc_entry.atom_mask,
                past_st_mask: wc_entry.past_st_mask | (1 << id),
                cache,
            };
        }
        // B is similar to S
    }
}
```

B

Disclaimer

Generative AI was utilized in the writing of this thesis solely for grammar and spelling corrections, with all suggestions undergoing subsequent manual inspection. The author promises that all ideas, data, and the initial content of this thesis are original and produced without generative AI assistance.