Learning and Verifying Temporal Specifications for Cyber-Physical Systems

Ritam Raha

▶ To cite this version:

Ritam Raha. Learning and Verifying Temporal Specifications for Cyber-Physical Systems. Artificial Intelligence [cs.AI]. Université de Bordeaux; University of Antwerp, 2023. English. NNT : 2023BORD0208 . tel-04267885

HAL Id: tel-04267885
https://theses.hal.science/tel-04267885

Submitted on 2 Nov 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Learning and Verifying Temporal Specifications for Cyber-Physical Systems

Ritam Raha

Supervisors **Dr. Nathanaël Fijalkow | Dr. Guillermo Pérez | Dr. Jérome Lerôux**
Jury President **Prof. Toon Calders**

Thesis submitted in fulfilment of the requirements for the degree of Doctor in Computer Science **at the University of Antwerp, Belgium & the University of Bordeaux, France | Department of Computer Science | Antwerp, 2023**

University of Antwerp

université de BORDEAUX

COTUTELLE THESIS PRESENTED

for the degree of

Doctor in Computer Science

at

The University of Antwerp, Belgium

&

The University of Bordeaux, France

Faculty of Science
and École Doctorale de Mathématiques et Informatique

by **Ritam Raha**

Learning and Verifying Temporal Specifications for Cyber-Physical Systems

2023/09/12

Supervisors
Dr. Nathanaël Fijalkow
Dr. Guillermo Pérez
Dr. Jérome Lerôux

Jury

Chair
Prof. Toon Calders, University of Antwerp, Belgium

Supervisors
Dr. Nathanaël Fijalkow, LaBRI, University of Bordeaux, France
Dr. Guillermo Pérez, University of Antwerp, Flanders Make, Belgium
Dr. Jérome Lerôux, LaBRI, University of Bordeaux, France

Members
Prof. Véronique Bruyère, Université de Mons, Belgium
Prof. Jean-François Raskin, Université libre de Bruxelles, Belgium
Prof. Anca Muscholl, LaBRI, University of Bordeaux, France
Prof. Emmanuel Filiot, Université libre de Bruxelles, Belgium
Prof. Suguman Bansal, Georgia Institute of Technology, USA

Contact
Ritam Raha

University of Antwerp, Flanders Make, Belgium &
LaBRI, University of Bordeaux, France
ritam.raha@uantwerpen.be

© 2023 Ritam Raha
All rights reserved.

To my Sister (Didi),

for being the unwavering support and strength in my life

The wings that yearned forever to fly,
Bound, denied the freedom of the sky,
Will get riddance of their plight,
Will soar high in unshackled flight.

- Parash Pathar

একঝাঁক ইচ্ছে ডানা,
যাদের আজ উড়তে মানা
মিলবেই তাদের অবাধ স্বাধীনতা

- পরশ পাথর

Acknowledgements

[The scene is set during the COVID-19 pandemic, two years into the PhD; challenging times, your first paper just got rejected for the third time, and you seem frustrated.]

Scene: Virtual Meeting Room

Person 1: This is absolutely okay, things happen. Just continue what you are doing, and it will be alright.

Person 2: I know these past two years have been tough for you, and back-to-back rejections have been disheartening. But remember, it is all about making progress and putting things in the pipeline.

[You nod, appreciating the support and encouragement. They probably believe in you more than you do at this point.]

[Cut to your first paper acceptance in PhD, 29th September 2021]

Person 1 (via email): Congratulations Ritam! That is a great accomplishment.

Person 2: Congratulations, you got it. I told you this would happen soon. This has to call for a celebration. I am sending you celebratory beers at home. We have to savour the moment.

[You're touched by the gestures, feeling a renewed sense of determination and gratitude.]

The above incident was me recounting one of the very fond memories from the early years of my PhD journey. The two persons above are my two wonderful supervisors, Nathanaël Fijalkow and Guillermo Perez, who have been nothing but absolute support for me during these four years of my PhD. That is why I want to take a step back and start by thanking Prof. B. Srivathsan from Chennai Mathematical Institute and Prof. Filip Mazowiecki from the University of Warsaw for helping me to choose them as my supervisors.

During my PhD journey, I had the privilege of being mentored by Nath and Guillermo. Starting from literature surveys, they taught me how to approach and tackle problems, formalize solutions into research papers, and then present them engagingly to the world. I will do my utmost to carry these valuable lessons with me throughout my research endeavours and try to make them proud. It is not just the research lessons I am thankful for; I am immensely grateful for their unwavering emotional support, invaluable life lessons, and remarkable tolerance even when I was not at my best. Tolerating my constant nagging, handling the mysterious world of administrative

procedures, and putting up with my frequent oversight of adding spaces after commas in research papers, they were like the epitome of patience throughout the years.

PhD life is no walk in the park, and the added curveball was the pandemic hitting just four to five months after I started on this journey. Throughout these tough times, their genuine care, empathy, and mental support went above and beyond mere academic guidance. I will always be grateful for their constant support, which extended far beyond the confines of academia.

I would like to thank Prof. Jean-François Raskin and Prof. Véronique Bruyère for carefully reviewing this dissertation. I am also thankful to all my jury members for their helpful feedback and suggestions to improve and polish the writing.

During the past four years, I have met and talked to several other PhD students and PostDocs who shared their thoughts on life in and beyond academia, which influenced me in some way. I owe many thanks to the AnSyMo group: Tim, Ramesh, Shrisha, Kasper, and Alejandro. Cheers to all the great times: discussions, coffee, pizza, chess, papers, travel, beers, jokes and whatnot! It was great interacting with all of you, and I will cherish them a lot.

I would also like to thank all my collaborators for their great discussions and insights. Prof. Michaël Cadilhac, our beer-fueled discussion was incredibly enlightening, offering me fresh perspectives and a much-needed dose of encouragement. I am genuinely grateful for your insights.

A PhD life goes way beyond just the academic interactions and exceptional supervisors; it is a journey where friends become the indispensable knot that ties together the pieces of this intricate puzzle of academic and personal growth. Rajarshi, an unexpected academic collaborator and a wonderful friend beyond, thank you for being this incredible friend (and colleague!!!) during both the ups and downs of my journey. Kush, let's share and laugh at those (silly?) jokes forever! Dhruvi, thank you for being a fantastic roommate and friend; you were there to provide great support, to share a laugh, to lend a listening ear, or to simply enjoy a comforting meal together. Niladri and Anirban Da, thank you for making Belgium feel like home in every way possible. Tamalika, thank you for all the conversations, sharing highs and lows, and being a wonderful friend to share daily musings: the first page of this dissertation is owed to you. My CMI friends and seniors – Vasudha, Arijit, Ananyo, Sougata Da, Suman Da, Prantar Da, Soumyajit Da, Chayan Da, Rajat Da, Debraj Da, and Ranadeep Da – you were my pillars of support, helping me preserve my sanity every single day. Whether through countless video calls, Discord gaming sessions, or shared travelling adventures, your daily online or offline interactions meant the world to me. I had wonderful times and memories with many old and new friends here; some left, some stayed, but each and every one of you has left a positive mark on my life. I want to extend my heartfelt gratitude to all of you for being part of this remarkable journey.

Finally, as I stand here today with a broad smile on my face, I owe it all to the unwavering encouragement, understanding, and sacrifices of my family. Shiuli, you've been my rock, my confidant, and the person I always look forward to. Miles apart or not, you were always there and always understood, and I love you.

Baba and Ma (my parents), I will never fully comprehend the sacrifices you made to

make all of this possible. Thank you for believing in me through thick and thin.

During my PhD, I lost one of my grandmothers (Thamma), but I'm certain she's always cheering for me, just as she always did. I miss you!

Didan, I'll be there to see you soon, and I couldn't be more excited!

Mamoni (my Aunt), thank you for your unwavering support for me and my parents.

Didi (my sister), from being just centimetres away to thousands of miles apart, you've been my lifelong support, allowing me to sleep peacefully each night. This achievement is dedicated to you with all my heart!

Abstract

In the past decade, there has been an unprecedented rise in the incorporation of cyber-physical systems being used in performing complex tasks. Verification of these systems is necessary to ensure the safety and reliability of them, especially in safety-critical scenarios. Formal verification has been proven to be a time-tested method to systematically verify these systems. In particular, it provides techniques for monitoring system executions against a formal specification. Temporal logic has gained popularity as a formal specification due to its mathematical rigour, human interpretability, and compatibility with various formal verification techniques. But, often, these specifications are hard to design manually, necessitating a need to automatically generate them from system executions. The objective of this thesis is twofold: (i) Develop algorithms to automatically learn temporal specifications from system executions; (ii) Apply formal verification techniques to assess the correctness of the system against the acquired specifications.

To achieve these objectives, in the learning part, we present two algorithms to learn specifications in temporal logics such as LTL, MTL and STL. Our algorithms are in the *passive learning* setting where one learns specifications that separate a finite set of positive system behaviours from a finite set of negative ones. For LTL specifications, we present a dynamic programming based algorithm that leverages a normal form for a fragment of LTL and uses efficient enumeration techniques to learn concise formulas from system executions. For MTL and STL, we develop SMT-solver based techniques to learn formulas that are not only concise but also efficient to be applied for formal verification of the system. Our algorithms are implemented in tools called `SCARLET` and `TEAL`, and their efficiency is demonstrated through experimental results. In the verification part, we focus on the LTL parameter-synthesis problems of one-counter automata, which is one of the fundamental formal models to represent complex systems, namely systems with discrete yet infinite state space. One-counter automata are obtained by extending classical finite-state automata with a counter whose value can range over non-negative integers and be tested for zero. The updates and tests applicable to the counter can further be made parametric by introducing a set of integer-valued variables called parameters. The LTL parameter synthesis problem for such automata asks whether a valuation of the parameters exists such that all infinite runs of the automaton satisfy an LTL specification. Building upon the existing works in the literature, (i) we show a careful re-encoding of the problem into a decidable restriction of Presburger Arithmetic with Divisibility (PAD), the largest known decidable fragment of PAD till now; (ii) We prove that the parameter synthesis problem is decidable and in 3NEXP; (iii) We give EXPSPACE algorithms for the special cases of the problem where parameters can only be used in counter tests. Finally, (iv) we also establish decidability of this problem on an extension of the model with Parikh constraints.

Résumé

Au cours de la dernière décennie, on a assisté à une augmentation sans précédent de l'incorporation de systèmes cyber-physiques complexes utilisés pour effectuer des tâches complexes. La vérification de ces systèmes est nécessaire pour garantir leur sécurité et leur fiabilité, en particulier dans les scénarios de sécurité critiques. La vérification formelle s'est avérée être une méthode éprouvée pour vérifier systématiquement ces systèmes. Elle fournit notamment des techniques permettant de contrôler l'exécution des systèmes par rapport à une spécification formelle. Souvent, ces spécifications sont difficiles à concevoir manuellement, d'où la nécessité de les générer automatiquement à partir des exécutions du système pour les utiliser à des fins de vérification. La logique temporelle a gagné en popularité en tant que spécification formelle en raison de sa rigueur mathématique, de son interprétabilité par l'homme et de sa compatibilité avec diverses techniques de vérification formelle. L'objectif de cette thèse est double : (i) Développer des algorithmes pour apprendre automatiquement des spécifications temporelles à partir d'exécutions de systèmes ; (ii) Appliquer des techniques de vérification formelle pour évaluer l'exactitude du système par rapport aux spécifications acquises.

Pour atteindre ces objectifs, dans la partie apprentissage, nous présentons deux algorithmes pour apprendre des spécifications dans des logiques temporelles telles que LTL, MTL et STL. Nos algorithmes sont dans le cadre de l'apprentissage passif et apprennent des spécifications qui séparent un ensemble fini de comportements positifs du système d'un ensemble fini de comportements négatifs. Pour les spécifications LTL, nous présentons un algorithme basé sur la programmation dynamique qui s'appuie sur une forme normale pour un fragment de LTL et utilise des techniques d'énumération efficaces pour apprendre des formules concises à partir des exécutions du système. Pour MTL et STL, nous développons des techniques basées sur un résolveur SMT pour apprendre des formules qui sont non seulement concises mais aussi efficaces pour être appliquées à la vérification formelle du système. Nos algorithmes sont mis en œuvre dans des outils appelés `SCARLET` et `TEAL`, et leur efficacité est démontrée par des résultats expérimentaux. Pour la partie vérification, nous nous concentrons sur les problèmes de synthèse de paramètres LTL des One-counter automata, qui est l'un des modèles formels fondamentaux pour représenter les systèmes complexes, à savoir les systèmes avec un espace d'état discret mais infini. One-counter automata sont obtenus en étendant les automates classiques à états finis avec un compteur dont la valeur peut s'étendre sur des entiers non négatifs et être testée pour zéro. Les mises à jour et les tests applicables au compteur peuvent être paramétrés en introduisant un ensemble de variables à valeur entière appelées paramètres. Le problème de la synthèse des paramètres LTL pour de tels automates est de savoir s'il existe une évaluation des paramètres telle que toutes les exécutions infinies de l'automate satisfont une spécification LTL. En s'appuyant sur les travaux existants dans la littérature, (i) nous montrons un réencodage prudent du problème en une restriction décidable de

l'arithmétique de Presburger avec divisibilité (PAD), le plus grand fragment décidable connu de PAD jusqu'à présent ; (ii) Nous prouvons que le problème de la synthèse des paramètres est décidable en général et en 3NEXP ; (iii) Nous donnons des algorithmes EXPSPACE pour les cas particuliers du problème où les paramètres ne peuvent être utilisés que dans des contre-tests. Enfin, (iv) nous présentons brièvement le résultat de décidabilité de ce problème sur une extension de ce modèle avec des contraintes de Parikh.

Samenvatting

In het afgelopen decennium is er een ongekende groei geweest bij het bouwen van zogenaamde cyber-fysische systemen die worden gebruikt bij het uitvoeren van complexe taken. Verificatie van dergelijke systemen is noodzakelijk om de veiligheid en betrouwbaarheid van deze systemen te garanderen, vooral in veiligheidskritische scenario's. Formele verificatie is een beproefde methode gebleken om dergelijke systemen systematisch te verifiëren. Het biedt met name technieken om de uitvoering van een systeem te toetsen aan een formele specificatie. Het handmatig opstellen van dergelijke specificaties is vaak moeilijk. Het automatisch leren (via AI technieken) van op basis van systeemuitvoeringen een uitweg kunnen bieden. Temporele logica heeft aan populariteit gewonnen als formele specificatie vanwege haar wiskundige nauwkeurigheid, menselijke interpreteerbaarheid en compatibiliteit met verschillende formele verificatietechnieken. Het doel van dit proefschrift is tweeledig: (i) algoritmen ontwikkelen om automatisch temporele specificaties te leren van systeemuitvoeringen; (ii) formele verificatietechnieken toepassen om de juistheid van het systeem te beoordelen aan de hand van de verkregen specificaties.

Om deze doelen te bereiken, presenteren we in het leergedeelte twee algoritmen om specificaties te leren in temporele logica's zoals LTL, MTL en STL. Onze algoritmen situeren zich in een zogenaamde passieve leeromgeving die specificaties leren op basis van een eindige verzameling positieve systeemgedragingen gescheiden van een eindige verzameling negatieve. Voor LTL specificaties presenteren we een algoritme gebaseerd op dynamische programmering dat gebruikmaakt van een normale vorm voor een fragment van LTL en efficiënte opsommingstechnieken gebruikt om beknopte formules te leren van systeemuitvoeringen. Voor MTL en STL ontwikkelen we SMT-solvergebaseerde technieken om formules te leren die niet alleen beknopt zijn, maar ook efficiënt om toe te passen voor formele verificatie van het systeem. Onze algoritmen zijn geïmplementeerd in tools genaamd `SCARLET` en `TEAL`, en hun efficiëntie wordt aangetoond door middel van experimentele resultaten. Voor het verificatiegedeelte richten we ons op de LTL parameter synthese problemen van one-counter automaten, een van de fundamentele formele modellen om complexe systemen te representeren, namelijk systemen met een discrete maar oneindige state space. One-counter automaten worden verkregen door klassieke eindige-statenautomaten uit te breiden met een teller waarvan de waarde kan variëren over niet-negatieve gehele getallen en getest kan worden op nul. De updates en tests die van toepassing zijn op de teller kunnen verder parametrisch gemaakt worden door een verzameling integer-gewaardeerde variabelen te introduceren die parameters genoemd worden. Het LTL parametersyntheseprobleem voor zulke automaten vraagt of er een waardering van de parameters bestaat zodat alle oneindige runs van de automaat voldoen aan een LTL specificatie. Voortbouwend op bestaand werk in de literatuur, (i) tonen we een zorgvuldige hercodering van het probleem in een beslisbare beperking van Presburger Arithmetic with Divisibility (PAD), het grootste tot nu toe bekende

beslisbare fragment van PAD; (ii) bewijzen we dat het parametersyntheseprobleem in het algemeen en in 3NEXP ontcijferbaar is; (iii) geven we EXPSPACE-algoritmen voor de speciale gevallen van het probleem waarbij parameters alleen kunnen worden gebruikt in tellertests. Tot slot (iv) presenteren we kort het beslisbaarheidsresultaat van dit probleem op een uitbreiding van dit model met Parikh beperkingen.

Contents

1 Introduction **1**

1.1 Cyber-Physical Systems . . . 1

1.2 Verification . . . 2

1.2.1 Formal Methods . . . 2

1.2.1.1 Formal Specifications and its Automated Learning . . . 3

1.2.2 Formal Verification . . . 4

1.2.2.1 Counter Machines and One-Counter Automata . . . 5

1.2.2.2 Model Checking and Parameter Synthesis . . . 6

1.3 Contributions . . . 7

1.3.1 Learning Temporal Specifications . . . 7

1.3.2 Verifying Linear Temporal Properties of One-Counter Automata . 9

1.4 Structure and Style of this Thesis . . . 9

1.5 Joint Work and Copyright Notice . . . 10

2 Preliminaries **13**

2.1 Notations and Conventions . . . 13

2.2 Formal Logic . . . 13

2.2.1 Propositional Logic . . . 13

2.2.2 First-Order Logic . . . 14

2.2.2.1 Presburger Arithmetic . . . 15

2.2.3 Temporal Logic . . . 15

2.2.3.1 Linear Temporal Logic . . . 16

2.2.3.2 Metric Temporal Logic . . . 17

2.2.3.3 Signal Temporal Logic 19
2.3 Graph, Automata, and Languages 20
2.3.1 Graphs and Flows 20
2.3.2 Automata and Languages 21
2.3.3 Büchi Automata and Connection to LTL 21

I Learning Specifications 23

3 Learning Formulas in LTL 25
3.1 Background 25
3.1.1 Related Works 26
3.1.2 Our Approach 27
3.2 Outline of the Chapter 28
3.3 LTL on Finite Traces 28
3.4 LTLf Learning Problem: Exact and Noisy 29
3.5 The Exact Learning Algorithm for LTLf 29
3.5.1 Searching for Directed LTL 30
3.5.1.1 Generating Directed LTL Formulas 31
3.5.1.2 The Dual Point of View: Capturing the G operator 34
3.5.2 Boolean Combinations of Formulas 34
3.5.2.1 Boolean Set Cover Problem 35
3.5.2.2 Decision Tree 37
3.5.3 Anytime Property 38
3.6 Extension to the Noisy Setting 38
3.7 Theoretical Guarantees 38
3.7.1 Correctness 39
3.7.2 Termination 39
3.7.3 Completeness 39
3.8 Experimental Evaluation 41

3.8.1 Experimental Setup 41
3.8.1.1 Sample Generation 42
3.8.2 RQ1: Performance Comparison 43
3.8.3 RQ2: Scalability 48
3.8.4 RQ3: Anytime Property 49
3.8.5 RQ4: Noisy Setting 49
3.9 Conclusion 50

4 Learning formulas in MTL **53**
4.1 Background 53
4.1.1 Related Works 55
4.2 Outline of the Chapter 56
4.3 The Problem Formulation 56
4.3.1 Signals and Prefixes. 56
4.3.2 Metric Temporal Logic 57
4.3.3 The Problem Formulation 58
4.4 An SMT-based Algorithm 60
4.4.1 Linear Real Arithmetic (LRA) 60
4.4.2 Algorithm Overview 60
4.5 Theoretical Guarantees 72
4.5.1 Correctness 72
4.5.2 Termination 72
4.5.3 Complexity Results 72
4.6 Experimental Evaluation 74
4.6.1 Experimental Setup 75
4.6.2 RQ1: Performance Evaluation 75
4.6.3 RQ2: Future-reach vs Size 76
4.6.4 RQ3: Scalability 77
4.7 Discussion 77

4.8 Extension to STL Learning 78
4.9 Conclusion 80

II Verification: Parameter Synthesis for One-Counter Automata 81

5 Parameter Synthesis for One-Counter Automata with Parametric Updates and Tests 83
5.1 Background 83
5.1.1 Related Works 85
5.1.2 Problems in the Literature 85
5.1.3 Our Contribution 86
5.2 Outline of the Chapter 87
5.3 Presburger Arithmetic with Divisibility 87
5.3.1 Allowing One Restricted Alternation 87
5.3.2 Undecidability of Both One-alternation Fragments 88
5.3.3 The Bozga-Iosif-Lechner (BIL) Fragment 89
5.3.3.1 The Formulation 89
5.3.3.2 The Flaw in the Previous Attempt 90
5.3.3.3 Decidability of BIL 91
5.4 Succinct One-Counter Automata with Parameters 93
5.4.1 Universal Parameter Synthesis problems for One-Counter Automata 95
5.5 Complexity of the Parameter Synthesis Problems 96
5.5.1 Reduction to coBüchi Synthesis Problem 96
5.5.2 Encoding coBüchi Parameter Synthesis to BIL 98
5.5.2.1 Reachability Certificates 101
5.5.2.2 Encoding the Reachability Certificates 101
5.5.2.3 Putting Everything Together 104
5.6 Parikh One-counter Automata 106
5.7 Conclusion 108

6 Parameter Synthesis for One-Counter Automata with Parametric Tests Only 109

6.1 Outline of the Chapter . . . 109
6.2 One-Counter Automata with Parametric Tests . . . 109
6.3 Alternating Two-way Automata (A2A) . . . 110
6.4 Transformation to Alternating Two-way Automata . . . 111
6.4.1 Parameter Word: Encoding Valuation to Words . . . 111
6.4.2 Overview of the Construction . . . 112
6.4.3 Construction of Sub-A2As . . . 113
6.4.3.1 Verifying the Input Word . . . 113
6.4.3.2 Encoding the Increment Operations . . . 114
6.4.3.3 Encoding the Decrement Operations . . . 115
6.4.3.4 Encoding the Zero Tests . . . 115
6.4.3.5 Encoding the Parametric Equality Tests . . . 116
6.4.3.6 Encoding the Parametric Lower-bound Tests . . . 116
6.4.4 Correctness of the Encoding . . . 117
6.5 Improving Upper Bound for Reachability Parameter Synthesis . . . 118
6.5.1 Universal Reachability for Non-parametric One-Counter Automata 119
6.5.2 Back to Parameter Synthesis . . . 120
6.6 Conclusion . . . 121

7 Conclusion and Future Work 123

Chapter 1
Introduction

Formal verification has proven to be a time-tested method to verify complex systems systematically. In the context of hardware and software systems, formal verification ensures the correctness of system executions against a property or a formal specification. Manually designing the specifications is often challenging and resource-expensive. The aim of this dissertation is two-fold: (i) Develop algorithms to automatically synthesise (learn) specifications from system behaviours, and (ii) Use formal verification techniques to check the correctness of the system against these learnt specifications. In this chapter, we first familiarize the reader with the concept of "specifications" and "formal verification" and, in particular, what kind of systems we want to verify. We will then focus on automatically learning these specifications from system behaviours. Then, we elaborate on the specific formal models and verification techniques we use to verify them. Finally, we will list the contributions towards these aspects developed in this dissertation.

1.1 Cyber-Physical Systems

Cyber-Physical Systems (CPS) refer to systems that consist of physical entities wherein computer-based algorithms are employed to exercise control or monitor the operation of these physical mechanisms. These systems are integrated with computing and communication technologies, creating a complex, interconnected network of physical and digital components. They are ubiquitous in the modern world as they are being used everywhere, including the industries of manufacturing, transportation, energy, healthcare, and infrastructure. Examples of CPS include autonomous vehicles, medical implantation, robotics systems, automatic aviation, and smart grids. They are extensively being used in safety-critical scenarios, where the failure of the systems may lead to hazardous economic and life-threatening situations. For example, a bug in the code that managed the Therac-25 radiation therapy machine caused at least five patient fatalities during the 1980s by delivering too much beta radiation [97]. Multiple reports of accidents and deaths have been reported for software bugs in Tesla's autonomous vehicle models [54]. Hence, systematic analysis and verification of the safety and correctness of CPS have emerged as a necessary and growing research area.

1.2 Verification

Verification aims to establish whether a system meets a set of requirements. It is crucial to develop cyber-physical systems that are safe and have a high level of assurance. Despite its necessity, this process is difficult and often error-prone. Towards this, *formal methods*, including techniques like model checking, abstract interpretation, parameter synthesis and theorem proving, have shown promise in addressing the challenges associated with the verification of CPS. Other methods used in the verification of CPS include simulation and testing. However, formal methods are increasingly being adopted in industry and academia due to their ability to offer mathematical rigour and systematically ascertain the correctness of system properties. More detailed explanations follow in the next subsections.

1.2.1 Formal Methods

Formal Methods are mathematically precise approaches utilized for outlining, developing, and validating cyber-physical systems based on mathematical reasoning. Formal methods employ mathematical models and techniques like automata, counter-machines, set theory and logical formalisms [86, 110, 108]. Formal methods have gained popularity in recent years due to the increasing complexity of software and hardware systems [28]. The use of formal methods helps to detect and correct errors early in the design process, reducing the cost and time required for testing and debugging. Formal methods have been used successfully in a wide range of CPS, including avionics, automotive, medical devices, and telecommunications [109, 5, 26].

One way of categorizing formal methods is as follows:

- *Formal Specifications.* Formal specifications refer to system properties that are formalized mathematically. They can be employed at an early stage to develop the system or at a later stage to check if the developed system satisfies the desired properties
- *Formal Verification.* Formal verification aims to establish mathematically whether the underlying system meets a formal specification or produces a bad trace of the system that does not satisfy. This technique can be applied during the development process to verify some components of the system or at the end of the development process.
- *Theorem Provers.* Theorem proving is concerned with mechanically proving the correctness of mathematical statements or theorems to verify system properties. Although theorem provers can help improve tools, their associated cost can be substantial due to the complexity of the work they aim to accomplish. Thus, theorem provers are only justifiable if the cost of errors or inaccuracies is significantly high.

This work focuses on formal specifications and formal verification of CPS. We elaborate on this below.

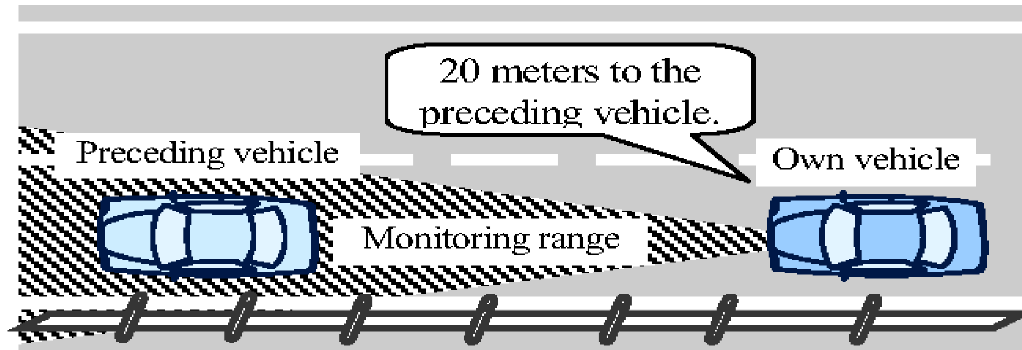


Figure 1.1: Safety-critical specifications for autonomous vehicles[1]

1.2.1.1 Formal Specifications and its Automated Learning

Formal specifications are models to formally describe a system, analyze its behaviour, and assist in its design by verifying essential properties of interest using effective verification tools. Such specifications are 'formal' in the sense that they possess a well-defined syntax, their semantics fall within a specific domain, and they can be used to infer useful information. By utilizing formal specifications, it is possible to precisely and unambiguously define the system's requirements, behaviour, and interface. Formal specifications can be implemented using different formalisms, such as mathematical logic, automata theory, or process algebras, depending on the system's characteristics and the properties of interest. In this thesis, we focus on *temporal specifications*, which consists of a set of temporal logic formalisms such as Linear Temporal Logic (LTL), Metric Temporal Logic (MTL) etc., that describe the evolution of a system's behaviour over time in terms of logical propositions and temporal operators. The development of temporal logic as a tool for formal methods can be traced back to the mid-20th century, with the pioneering work of Arthur Prior [74] and later followed by the introduction of LTL by Pnueli [118] in the 1980s. In the context of cyber-physical systems, temporal specifications are particularly important, as these systems often involve complex interactions between physical and computational components.

Example 1 *Let us consider an example from modelling autonomous vehicles. One of the safety-critical requirements for such a vehicle is as follows:*

At any time point, if the sensor detects a preceding vehicle at an unsafe distance (less than twenty meters), the brake of the vehicle needs to be triggered. The requirement is shown in Figure 1.1, where the unsafe distance is 20 meters. This can be expressed as an LTL specification that says $Always(unsafe\ distance \rightarrow trigger\ brake)$, *where* $unsafe\ distance$ *denotes the event that there is a preceding vehicle at a distance of less than twenty meters and* $trigger\ brake$ *denotes the action of triggering the brake.*

Learning Specifications. The process of manually writing specifications is widely recognized as tedious and error-prone [3]. A significant challenge in formal verification

[1]Image taken from [131]

lies in synthesising specifications that are functional, correct, and easily interpretable, accurately capturing the design requirements [21, 130].

To address the lack of formal specifications, researchers have made efforts to automatically synthesise specifications from system executions. Many of these approaches aim to produce concise specifications, as they are deemed preferable over larger ones due to their ease of human understanding, following the principle of Occam's razor [129]. These methods have gained considerable popularity in two growing research fields:

- *Specification Mining:* Specification mining [3] is closely associated with formal verification and focuses on automatically extracting formal specifications from system data to assist programmers and verification tools. It plays a crucial role in formal verification by enabling analysis of system behaviour, properties, and vulnerabilities based on observed data. By analyzing logs, traces, or other execution data, specification mining techniques uncover implicit or latent rules, patterns, and dependencies that govern the system. These extracted specifications enhance system understanding, testing, debugging, verification, and validation processes and enhance the overall reliability and security of cyber-physical systems.
- *Explainable AI.* The integration of automation in software systems has been driven by the adoption of data-driven techniques, commonly known as Artificial Intelligence (AI), in system design. However, the complexity of intelligent systems often leads to a limited understanding of their inner mechanisms. To establish human 'trust' in these systems, there is a need for techniques that explain their behaviour in a human-interpretable manner and facilitate system verification. Explainable AI (XAI) [112] addresses this challenge by focusing on explaining the behaviour of these black-box systems, particularly in relation to the interpretable explanations of temporal behaviour in cyber-physical systems.

One of the main focus points of this thesis is to automatically learn temporal specifications from system executions. Previously there have been works on learning automaton models as concise specifications from systems [4, 65, 66]. In this thesis, we focus on learning specifications in temporal logics as (i) they are very close to natural language [113, 25, 34], (ii) they can formalize temporal specifications of a system, and (iii) they are widely used in formal verification [8, 118]. Note that our learning techniques are in the *passive learning* setting [20], a common classification (active vs passive) in the literature of automata learning, where a learning algorithm is confronted with a finite set of traces or executions that have some kind of classification (positive and negative, for binary classification) attached. The goal is to find the most concise formal specification (automata, temporal logic formula etc.) that is consistent with the set of classified traces.

1.2.2 Formal Verification

Formal verification involves the construction of a mathematically rigorous "formal model" that abstracts the mechanics of the system. Mathematical proofs are then

employed to establish the validity of a property, represented as a formal specification, within the formal model. Formal verification has evolved as a prominent field since its inception in the mid-20th century, drawing inspiration from automata theory, logic, and pioneering works by researchers such as Kleene, Büchi, and Rabin [86, 121]. Various formal models have been developed and utilized in the field of formal verification, such as finite state machines, logic, and automata.

Finite state machines (FSMs) and automata serve as fundamental models in formal verification, enabling the analysis of system behaviour based on state transitions and inputs. A finite state machine consists of a finite set of states, transitions between states based on inputs, and outputs associated with each transition. By capturing the system's behaviour in terms of states and transitions, FSMs allow for systematic analysis and verification of system properties. However, FSMs have inherent limitations in capturing complex and dynamic (potentially infinite) system evolution over time, highlighting the need for more expressive models that are able to argue about various quantitative properties in verification processes.

1.2.2.1 Counter Machines and One-Counter Automata

Counter machines are computational models equipped with one or more counters that can be incremented, decremented, and tested. They extend the expressive power of finite state machines by incorporating counters to track and enforce quantitative properties, making them essential in formal verification for reasoning about quantitative aspects of system behaviour. We demonstrate this in the example of the autonomous vehicle we mentioned to introduce formal specifications.

Example 2 *Let us consider the scenario of modelling and verifying an autonomous vehicle in the context of potential unsafe behaviours, as illustrated in Example 1. We need to track the distance to the preceding vehicle to check if the vehicle is in a 'normal' or an 'unsafe' state. Now the number of such configurations (state, distance) can be potentially infinite as the distance can be any number, making it impossible to be modelled as a finite state machine. On the other hand, reasoning about all possible configurations of a system (normal/unsafe with the actual distance to the preceding vehicle, in this particular example) is very costly and not even clearly possible. Here the counter machine comes to the rescue; we can model this with a counter machine, where a counter keeps track of the distance to the preceding vehicle. We illustrate this in detail below.*

Let there be a sensor in the vehicle that controls the driving mode, which we call a driver. We abstract its behaviour as discrete actions, accelerate *and* decelerate*. For simplicity, we assume that when it accelerates, the distance to the preceding vehicle decreases by one meter, and when it decelerates, the distance increases by one meter (this actually depends on the vehicle). The acceleration and deceleration of the driver are controlled by two mutually exclusive signals it gets:* Brake *and* noBrake*. We can model this safety feature as a counter machine, as illustrated in Figure 1.2, that can trigger these signals to the driver. We denote the counter measuring the distance to the preceding vehicle as d. Let safeDistance be the threshold of the safe distance to the preceding vehicle, which we set here as 20 meters.*

The counter machine in the figure represents a formal model for verifying the safety of the autonomous vehicle. The alphabet or the labels of the transitions are a combination of actions of the driver and the signals that it receives: accelerate, decelerate, Brake and noBrake. It consists of two states, "Normal" and "Unsafe," representing the vehicle's normal and potentially dangerous

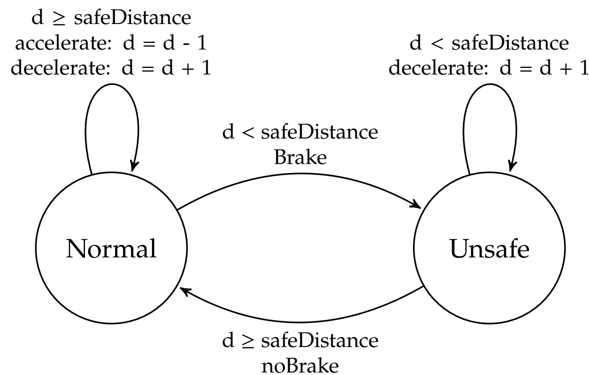


Figure 1.2: Counter machine model for verifying the behaviour of an autonomous vehicle in relation to unsafe distances and triggering brake actions.

conditions, respectively. Transitions between states depend on the value of the counter variable d, which represents the distance to the preceding vehicle. In the "Normal" state, the counter is updated based on the vehicle's normal behaviour as controlled by the driver, and if it goes below the threshold safeDistance (20 meters in the example), the counter machine forces the output action Brake and goes to "Unsafe" state. The driver receives the signal from the counter machine and then continues the decelerate action until it stays in the "Unsafe" state. In the "Unsafe" state, the value of the counter, i.e., the distance, increases gradually due to the decelerate action and eventually when it surpasses the safe threshold, the driver recovers to the "Normal" state, as it receives the 'noBrake' signal from the counter machine. This counter machine facilitates the formal verification of the properties concerning maintaining a safe distance and triggering appropriate actions in the autonomous vehicle.

Counter machines were introduced by Minsky [110] as a formal model, and they are also commonly known as counter automata, counter nets or Minsky machines. He showed that counter automata with two counters possess the same computational power as Turing machines. Consequently, nearly all verification problems related to counter machines are undecidable. To maintain decidability, various restrictions have been proposed in the literature, e.g. reversal-bounded counter automata [81] and automata with a single counter. In this thesis, we focus on the latter, called *one-counter automata* as a formal model where we allow one counter and the counter can be updated and tested along the transitions. This model has found applications in various fields of formal verification [27, 36], and we motivate and formally introduce the model in the corresponding chapter in this thesis later.

1.2.2.2 Model Checking and Parameter Synthesis

Given a formal model of the systems and a formal specification, a formal verification technique typically comprises mathematical techniques to check if the model satisfies the specification. One of the most widespread and popular techniques used is *model checking*, which systematically explores all possible states of a state machine to verify desired properties. For the counter-machine in Figure 1.2, we can verify simple properties like 'safety' that asks if we start from a particular distance, say 30 meters

from the preceding vehicle, i.e., with counter value 30 at the 'Normal' state, do all runs avoid the 'Unsafe' state? In this thesis, we focus on different properties like 'safety, 'reachability', 'Büchi' (repeated reachability), and general ones expressed in LTL.

When designing complex cyber-physical systems, it is often necessary to consider various factors and requirements. These systems often communicate with and evolve based on their surrounding environment. To formalize the varying conditions provided by the environment, the researchers introduce *parameters* in the formal computation models. Parameters are (usually integer-valued) variables [2] that allow for customization and flexibility in system design, enabling different configurations to be explored. However, manually exploring all possible parameter combinations is time-consuming and impractical.

To tackle this, *parameter synthesis problems* are the automated exploration of parameterized design spaces to generate system designs that satisfy specified properties. It involves identifying optimal values or ranges for the parameters that satisfy given specifications. The goal is to efficiently search the parameter space and find designs that meet the desired properties. Given a state machine with parameters and a formal specification, we can ask two kinds of parameter synthesis problems:

- *Existential Parameter Synthesis.* Does there exist a valuation of the parameters such that there exists an execution of the machine that satisfies the specification?
- *Universal Parameter Synthesis.* Does there exist a valuation of the parameters such that all executions of the machine satisfy the specification?

In this thesis, we particularly focus on the latter. In the same example illustrated in Figure 1.2, we can make the 'safeDistance' a parameter (instead of fixing it to a particular value) and ask questions like does there exist a valuation of 'safeDistance' such that all runs from 'Normal' state avoid the 'Unsafe' state? Note that this is an example of a universal parameter synthesis problem for the safety specification.

1.3 Contributions

The present dissertation summarizes our contributions to the theory of learning and verifying formal specifications for cyber-physical systems. The main focus of this thesis is twofold: (i) develop algorithms to automatically learn temporal specifications from complex system behaviours, and (ii) use verification techniques to check the correctness of the system against the learnt specifications. We elaborate on this below.

1.3.1 Learning Temporal Specifications

Towards learning system properties from its executions, the works in this thesis focus on logical specifications. In particular, we aim to learn formulas in temporal logic (e.g. Linear Temporal Logic, Metric Temporal Logic and Signal Temporal Logic). These logics can specify system evolution over time and have been a de facto standard in the

field of formal verification. Over the past few years, learning temporal logic has been identified as an important goal in the field of robotics, specification mining, and artificial intelligence. Explanation in temporal logic is especially desirable as they are not only mathematically sound and precise but also resemble natural language, as argued in the literature. The fundamental problem is to build a specification in the form of a temporal logic formula from a set of execution traces of a system that are classified as positive and negative such that,

- The formula is consistent with the sample, i.e. it agrees with the positive executions and does not with the negative ones;
- The formula is the smallest consistent one, and
- The formula can be efficiently used to apply in formal verification of the system.

Our work first aims to develop algorithms for automatically learning specifications in Linear Temporal Logic (LTL) from labelled execution traces of the system, which is given as a sample. More formally, given a set $u_1, \ldots, u_n$ of positive traces and a set $v_1, \ldots, v_n$ of negative traces, the goal is to construct an LTL formula φ that is consistent with the sample, i.e., all u_i's satisfy the formula φ, and none of the v_i's does. Several approaches towards this problem already exist in the literature leveraging SAT solvers, automata, and Bayesian inference. These existing approaches often suffer from the limitations of scaling. Indeed, theoretical studies have shown that constructing the minimal LTL formula is NP-hard already for very small fragments of LTL, explaining the difficulties found in practice. To address these, we turn to approximation and anytime algorithms. While approximation means that the algorithm may not return the smallest size formula, it does ensure that the output formula is consistent with the sample; the anytime property denotes that the algorithm finds better solutions recursively by refining them the longer it keeps running. On a high-level overview, our algorithm searches for LTL formulas in a particular shape; we call it directed formulas and then try to find the separating formulas using boolean combinations of the same. To analyze and assess our algorithms against the previous algorithms, we have implemented a prototype of our algorithm in Python 3 in a tool named `SCARLET` and ran it on several benchmarks. We present the comparative results in a graphical interpretable manner in our paper. We also establish how to adapt our algorithm for noisy data, a common aspect for applications in AI domains.

Next, we shift our focus to automatically learn specifications in Metric Temporal Logic (MTL), a logical formalism that extends LTL to reason about the continuous evolution of a CPS over time. MTL is popularly employed in the verification of these complex systems. Despite being extensively used in formal and runtime verification, to the best of our knowledge, there are a limited number of works that focus on learning MTL specifications automatically from system executions. The few existing works also are not tailored towards learning MTL formulas to aid verification. Towards this, we present a novel algorithm to automatically learn MTL specifications using two regularizers– size and lookahead. We also implemented our algorithm in a Python 3 based tool named `TEAL` and ran it over several benchmarks to establish the efficiency of our algorithm. In the end, we also show how to lift our techniques to learn specification in Signal Temporal Logic (STL) that extends MTL to handle real-valued predicate and is particularly useful in the verification of hybrid and continuous evolution of CPS.

1.3.2 Verifying Linear Temporal Properties of Parametric One-Counter Automata

In the verification part, this thesis primarily focuses on the computational complexity of verifying one-counter automata, which are counter automata with the restriction of having only one counter. These are particularly useful for modelling programs with a single variable, protocols with unbounded integer storage space, or systems where transitions consume resources like time or money. Note that the counter machine illustrated in Figure 1.2 is a one-counter automaton. The counter can be updated or tested along the transitions. Moreover, we consider a generalization called parametric one-counter automata, where a finite set of parameters can be used at transitions to update or test the counter. We say that an OCA satisfies an LTL formula if there exists a run in the OCA that agrees with the formula. We particularly establish complexity results of *the universal parameter synthesis for OCA against LTL specifications* that asks, given an OCA with parameters $\mathcal{A}$ and a LTL specification φ, does there exist a valuation of the parameters, such that all runs of $\mathcal{A}$ satisfy the formula φ? As a stepping stone to establish this, we also explore simpler properties like reachability, safety, Büchi and coBüchi. For readability, we refer to the universal parameter synthesis problem for OCA against property *property* as the '*property* parameter-synthesis problem'.

First, we show that the LTL parameter-synthesis problem for general OCA with parameters is decidable in 3NEXP. Our algorithm works by reducing the problem to a decidable fragment of *Presburger arithmetic with divisibility (PAD)*, which is heavily inspired by existing works by Christoph Haase and Antonia Lechner [73, 93]. In this work, first, we define *BIL, the largest decidable fragment of PAD* known till now. Then we leverage Haase's existing techniques and formalisms in literature to reduce the LTL parameter-synthesis problem for OCA with parameters to the satisfaction of a BIL sentence. In this chapter, we also briefly touch upon the same problems for an extension of OCA, named *Parikh One-counter Automata*, the study of which was originally motivated by the logical fragment BIL.

Next, we show that the LTL parameter-synthesis problem is in EXPSPACE for a restricted model, where the parameters can only appear in tests and not updates. For this, we take inspiration from the work of [23] and encode parameter valuations of this model into words accepted by an alternating two-way automaton.

1.4 Structure and Style of this Thesis

The organisation of this thesis is as follows. Chapter 2 introduces basic notation and concepts that this thesis builds upon. The aim of this chapter is to introduce all the very general concepts relevant to this thesis and covers definitions from areas such as logic, arithmetic theories, temporal logics, computational complexity, automata theory and one-counter automata. Definitions relevant to a specific chapter are introduced and recalled in the respective chapter. After Chapter 1, the thesis is split into two independent parts.

The first part deals with automatically learning temporal specifications. Chapter 3 presents a novel LTL learning algorithm and empirically establishes its efficiency

against the state-of-the-art tools already available. Chapter 4 presents an algorithm for automatically learning MTL specifications to aid formal verification and show its efficiency using implementation and experimental results. This chapter also presents methods to lift these techniques to automatically learn STL specifications from system executions.

The second part focuses on the formal verification of temporal specifications on a formal model called One-Counter Automata. Specifically, Chapter 5 establishes complexity results of the LTL parameter-synthesis problem for One-counter automata with parameters. It also briefly discusses the same problem for an extension of this model with Parikh constraints. Then, Chapter 6 establishes better complexity results for the same problem in a fragment of the model where parameters can only appear in tests.

Each chapter motivates the specific problem discussed in more detail and also introduces all the required concepts. If a general concept required in the chapter has already been introduced in Chapter 1, we recall it there briefly. We also discuss how these works fit into the literature and all the relevant related works in each chapter. We omit a chapter or section on related works in the literature as it would just consist of a repetition of the related works section in each chapter. In the end, each chapter closes with a discussion about the results obtained and possible future work.

1.5 Joint Work and Copyright Notice

The results presented in this thesis are based on peer-reviewed publications that have been co-authored with several collaborators. These publications reflect the outcome of extensive discussions between the author and the collaborators conducted through in-person meetings or email correspondence. The following paragraph contains the list of publications on which the thesis is based.

In the first part of the thesis, the LTL learning algorithm presented in Chapter 3 has been published in the proceedings of TACAS'22 [122] and was originally published as Scalable Anytime Algorithms for Learning Fragments of Linear Temporal Logic by Ritam Raha, Rajarshi Roy, Nathanaël Fijalkow and Daniel Neider. The corresponding tool called SCARLET has been artifact evaluated at the same conference and has been awarded available, functional and reusable badges. An extension of the tool that supports noisy data is currently under submission in The Journal of Open Software. The MTL learning algorithm presented in Chapter 4 and the corresponding tool TEAL are currently under submission with the name: Synthesising Efficiently Monitorable Formulas in Metric Temporal Logic by Ritam Raha, Rajarshi Roy, Nathanaël Fijalkow, Daniel Neider and Guillermo Perez.

In the second part, the complexity results for the universal parameter synthesis problems for One-counter automata presented in Chapter 5 and Chapter 6 have been published in the proceedings of CSL 2022 [117] and were originally published as Revisiting Parameter Synthesis for One-Counter Automata by Guillermo Perez and Ritam Raha. The results on Parikh One-counter Automata at the end of Chapter 5 are based on the work accepted (to appear) at the conference MFCS 2023 [31] as Parikh One-Counter Automata by Michaël Cadilhac, Arka Ghosh, Guillermo Perez, and Ritam

Raha. In this paper, we also studied this model extensively from language theoretic perspectives, but those results are out of the scope of this thesis and hence are not included in the chapter.

Chapter 2
Preliminaries

In this chapter, we will define the general mathematical notations we will use throughout this thesis.

2.1 Notations and Conventions

We denote by $\mathbb{R}$ the set of real numbers; $\mathbb{Z}$ the set of integers; $\mathbb{N}$ the set of natural numbers—including 0; and $\mathbb{N}_{>0}$ the set of positive integers.

For a set S, we denote the power set (the set of subsets of S) of S by $\mathrm{P}(S)$ or by 2^S interchangeably. We use the symbol $|$ to separate elements of a set and the properties these must satisfy, variable quantification and formulas, etc. Both of them should be read as "such that". For example, $\{x \mid x > 2\} \subset \mathbb{N}$ denotes all natural numbers which are greater than 2.

Let $f : S_1 \rightarrow S_2$ be a function mapping elements from the set S_1 to elements from the set S_2. We call f a *bijection* if for all $s \in S_1$, $f(s) \in S_2$ is unique and for all $s' \in S_2$, $s \in S_1$ such that $f(s) = s'$ is unique. The set S_1 is the *domain* of f, and the set S_2 is the *range* of f.

2.2 Formal Logic

In this section, we will introduce the notion of Formal logic and recall results on decidable and undecidable theories of the same. In the subsequent sections, we will focus especially on its various branches called *propositional logic*, *first-order logic* and *temporal logic*.

2.2.1 Propositional Logic

Propositional logic (also known as *Boolean logic*) is the simplest form of formal logic that includes a set of propositions $\mathcal{P} = \{p_0, p_1, \ldots\}$ and Boolean connectives like $\neg$ (not) and

$\vee$ (or). The other Boolean connectives $\wedge$ (and), $\rightarrow$ (imply) are derived in a standard way:

$$p_i \wedge p_j := \neg(p_i \vee p_j),\ p_i \rightarrow p_j := \neg p_i \vee p_j$$

The set Φ of formulas in propositional logic is the smallest set inductively defined as follows:

- $p \in \Phi$ for all $p \in \mathcal{P}$,
- $\varphi \in \Phi$ implies that, $\neg\varphi \in \Phi$,
- $\varphi_1, \varphi_2 \in \varphi$ imply that, $\varphi_1 \vee \varphi_2 \in \Phi$.

The *semantics* (or meaning) of a propositional formula is determined by assigning semantics to the atomic propositions in the formula. Towards this, we define a valuation function $v : \mathcal{P} \rightarrow \{true, false\}$. Intuitively, a propositional variable in $\mathcal{P}$ represents a sentence or proposition to be true or false. This valuation v can be easily extended to determine the truth value of a propositional formula using standard semantics of the Boolean connectives.

A valuation v *satisfies* a formula if the formula evaluates to $true$ with respect to the valuation. A formula is *satisfiable* if there exists a valuation that satisfies the formula. As an example, the formula $\varphi = p_1 \wedge (p_2 \vee p_3)$ is satisfiable as the valuation $v(p_1) = true, v(p_2) = true$ satisfies φ. On the other hand, the formula $\psi = \neg p_1 \wedge p_1$ is not satisfiable as there exists no valuation that will satisfy the formula, as every valuation that assigns p_1 to be $true$, assigns $\neg p_1$ to be $false$ and vice-versa.

2.2.2 First-Order Logic

Propositional logic only deals with simple propositions or declarative statements that can either be $true$ or $false$. First-order logic goes beyond—it takes an underlying structure and allows quantifying over elements of that structure. For example, Propositional logic cannot capture statements such as 'There exists an infinite number of prime numbers', which requires the use of first-order logic.

Formally, first-order logic extends propositional logic with a *domain of discourse* $\mathbb{D}$, constants C, a set of predicates $\mathbb{P}$, a set of functions $\mathcal{F}$ and two quantifiers: $\exists$ (existential quantifier) and $\forall$ (universal quantifier). We denote the structure of any first-order theory as $\langle \mathbb{D}; \mathbb{P}; \mathcal{F}; C \rangle$. As an example, the first-order theory of arithmetic $\langle \mathbb{Z}; <; s, +, \times; 0 \rangle$ consists of a single predicate $<$ of arity 2, a single constant 0, a unary function s denoting the successor function and two binary functions $+$ and $\times$ denoting addition and multiplication. The domain of discourse is the set of integers $\mathbb{Z}$.

In a formula in first-order logic, we refer to variables being *free* if they have not been bounded by a quantifier. The values of free variables are not specified within the formula itself and can take on any value from the domain of discourse when the formula is evaluated. A formula that does not have any free variables is called a *sentence*. For example, the above-mentioned statement 'There exists an infinite number

of prime numbers' can be written as a first-order sentence $\exists x \mathrm{Prime(x)} \wedge \forall y \exists z (\mathrm{Prime}(z) \wedge z > y)$. Note that none of the x, y and z variables are free variables, as all of them are bounded by a quantifier.

In the following section, we are going to introduce Presburger arithmetic and some of its decidability results. Note that the general first-order theory of integers $\langle \mathbb{Z}; <; +, \times; 0, 1 \rangle$ is undecidable [38].

2.2.2.1 Presburger Arithmetic

Presburger arithmetic is a formal system developed by Mojżesz Presburger in 1929, which deals with the arithmetic of integers without multiplication. Unlike the general first-order theory of integers with multiplication, Presburger arithmetic is decidable [119].

Formally, *Presburger arithmetic (PA)* is the first-order theory of integers in the structure $\langle \mathbb{Z}, <, +, 0, 1 \rangle$. Let X be a set of infinite first-order variables. A *linear polynomial* over $\vec{x} = (x_1, \ldots, x_n) \in X^n$ is given by the syntax rule

$$p(\vec{x}) ::= \sum_{1 \le i \le n} a_i x_i + b,$$

where a_i, b and x_i's range over $\mathbb{Z}$. Then formulas in Presburger arithmetic can be defined by the following grammar:

$$\varphi ::= \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid p_1(\vec{x}) \le p_2(\vec{x}),$$

where, p_1 and p_2 are linear polynomials. We define the standard Boolean abbreviation $\varphi_1 \vee \varphi_2 \leftrightarrow \neg(\neg\varphi_1 \wedge \neg\varphi_2)$. Moreover we introduce the abbreviations $p_1(\vec{x}) < p_2(x) \leftrightarrow p_1(\vec{x}) \le p_2(\vec{x}) + 1$ and $p_1(\vec{x}) = p_2(\vec{x}) \leftrightarrow p_1(\vec{x}) \le p_2(\vec{x}) \wedge p_2(\vec{x}) \le p_1(\vec{x})$. Given a finite set $X = \{x_1, \ldots, x_n\}$ of first-order variables, we sometimes use a generalised existential quantifier and write $\exists_{x \in X} \varphi(\vec{x})$ to abbreviate the formula $\exists x_1 \ldots x_n \varphi(\vec{x})$.

The *size* $|\varphi|$ of a PA formula φ is defined by structural induction over $|\varphi|$: For a linear polynomial $p(\vec{x})$ we define $|p(\vec{x})|$ as the number of symbols required to write it if the coefficients are given in binary. Then, we define $|\varphi_1 \wedge \varphi_2| \stackrel{\text{def}}{=} |\varphi_1| + |\varphi_2| + 1$, $|\neg\varphi| \stackrel{\text{def}}{=} |\exists x.\varphi| \stackrel{\text{def}}{=} |\varphi| + 1$, $|p_1(\vec{x}) < p_2(\vec{x})| \stackrel{\text{def}}{=} |p_1(\vec{x})| + |p_2(\vec{x})| + 1$.

2.2.3 Temporal Logic

First-order logic helps quantify underlying structures and represent relations between different components of a system. But it does not directly capture the evolution of a CPS over time. Temporal logic introduces additional operators that allow us to reason about time explicitly[1] . For example, we can use temporal logic to express statements like "A

[1]LTL can be expressed in First-order logic, but it is popular as it has nicer complexity results

happens *before* B" or "C *always eventually* leads to D." In the following subsections, we will introduce different temporal logic formalisms, e.g. Linear Temporal Logic (LTL), Metric Temporal Logic (MTL) etc.

2.2.3.1 Linear Temporal Logic

Linear Temporal Logic (LTL) extends propositional logic by adding temporal operators that allow us to reason about events that occur over time. LTL is widely used in computer science, particularly in the verification and synthesis of software and hardware systems. It imposes a linear ordering on the temporal structure of a system. In other words, it assumes that time can be modelled as a sequence of discrete steps that follow a linear order. Before defining its syntax and semantics, we first define *traces*, which represents the discrete temporal structure itself.

Traces. Let $\mathcal{P}$ be a finite set of atomic propositions. An *alphabet* is a finite non-empty set $\Sigma = 2^{\mathcal{P}}$, whose elements are called *symbols*. A *trace* over Σ is a sequence $t = a_1 a_2 \dots$ such that for every $i \geq 1$, $a_i \in \Sigma$. In general, a trace is an infinite sequence. We call a trace a *finite trace* if it is a finite sequence.

For example, let $\mathcal{P} = \{p, q\}$. In the trace $t = \{p, q\} \cdot \{p\} \cdot \{q\} \cdot \{q\} \dots$, both p and q hold at the first position, only p holds in the second position, and then from the third position onward, q holds at every position.

Given a trace $t = a_1 a_2 \dots$ and $1 \leq i \leq j$, let $t[i, j] = a_i \dots a_j$ be the *infix* of t from position i up to and including position j. Moreover, $t[i] = a_i$ is the symbol at the ith position.

Syntax of LTL. The syntax of LTL is defined by the following grammar:

$$\varphi := p \in \mathcal{P} \mid \neg\varphi \mid \varphi \wedge \psi \mid \mathbf{X}\,\varphi \mid \varphi\,\mathbf{U}\,\psi,$$

where $\mathbf{X}$ is the *neXt* operator and $\mathbf{U}$ is the *Until* operator. We use the standard formulas: $true = p \vee \neg p$, $false = p \wedge \neg p$, $\varphi \vee \psi = \neg(\neg\varphi \wedge \neg\psi)$ and $\neg\,\mathbf{X}\,\varphi = \mathbf{X}\,\neg\varphi$. We call the formula (or a fragment) to be in *negation normal form* if $\neg$ is only allowed in front of a proposition. We also define two new operators $\mathbf{F}$ (the *Finally* operator) and $\mathbf{G}$ (the *Globally* operator) as follows:

$$\mathbf{F}\,\varphi \stackrel{\text{def}}{=} true\,\mathbf{U}\,\varphi \text{ ,and } \mathbf{G}\,\varphi \stackrel{\text{def}}{=} \neg\,\mathbf{F}\,\varphi$$

The subformulas $subf(\varphi)$ of an LTL formula φ are defined recursively as follows:

$$\begin{aligned}
subf(p) &= \{p\} \\
subf(\neg\varphi) &= subf(\varphi) \cup \{\neg\varphi\} \\
subf(\mathbf{X}\,\varphi) &= subf(\varphi) \cup \{\mathbf{X}\,\varphi\} \\
subf(\varphi_1 \wedge \varphi_2) &= \{\varphi_1 \wedge \varphi_2\} \cup subf(\varphi_1) \cup subf(\varphi_2) \\
subf(\varphi_1\,\mathbf{U}\,\varphi_2) &= subf(\varphi_1) \cup subf(\varphi_2) \cup (\varphi_1\,\mathbf{U}\,\varphi_2)
\end{aligned}$$

As a syntactic representation of an LTL formula φ, we use *syntax-DAGs*. Syntax-DAGs are essentially similar to the parse-tree of a formula, but the common subformulas and hence the nodes containing them in the tree are shared.

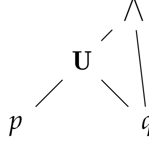


Figure 2.1: Syntax-DAG for the formula $(p \mathbf{U} q) \wedge q$

We define the *size* $|\varphi|$ of an LTL formula φ to be the number of its unique subformulas. For instance, the size of formula $\varphi = (p \mathbf{U} q) \wedge q$ is 4, since the distinct subformulas of φ are $p, q, (p \mathbf{U} q)$ and φ itself. The syntax-DAG of $(p \mathbf{U} q) \wedge q$ also contains 4 nodes, where each Node i denotes the subformula $(p \mathbf{U} q) \wedge q$, starting from that node. It is shown in Figure 2.1.

Semantics of LTL. Formulas in LTL are usually evaluated over infinite traces. To define the semantics of LTL, we introduce the notation $(t, i) \models \varphi$, which denotes that the LTL formula φ holds over trace t from position i. We say that t satisfies φ and we write $t \models \varphi$ when $(t, 1) \models \varphi$. The definition of $\models$ is inductive on the formula φ:

$$
\begin{aligned}
(t, i) \models p \in \mathcal{P} \quad & \Longleftrightarrow \quad p \in t[i] \\
(t, i) \models \neg\varphi \quad & \Longleftrightarrow \quad (t, i) \not\models \varphi \\
(t, i) \models \varphi \wedge \psi \quad & \Longleftrightarrow \quad (t, i) \models \varphi \text{ and } (t, i) \models \psi \\
(t, i) \models \mathbf{X}\,\varphi \quad & \Longleftrightarrow \quad (t, i + 1) \models \varphi \\
(t, i) \models \varphi \,\mathbf{U}\, \psi \quad & \Longleftrightarrow \quad (t, j) \models \psi \text{ for some } j \geq 1 \text{ and } (t, i') \models \varphi \text{ for all } i \leq i' < j
\end{aligned}
$$

We can derive the semantics for the **F** and the **G** operator using their syntactic definition and semantics of the above existing operators:

$$
\begin{aligned}
(t, i) \models \mathbf{F}\,\varphi \quad & \Longleftrightarrow \quad (t, i') \models \varphi \text{ for some } i' \geq i \\
(t, i) \models \mathbf{G}\,\varphi \quad & \Longleftrightarrow \quad (t, i') \models \varphi \text{ for all } i' \geq i
\end{aligned}
$$

In this thesis, we also consider LTLf, a fragment of LTL that is interpreted over finite traces. This has been introduced by Vardi et al. in [63]. We will define LTLf and highlight the changes in its semantics from the standard semantics of LTL in Chapter 3.

2.2.3.2 Metric Temporal Logic

Linear Temporal Logic is used extensively in monitoring and runtime verification of CPS. But most applications that use LTL as a tool opts for a discrete model of time due

to its formalism. But such a model is inadequate for real-time systems as the model of time is dense, i.e., the system is modelled as a sequence of events that are time-stamped with real values, and the system state changes continuously over time.

For example, in a printing machine, we can represent that "every request is eventually followed by a print action" with an LTL formula of the form $\mathbf{G}(\text{request} \rightarrow \mathbf{F}\,\text{print})$. But, one may need to specify that this print action is executed within 10-time units of the corresponding request. This is not possible to be represented as an LTL formula.

Towards this, *Metric Temporal Logic (MTL)* [91] is a widely used formalism that extends LTL to describe dense-time (real-time) properties of reactive systems. Note that MTL has two commonly adopted semantics in literature [116]: *pointwise* and *continuous*. We describe the continuous semantics below that is relevant to this thesis. For pointwise semantics (defined over discrete timed words), we refer the interested readers to [116]. To model the continuous system executions, we define *signals* analogous to traces.

Signals. A *signal* is an infinite time series that describes system behaviours over time. Formally, given a set of atomic propositions $\mathcal{P}$, a signal $\vec{x} := \mathbb{R}_{\geq 0} \rightarrow 2^{\mathcal{P}}$ is a function defined over a continuous time domain. A *finite signal* $\vec{x}_T$ with upper-bound T is similarly denoted by a function with bounded domain $\vec{x}_T : [0, T) \rightarrow 2^{\mathcal{P}}$.

Syntax of MTL. The syntax of MTL is similar to the syntax of LTL but it allows *timed-operators*. Formally the syntax can be defined by the following grammar:

$$\varphi := p \in \mathcal{P} \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \,\mathbf{U}_I\, \varphi_2$$

where $\mathbf{U}_I$ is the timed-Until operator. Here, $I \subseteq (0, \infty)$ is a time interval of non-negative real numbers. All the standard formalisms of *true*, *false*, the $\vee$-operator and the $\rightarrow$ operator are defined analogously to the syntax of LTL. We can also define two more timed operators as follows:

$$\mathbf{F}_I\,\varphi \stackrel{\text{def}}{=} \mathit{true}\,\mathbf{U}_I\,\varphi \text{ ,and } \mathbf{G}_I\,\varphi \stackrel{\text{def}}{=} \neg\,\mathbf{F}_I\,\varphi$$

,

where $\mathbf{F}_I$ is the timed-Finally operator and $\mathbf{G}_I$ is the timed-Globally operator. We rewrite any timed-operator with interval $I = [0, \infty)$ as the untimed operator without interval, e.g., $\mathbf{F}_{[0.\infty)}$ is written as just $\mathbf{F}$.

The subformulas $\mathit{subf}(\varphi)$ of an MTL formula φ are defined recursively as follows:

$$\begin{aligned}
\mathit{subf}(p) &= \{p\}, \\
\mathit{subf}(\neg\varphi) &= \mathit{subf}(\varphi) \cup \{\neg\varphi\}, \\
\mathit{subf}(\varphi_1 \wedge \varphi_2) &= \{\varphi_1 \wedge \varphi_2\} \cup \mathit{subf}(\varphi_1) \cup \mathit{subf}(\varphi_2), \\
\mathit{subf}(\varphi_1 \,\mathbf{U}_I\, \varphi_2) &= \mathit{subf}(\varphi_1) \cup \mathit{subf}(\varphi_2) \cup \{\varphi_1 \,\mathbf{U}_I\, \varphi_2\}.
\end{aligned}$$

Similar to LTL, we use *syntax-DAGs* as a syntactic representation of an MTL formula. We define the size $|\varphi|$ of an MTL formula φ to be the number of its unique subformulas. This again similarly corresponds to the number of nodes in its syntax-DAG. For instance, the size of formula $\varphi = (p\, \mathbf{U}_I\, q) \wedge q$ is 4.

Semantics of MTL. Typically, MTL formulas are interpreted over infinite signals. Given a signal $\vec{x}$ and a time-point t, the semantics of MTL over infinite signal is defined recursively as follows:

$$\begin{aligned}
(\vec{x}, t) &\models p && \iff p \in \vec{x}(t) \\
(\vec{x}, t) &\models \neg\varphi && \iff (\vec{x}, t) \not\models \varphi \\
(\vec{x}, t) &\models \varphi_1 \wedge \varphi_2 && \iff (\vec{x}, t) \models \varphi_1 \text{ and } (\vec{x}, t) \models \varphi_2 \\
(\vec{x}, t) &\models \varphi_1\, \mathbf{U}_I\, \varphi_2 && \iff \exists t' \in t + I \text{ such that } (\vec{x}, t') \models \varphi_2 \\
& && \quad\; \text{and } \forall t'' \in [t, t'],\ (\vec{x}, t'') \models \varphi_1
\end{aligned}$$

The semantics of $\mathbf{F}_I$ and $\mathbf{G}_I$ can be derived as follows:

$$\begin{aligned}
(\vec{x}, t) &\models \mathbf{F}_I\, \varphi && \iff \exists t' \in t + I \text{ such that } (\vec{x}, t') \models \varphi \\
(\vec{x}, t) &\models \mathbf{G}_I\, \varphi && \iff \forall t' \in t + I, (\vec{x}, t') \models \varphi
\end{aligned}$$

Continuing from the previous example, now the specification "every request is eventually followed by a print action within 10-time units" can be written as the following MTL formula:

$$\mathbf{G}(\text{request} \rightarrow \mathbf{F}_{[0,10]}\, \text{print})$$

Note that we replaced $\mathbf{G}_{[0,\infty)}$ by only $\mathbf{G}$ in the formula, as mentioned above.

2.2.3.3 Signal Temporal Logic

Signal Temporal Logic (STL) is a temporal logic formalism for specifying properties of continuous *real-valued signals*. It extends MTL with real-valued predicates instead of propositions. The syntax of STL is similar to MTL:

$$\varphi := \mathit{true} \mid \pi \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1\, \mathbf{U}_I\, \varphi_2$$

Here π is a predicate of the form $f_\pi(\vec{x}) > 0$ where, $f_\pi : \mathbb{R}^n \rightarrow \mathbb{R}$ is a real-valued function. We define the standard-derived operators similar to MTL. The subformulas and the size of an STL formula are also defined analogously to MTL.

Typically, STL formulas are interpreted over real-valued signals. Formally, an n-dimensional real-valued signal $\vec{x} := \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^n$ is a function defined over a continuous time domain. Intuitively, MTL is interpreted over *Boolean signals*. The semantics of STL are also analogous to MTL, with the only key difference being the interpretations over a predicate (instead of propositions):

$$(\vec{x}, t) \models \pi \iff f_\pi(x_1(t), \ldots, x_n(t)) > 0$$

2.3 Graph, Automata, and Languages

In this section, we introduce the fundamental concepts of graphs, automata, and languages, exploring their intricate connections and applications in various fields.

2.3.1 Graphs and Flows

A *graph* is a mathematical structure that consists of a set of nodes, along with a set of edges that connect pairs of nodes. It is typically represented as $G = (V, E)$, where V represents the set of nodes and $E \subseteq V \times V$ represents the set of edges. A *directed graph* is a graph where every edge has a direction associated to them, i.e. each edge $(u, v) \in E$ is an ordered pair, where the *predecessor* or the *parent* is u, and the *successor* or the *child* is v. A node $u \in V$ has a self-loop if $(u, u) \in E$. A graph is *weighted* if there is a weight function $\mathrm{weight} : E \to \mathbb{Z}$ that assigns a weight to each edge. A *skew-transpose* of a weighted directed graph G can be obtained by multiplying each weight by -1 and reversing the direction of each edge.

A *path* is a sequence of nodes $v_0v_1\ldots$ where v_{i+1} is a successor of v_i, for all $i \geq 0$; A path is *finite* if the sequence is finite. A path is said to be *simple* if $v_i \neq v_j$ for all $0 \leq i < j$. If a path contains a node v, we say that the path 'visits' or 'reaches' v; Given two nodes u and v, we say that $v \in V$ is reachable from $u \in V$, if there is a finite path starting from u that visits v. A finite path $v_0v_1\ldots v_n$ is a *cycle* if $v_0 = v_n$. It is a *simple* cycle if $\forall 0 \leq i < j \leq n, v_i \neq v_j$, when $i \neq 0$ or $j \neq n$.

Let us denote the set of children of $v \in V$ by $vE := \{w \in V \mid (v, w) \in E\}$ and the parents of v by Ev. Then, an *s–t flow* is a mapping $f : E \to \mathbb{N}$ that satisfies flow conservation, i.e., $\forall v \in V \setminus \{s, t\} : \sum_{u\in Ev} f(u, v) = \sum_{u\in vE} f(v, u)$. That is, the total incoming flow equals the total outgoing flow for all but the source (s) and the target (t) vertices. We then define the *value* of a flow f as: $|f| \stackrel{\text{def}}{=} \sum_{v\in sE} f(s, v) - \sum_{u\in Es} f(u, s)$., where s is the source. We denote by *support*(f) the set $\{e \in E \mid f(e) > 0\}$ of edges with a non-zero flow. A cycle in a flow f is a cycle in the sub-graph induced by $\mathrm{support}(f)$. For weighted graphs, we define $\mathrm{weight}(f) \stackrel{\text{def}}{=} \sum_{e\in E} f(e) \cdot \mathrm{weight}(e)$. The following is known as Euler's theorem for directed graphs.

Given a directed graph $G = (V, E)$, consider a path $\pi = v_0v_1\ldots$ in G. We denote *path-flow* of the path π (f_π) by its *Parikh image*, i.e., f_π maps each edge e to the number of times e occurs in π. A flow f is called a *path-flow* if there exists a path π such that $f = f_\pi$. Finally, we observe that an s–t path-flow f in G induces a t–s path-flow f' with $f'(u, v) = f(v, u)$, for all $(u, v) \in E$, in the *skew transpose* of G.

Theorem 1 (Euler's theorem) *Let f be an s–t flow. Then, there exists an s–t path π such that $f = f_\pi$ if and only if $|f| = 1$ and the sub-graph induced by the set of edges $\mathrm{support}(f) \cup \{(t, s)\}$ is strongly connected.*

2.3.2 Automata and Languages

An *automaton* is a directed graph with every edge containing a label from a particular finite set, which we call an *alphabet*. Formally, an automaton $\mathcal{A} = \{Q, \delta, \Sigma, q_{in}, F\}$ is a tuple where Q is a finite set of *states*, Σ is a finite alphabet, $q_{in} \in Q$ is an initial state, $F \subseteq Q$ is a set of final states, and $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation. Intuitively, (Q, δ) is a directed graph with each edge labelled by a *letter* $a \in \Sigma$. An automaton is *deterministic* (DFA) if for every $q \in Q$ and $a \in \Sigma$, we have a unique $q' \in Q$ such that, $(q, a, q') \in \Delta$. Otherwise, the automaton is *non-deterministic* (NFA).

An automaton can be viewed as an abstract model of a CPS, where each state represents a particular system state, and each transition denotes an event that evolves from one system state to another. In literature, automata are also viewed as a 'language acceptor' by defining a language that describes the desired or expected system inputs and outputs and using an automaton to accept or reject sequences of events or states based on whether they conform to the specified language. We first define the concept of a language below.

Let an alphabet Σ be a finite set of letters. A *word* $w \in \Sigma$ is defined as a sequence $w = a_0 a_1 \ldots$ such that, $a_i \in \Sigma$. A word is *finite* or *infinite* based on the sequence being finite or infinite. The special *empty word* is denoted by ε. We denote by Σ^* the set of all finite words in Σ, i.e. the set $\{a_0 \ldots a_n \mid a_i \in \Sigma \text{ for all } 0 \leq i \leq n\}$. A language $L \subseteq \Sigma^*$ over Σ is a set of finite words. A language is *regular* if it is defined recursively as follows:

- The empty language $\emptyset$ and the the language only containing empty word $\{\varepsilon\}$ are regular;
- The singleton language $\{a\}$ for every $a \in \Sigma$ is regular;
- Union and concatenation of two regular languages are regular;
- L^* is regular for every language L;
- No other language over Σ is regular.

Now, we can define automata as language acceptors using their initial and final states. A *run* ρ of the automaton $\mathcal{A}$ (defined as above) over a word $w = a_0 a_1 \ldots a_{n-1}$ is a sequence $q_0 a_0 q_1 a_1 q_2 \ldots a_{n-1} q_n$ where $q_0 = q_{in}$. We say that ρ is accepting if $q_n \in F$. $\mathcal{A}$ accepts a word w if there is an accepting run of $\mathcal{A}$ over w. $\mathcal{A}$ accepts a language L if for every word $w \in L$, w is accepted by $\mathcal{A}$ and for every word $w' \notin L$, $\mathcal{A}$ does not accept w'. The language accepted by finite automata is exactly the set of regular languages (Kleene's theorem [87]).

2.3.3 Büchi Automata and Connection to LTL

Büchi Automata extends finite-state automata (described above) to accept languages of infinite words. A non-deterministic Büchi automata are syntactically equivalent to NFA. Formally, a Büchi automaton is a tuple $\mathcal{A} = \{Q, \delta, \Sigma, q_{in}, F\}$ where every component is similarly defined as in NFA.

A run ρ of a Büchi automaton $\mathcal{A}$ over a infinite word $w \in \Sigma^\omega = a_0 a_1 \dots$ is an infinite sequence $q_0 a_0 q_1 a_1 \dots$ where $q_0 = q_{in}$. Let $inf(\rho) = \{q \mid \forall n \in \mathbb{N}, \exists m \geq n, q_m = q\}$ denotes the set of states that appear infinitely often along the run ρ. We say ρ is accepting if $inf(\rho) \cap F \neq \emptyset$. The acceptance conditions for a word w, and a language L by a Büchi automaton $\mathcal{A}$ are defined similarly as in NFA. The concept of determinism also extends naturally.

A *generalised Büchi automaton* is exactly similar to a Büchi automaton. The only difference is that, instead of having a set of final states F, a generalised Büchi automaton has a set of sets of final states $\mathcal{F} \subseteq 2^Q$. The acceptance condition becomes: $inf(\rho) \cap F \neq \emptyset$ for all $F \in \mathcal{F}$. They are expressively equivalent to Büchi automata as described in the lemma below.

Lemma 1 *For any generalised Büchi automaton $\mathcal{A}$, one can construct in polynomial time a Büchi automaton $\mathcal{A}'$ such that $L(\mathcal{A}) = L(\mathcal{A}')$.*

Now the following theorem along with Lemma 1 establishes the fundamental connection between LTL and Büchi automata that we exploit later.

Theorem 2 *[From [138]] Let φ be an LTL formula over a set of propositions $\mathcal{P}$. One can construct a generalised Büchi automaton $\mathcal{A}$ such that for any word $w \in (2^{\mathcal{P}})^\omega$, $w \in L(\mathcal{A})$ if and only if $w, 0 \models \varphi$. The size of $\mathcal{A}$ is at most singly exponential in the size of φ.*

Part I

Learning Specifications

Chapter 3

Learning Formulas in LTL

Complex black-box systems are ubiquitous in the field of Artificial Intelligence. Extensive use of these systems in safety-critical scenarios has created a need for learning human-interpretable explanations of these systems. Towards this, Linear Temporal Logic (LTL) is a popular formalism due to its resemblance with natural language. This chapter focuses on learning LTL specifications from observed system behaviours. The content presented in this chapter is based on the work in [122].

3.1 Background

Linear Temporal Logic (LTL) is a prominent logic for specifying temporal properties [118]. It is commonly used in many fields, such as model checking, program analysis, and motion planning for robotics. Learning temporal logic has emerged as a crucial research area in artificial intelligence in the past decade. It addresses the challenging task of constructing interpretable models from data. As argued in the literature, e.g., by [34] and [129], LTL formulas are mathematically rigorous and typically easy to interpret by human users and, therefore, useful as explanations. Learning LTL specifications is being extensively applied at the field of program specification [96], anomaly and fault detection [25], robotics [37], and many more: we refer to Section 7 of [34], for a list of practical applications. An equivalent point of view on LTL learning is a specification mining question. The ARSENAL [62] and FRET [64] projects construct LTL specifications from natural language, we refer to [98] for an overview.

LTL has also been studied over finite traces, and the fragment is called LTLf introduced by Vardi et al. [63]. This fragment is beneficial for synthesizing specifications from finite simulations of black-box systems and then using it for formal run-time verification of infinite simulations of the system. Hence, LTLf has been a de facto standard in specification learning and analyzing complex systems. Due to its variable-free syntax and inherent inductive semantics, LTLf is a suitable candidate for constructing classifiers that distinguish positive traces from negative ones. We will formally present the syntax and the semantics of LTLf later in this chapter.

The fundamental problem we study here is to learn a specification in the form of an LTLf formula from a set of positive and negative system behaviours. More formally (we

refer to the next section for formal definitions), given a set $u_1, \ldots, u_n$ of positive behaviours and a set $v_1, \ldots, v_n$ of negative behaviours, the goal is to construct a formula φ of LTLf which satisfies all u_is and none of the v_is. We say that φ is a separating formula or—using machine learning terminology—a classifier.

To make things concrete, let us introduce our running example for this chapter, a classic motion planning problem in robotics inspired by [69]. A robot collects waste bin contents in an office-like environment and empties them into a trash container. Let us assume that there is an office o, a hallway h, a container c, and a wet area w. The following are possible traces obtained in experimentation with the robot (for instance, through simulation):

$$
\begin{aligned}
u_1 &= \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{o} \cdot \mathrm{h} \cdot \mathrm{c} \cdot \mathrm{h} \\
v_1 &= \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{c} \cdot \mathrm{h} \cdot \mathrm{o} \cdot \mathrm{h} \cdot \mathrm{h}
\end{aligned}
$$

In LTLf learning, we start from these labelled data: given u_1 as positive and v_1 as negative, what is a possible classifier including u_1 but not v_1? Informally, v_1 being negative implies that the order is fixed: o must be visited before c. We look for classifiers in the form of separating formulas, for instance

$$\mathbf{F}(\mathrm{o} \wedge \mathbf{F}\,\mathbf{X}\,\mathrm{c}),$$

Note that this formula requires the robot to visit the office first and only then visit the container.

Assume now that two more negative traces were added:

$$
\begin{aligned}
v_2 &= \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{o} \cdot \mathrm{w} \cdot \mathrm{c} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \\
v_3 &= \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{w} \cdot \mathrm{o} \cdot \mathrm{w} \cdot \mathrm{c} \cdot \mathrm{w} \cdot \mathrm{w}
\end{aligned}
$$

Then the previous separating formula is no longer correct, and a possible separating formula is

$$\mathbf{F}(\mathrm{o} \wedge \mathbf{F}\,\mathbf{X}\,\mathrm{c}) \wedge \mathbf{G}(\neg \mathrm{w}),$$

which additionally requires the robot never to visit the wet area.

3.1.1 Related Works

The task of inferring temporal logic formulas consistent with a given data has been studied extensively [25, 88, 140, 141]. Most of these works place limitations on the inferred formula's syntax. These methods typically generate formulas using predefined templates, which has some disadvantages. Crafting these templates can be challenging for users as it requires substantial knowledge of the underlying system. Secondly, limiting the formula's structure could potentially result in larger inferred formulas.

Several approaches to learning LTL formulas also exist without predefined templates in the literature. To learn separating LTL formulas, Neider and Gavran [113] leverage SAT solvers by reducing the learning problem to the Boolean satisfiability problem. Similarly, Camacho et al. [34] use the SAT solver to infer Alternating Finite Automaton as a classifier and then extract the separating LTLf formulas from it. Arif et al. [6] learn Past LTLf formulas from data using the bit-vector function synthesis problem in a

SyGuS (Syntax-Guided Synthesis) solver, while Kim et al. [84], uses Bayesian inference techniques to tackle the same problem. These techniques have been extended to learn more expressive logic also, such as Property Specification Language (PSL) [129] and Computational Tree Logic (CTL) [52]. Deploying the above existing methods for industrial cases is challenging as they can only handle formulas up to a small size. These methods can consume significant computational resources and time without producing any output. Even for small fragments of LTLf, constructing the minimal formula has been proven to be NP-hard based on theoretical studies [58], which sheds light on the practical difficulties encountered while scaling up these methods.

3.1.2 Our Approach

In this chapter, we discuss a novel algorithm for learning LTLf specification from system behaviours. To address the above-mentioned challenges, we propose *approximation* and *anytime* algorithms. Note that the approximation factor is on the size of the inferred LTLf formula, not on the separation of positive and negative system simulations, i.e., our output formula still satisfies all the positive behaviours and rejects all the negative ones, but it may not be the smallest such formula. On the other hand, an algorithm solving an optimisation problem is called *anytime* if it finds better and better solutions the longer it keeps running. This algorithm can be interrupted 'anytime' during its computation and yield some good albeit non-optimal solutions. In other words, anytime algorithms work by refining solutions. As we will also show in the experiments later, this implies that even if our algorithm timeouts, it may yield some concise separating formula. Note that our algorithm targets a strict fragment of LTLf, which does not contain the Until operator (nor its dual Release operator). It combines two ingredients:

- *Searching for directed formulas*: We define a space-efficient dynamic programming algorithm for enumerating formulas from a fragment of LTLf that we call Directed LTL.
- *Combining directed formulas*: We construct two algorithms for combining formulas using Boolean operators. The first is an off-the-shelf *decision tree algorithm*, and the second is a new greedy algorithm called *Boolean subset cover*.

The two ingredients yield two subprocedures: the first one finds directed formulas of increasing size, which are then fed to the second procedure in charge of combining them into a separating formula. This method yields an anytime algorithm as both subprocedures can output separating formulas even with a low computational budget and refine them over time.

Let us illustrate the two subprocedures in our running example. The first procedure enumerates so-called *directed formulas* in increasing size; we refer to the corresponding section for a formal definition. The directed formulas $\mathbf{F}(\mathrm{o} \wedge \mathbf{F}\,\mathbf{X}\,\mathrm{c})$ and $\mathbf{G}(\neg \mathrm{w})$ have small size hence will be generated early on. The second procedure constructs formulas as Boolean combinations of directed formulas. Without getting into the details of the algorithms, let us note that both $\mathbf{F}(\mathrm{o} \wedge \mathbf{F}\,\mathbf{X}\,\mathrm{c})$ and $\mathbf{G}(\neg \mathrm{w})$ satisfy u_1. The first does not satisfy v_1, and the second does not satisfy v_2 and v_3. Hence their conjunction $\mathbf{F}(\mathrm{o} \wedge \mathbf{F}\,\mathbf{X}\,\mathrm{c}) \wedge \mathbf{G}(\neg \mathrm{w})$ is separating, meaning it satisfies u_1 but none of v_1, v_2, v_3.

3.2 Outline of the Chapter

We introduce the fragment LTLf in Section 3.3. Then we formally describe the learning problems in Section 3.4. We elaborate on our algorithm in Section 3.5. In Section 3.6, we show how we can adapt our algorithm for noisy learning. We prove all the theoretical results related to the algorithms in Section 3.7. Finally, Section 3.8 contain all the experimental results and evolution of our algorithms.

3.3 LTL on Finite Traces

This section will formally define the fragment of LTL of finite traces (LTLf). Recall the definition of *trace* from Section 2.2.3.1, which characterises discrete system behaviours. Let us fix the set of propositions to be $\mathcal{P}$ and the alphabet to be $\Sigma = 2^{\mathcal{P}}$. A finite *trace* over Σ is a finite sequence $t = a_1 a_2 \ldots a_n$ such that for every $1 \leq i \leq n$, $a_i \in \Sigma$. We say that t has length n and write $|t| = n$. For convenience, we refer to finite traces as traces only in the rest of the chapter unless mentioned otherwise.

In this chapter, we are interested in the fragment of LTLf. The syntax of LTLf is exactly similar to the syntax of LTL (see Section 2.2.3.1). It contains the standard LTL operators: ne**X**t, **U**ntil, **F**inally and **G**lobally operators, the boolean connectives: $\wedge$ and $\vee$, the $\neg$-operator and the standard formulas: *true* and *false* defined analogously. In addition, we have an additional operator **last** $= \neg\, \mathbf{X}\, true$, which denotes the last position of the trace. As a shorthand, we use $\mathbf{X}^n\, \varphi$ for $\underbrace{\mathbf{X} \ldots \mathbf{X}}_{n \text{ times}} \varphi$. The *size of a formula* is the size of its underlying syntax-DAG.

Formulas in LTLf are evaluated over finite traces. The semantics of LTLf is similar to the semantics of standard LTL semantics:

$$
\begin{aligned}
(t,i) &\models p \in \mathcal{P} && \iff p \in t[i] \\
(t,i) &\models \neg\varphi && \iff (t,i) \not\models \varphi \\
(t,i) &\models \varphi \wedge \psi && \iff (t,i) \models \varphi \text{ and } (t,i) \models \psi \\
(t,i) &\models \mathbf{X}\,\varphi && \iff i < |t| \text{ and } (t,i+1) \models \varphi \\
(t,i) &\models \varphi\,\mathbf{U}\,\psi && \iff (t,j) \models \psi \text{ for some } i \leq j \leq |t| \text{ and } (t,i') \models \varphi \text{ for all } i \leq i' < j \\
(t,i) &\models \mathbf{F}\,\varphi && \iff (t,i') \models \varphi \text{ for some } i \leq i' \leq |t| \\
(t,i) &\models \mathbf{G}\,\varphi && \iff (t,i') \models \varphi \text{ for all } i \leq i' \leq |t|
\end{aligned}
$$

We say that t satisfies φ and we write $t \models \varphi$ when $(t,1) \models \varphi$. Note that our algorithm does not include the **U**ntil operator. We discuss this in more detail at the end of this chapter. For reading convenience, we refer to the fragment of LTLf without the **U**ntil operator as LTLf only throughout the rest of this chapter.

3.4 LTLf Learning Problem: Exact and Noisy

The LTLf exact learning problem we study here is in the *passive learning* setting: models are learnt based on a given data set. First, we define the input of our problem formulation, which we call a *Sample*.

Definition 1 (Sample) *A* sample *consists of a set of labelled (finite) signal prefixes. Formally, we rely on a sample $\mathcal{S} = (P, N)$ consisting of P, a set of positive signal prefixes, and N, a set of negative signal prefixes. We say a sample is* informative *if $P \cap N = \emptyset$. We say an LTLf formula φ is separating for $\mathcal{S}$ if it satisfies all the positive traces in P and does not satisfy any negative traces in N.*

There are two relevant parameters for a sample: its *size* $|\mathcal{S}| = |P| + |N|$, which is the number of traces, and its *length* $len(\mathcal{S}) = \max\{len(t) \mid t \in P \cup N\}$, which is the maximum length of all traces. Now, we define our exact learning problem as follows:

Problem 1 (LTLf exact learning) *Given a sample $\mathcal{S} = P \cup N$, construct a minimal LTLf formula φ that is separating for $\mathcal{S}$ i.e., $t \models \varphi$ for all $t \in P$ and $t \not\models \varphi$ for all $t \in N$.*

In practical scenarios, noise in data is omnipresent, i.e., they are misclassified as positive or negative behaviours of the underlying system. Exact learning algorithms on these noisy samples would often result in overfitting on the particular input data, and the extracted LTLf specifications might not be suitable to interpret the system properly. Hence, the problem is naturally extended to the LTLf noisy learning problem where the goal is to infer a separating LTLf formula with *low loss*, where loss indicates the fraction of the sample misclassified by the formula. Given a sample $\mathcal{S} = (P, N)$ and a formula φ, let us define the loss function $loss(\mathcal{S}, \varphi) = \frac{\mathrm{mc}(\mathcal{S},\varphi)}{|\mathcal{S}|}$ where,
$\mathrm{mc}(\mathcal{S}, \varphi) = |\{t \not\models \varphi \mid t \in P\}| + |\{t \models \varphi \mid t \in N\}|$ denotes the number of traces that φ misclassifies. Given a threshold ε, an LTLf formula φ is a ε-separating formula if $loss(\mathcal{S}, \varphi) \leq \epsilon$. Based on this definition of the loss function, we define our LTLf noisy learning problem as follows:

Problem 2 (LTLf noisy learning) *Given a sample $\mathcal{S}$ and a threshold $\epsilon \leq 1$, construct a minimal LTLf separating formula φ for $\mathcal{S}$ such that, $loss(\mathcal{S}, \varphi) \leq \epsilon$.*

Next, we present an algorithm for solving the LTLf exact learning problem; we later sketch how to adapt it to the noisy setting.

3.5 The Exact Learning Algorithm for LTLf

Let us first sketch a naive algorithm for the LTLf exact learning problem. Given a sample, we search through all LTLf formulas in the order of their size and check whether they are separating for $\mathcal{S}$ or not. Checking whether an LTLf formula is separating can be done using standard methods (e.g. using bit vector operations [11]). The major disadvantage of this naive method is scalability, as the number of LTLf

formulas multiplies. In face, the number of LTLf formulas of size k is asymptotically equivalent to $\frac{\sqrt{14}\cdot 7^k}{2\sqrt{\pi k^3}}$ [59].

To reduce the search space, instead of the entire LTLf fragment, our algorithm (as outlined in Algorithm 1) performs an iterative search through a fragment of LTLf, which we call *Directed LTL* (Line 4). Our algorithm runs through two stages. We first iteratively generate these Directed LTL formulas in a particular "size order" (not the usual size of an LTLf formula) and evaluate these formulas over the traces in the sample efficiently using dynamic programming techniques. Then, we store the "most promising" Directed LTL formulas based on a score function and generate and search through Boolean combinations of them (Line 11).

Algorithm 1 Overview of our algorithm

```
1:  F ← ∅
2:  ψ ← ∅: best formula found
3:  for all s in "size order" do
4:      D ← all Directed LTL formulas of parameter s
5:      for all φ ∈ D do
6:          if φ is separating and smaller than ψ then
7:              ψ ← φ
8:          end if
9:      end for
10:     F ← F ∪ D
11:     F ← Boolean combinations of the promising formulas in F
12:     for all φ ∈ F do
13:         if φ is separating and smaller than ψ then
14:             ψ ← φ
15:         end if
16:     end for
17: end for
18: Return ψ
```

When searching formulas, if we find a separating LTLf formula ψ, we set $|\psi|$ as an upper bound and continue our search for separating formulas with size ψ. Intuitively, this gives rise to the anytime property of our algorithm, meaning that it searches for better (smaller size) LTLf formulas at each iteration than the previously found ones, if any. This heuristic, along with aiding the search for minimal formulas, also reduces the search space significantly. In the following sections, we will elaborate on these two stages of our algorithm.

3.5.1 Searching for Directed LTL

We first define the fragment of LTLf that we introduce to reduce our search space. We call this fragment of LTLf as *Directed LTL*.

Definition 2 (Directed LTL) *Let us define a partial symbol as a conjunction of positive or negative atomic propositions. Given a set of propositions $\mathcal{P}$, a partial symbol can be defined by*

the following grammar:

$$s := p \in \mathcal{P} \mid \neg p \mid s \wedge s$$

Then, Directed LTL is defined by the following grammar:

$$\varphi := \mathbf{X}^n\, s \mid \mathbf{F}\,\mathbf{X}^n\, s \mid \mathbf{X}^n(s \wedge \varphi) \mid \mathbf{F}\,\mathbf{X}^n(s \wedge \varphi),$$

where s *is a partial symbol and* $n \in \mathbb{N}$.

For example, let $s = p \wedge q \wedge \neg r$ for the partial symbol specifying that p and q hold and r does not. A partial symbol's *width* is the number of atomic propositions it uses, e.g. $\text{width}(s) = 3$. Now an example of a Directed LTL formula will be

$$\varphi = \mathbf{F}((p \wedge q \wedge \neg r) \wedge \mathbf{F}\,\mathbf{X}^2\, \neg p)$$

where $p \wedge q \wedge \neg r$ and $\neg p$ will be two partial symbols. The formula reads that a position exists satisfying $p \wedge q$ and does not satisfy r, and at least two positions later, a position is satisfying $\neg p$. The length of a Directed LTL formula is the number of partial symbols it contains, and its width is the maximum of the widths of those partial symbols. For example, the length and the width of φ as defined above will be 2 and 3, respectively.

The intuition behind the naming of Directed LTL is that any formula in this fragment imposes an order ('direction') in which the partial symbols may occur in a trace that satisfies it. For example, consider a Directed LTL formula $\varphi = \mathbf{F}(p \wedge \mathbf{F}\, q)$ and a non-Directed LTL formula $\psi = \mathbf{F}\, p \wedge \mathbf{F}\, q$. Note that, in any trace t such that $t \models \varphi$, the partial symbol q can only occur at the same position or a later position where p occurs. Now, let $t_1 = \{p\}\{q\}$ and $t_2 = \{q\}\{p\}$ be two traces such that, p and q appear in different orders in them. It is easy to see that $\psi \models t_1$ and $\psi \models t_2$, meaning that it does not impose any order on the occurrences of its partial symbols.

Note that Directed LTL only uses the **X** and **F** operators as well as conjunctions and atomic propositions. We describe the dynamic programming algorithm to generate these Directed LTL formulas from the given sample $\mathcal{S}$.

3.5.1.1 Generating Directed LTL Formulas

We consider the following problem: given the sample $\mathcal{S}$, we want to generate all Directed LTL formulas and a list of traces in $\mathcal{S}$ that satisfy them. Our first technical contribution and key to the scalability of our approach is an efficient solution to this problem based on dynamic programming.

Search Ordering. First, we define the iteration order in which we generate Directed LTL formulas. The iteration order is $\langle(\ell, w)\rangle$, where ℓ denotes the length of the Directed LTL formulas and w denotes their width. Our choice of this iteration order allows us to efficiently construct a dynamic programming algorithm to generate Directed LTL formulas of length ℓ and width w from already constructed formulas at previous iterations.

As our search order has two parameters, we can define various total orders on them. In this work, we fix the order to be the standard bijection from $\mathbb{N}^2 \mapsto \mathbb{N}$ as follows:

$$(1,1),(2,1),(1,2),(3,1),(2,2),(1,3),\ldots$$

In practice, slightly more complicated orders on pairs are helpful since we want to increase the length more often than the width. Nevertheless, we use this as a heuristic in our implementation and fix the above order for presentation purposes for the rest of this chapter.

Preprocessing. Before proceeding toward the dynamic programming algorithm, a useful idea is to change the representation of the set of traces in the sample by precomputing the lookup table Index defined as follows: where t is a trace in S, s a partial symbol, and i in $[1,|t|]$:

$$\textsc{Index}(t,s,i) = \{j \in [i+1,|t|] : t[j] \models s\}\,.$$

Intuitively, for position i, partial symbol s, and a trace t, the table Index stores the next set of positions in the t, at which s holds. For example, let $t = \{p\}\{\emptyset\}\{p,q\}\{p\}$ and the partial symbol be the proposition p. Then, $\textsc{Index}(t,p,1) = \{3,4\}$.

The table Index can be precomputed in linear time from S and aids the formulation of the dynamic programming algorithm.

Dynamic Programming Algorithms. Once we fix the order and complete the preprocessing step, we recursively implement two enumeration procedures:

- *Length Increase Procedure*: we generate Directed LTL formulas of length $\ell+1$ and width w from previously generated set of Directed formulas at iteration (ℓ, w).
- *Width Increase Procedure*: we generate Directed LTL formulas of length ℓ and width $w+1$ from previously generated set of Directed formulas at iteration (ℓ, w) and at iteration $(\ell, 1)$.

While generating the Directed LTL formulas, we aim to keep track of the set of traces in the sample they satisfy. Evaluating every generated formula on each trace in a naive way is expensive. Toward this, we exploit a dynamic programming table called LastPos. We define it as follows, where φ is a directed formula and t a trace in S:

$$\textsc{LastPos}(\varphi,t) = \{i \in [1,|t|] : t[1,i] \models \varphi\}\,.$$

The table intuitively stores all the prefixes of t that satisfy φ. Note that, $t \models \varphi$ if and only if $|t| \in \textsc{LastPos}(\varphi,t)$. The main benefit of LastPos is that it meshes well with the generation of Directed LTL formulas: it is algorithmically easy to compute them recursively on the structure of the formulas.

We present our algorithm for generating Directed LTL formulas in Algorithm 2. This contains both the procedures for increasing a formula's length and width.

Algorithm 2 Generation of Directed LTL formulas

```
1:  procedure SEARCH DIRECTED LTL FORMULAS – LENGTH INCREASE(ℓ, w)
2:      for all Directed LTL formulas φ of length ℓ and width w do
3:          for all partial symbols s of width at most w do
4:              for all t ∈ S do
5:                  I = LASTPOS(φ, t)
6:                  for all i ∈ I do
7:                      J = INDEX(t, s, i)
8:                      for all j ∈ J do
9:                          φ_{=j} ← s ∧_{=(j−i)} φ
10:                         add j to LASTPOS(φ_{=j}, t)
11:                     end for
12:                     for all j′ ≤ max(J) do
13:                         φ_{≥j′} ← s ∧_{≥(j−i)} φ
14:                         add J ∩ [j′, |t|] to LASTPOS(φ_{≥j′}, t)
15:                     end for
16:                 end for
17:             end for
18:         end for
19:     end for
20: end procedure
21:
22: procedure SEARCH DIRECTED LTL FORMULAS – WIDTH INCREASE(ℓ, w)
23:     for all Directed LTL formulas φ of length ℓ and width w do
24:         for all Directed LTL formulas φ′ of length ℓ and width 1 do
25:             if φ and φ′ are compatible then
26:                 φ″ ← φ ⩑ φ′
27:                 for all t ∈ S do
28:                     LASTPOS(φ″, t) ← LASTPOS(φ, t) ∩ LASTPOS(φ′, t)
29:                 end for
30:             end if
31:         end for
32:     end for
33: end procedure
```

For the length increase algorithm, we define two extension operators $\wedge_{=k}$ and $\wedge_{\geq k}$ that "extend" the length of a Directed LTL formula φ by including a partial symbol s in the formula. Precisely, the operator $s \wedge_{=k} \varphi$ replaces the rightmost partial symbol s' in φ with $(s' \wedge \mathbf{X}^k\, s)$, while $s \wedge_{\geq k} \varphi$ replaces s' with $(s' \wedge \mathbf{F}\, \mathbf{X}^k\, s)$. Note that the resulting formulas will still be in the Directed LTL fragment and will have their lengths increased by one. For instance, $c \wedge_{=2} \mathbf{X}(a \wedge \mathbf{X}\, b) = \mathbf{X}(a \wedge \mathbf{X}(b \wedge \mathbf{X}^2\, c))$; the formula $\mathbf{X}(a \wedge \mathbf{X}\, b)$ has length 2 and the resulting formula $\mathbf{X}(a \wedge \mathbf{X}(b \wedge \mathbf{X}^2\, c))$ has length 3.

For the width increase algorithm, we say that two directed formulas are *compatible* if they are equal except for partial symbols. For two compatible formulas, we define a *pointwise-and* ($\curlywedge$) operator that takes the conjunction of the corresponding partial symbols at the same positions. Similar to the length increase procedure, note that the resulting formula is a Directed LTL formula with its width increased by 1. For instance,

$\mathbf{X}(a \wedge \mathbf{X}\, b) \wedge \mathbf{X}(b \wedge \mathbf{X}\, c) = \mathbf{X}((a \wedge b) \wedge \mathbf{X}(b \wedge c))$. Both the formulas $\mathbf{X}(a \wedge \mathbf{X}\, b)$ and $\mathbf{X}(b \wedge \mathbf{X}\, c)$ have width 1, and the resulting formula $\mathbf{X}((a \wedge b) \wedge \mathbf{X}(b \wedge c))$ has width 2.

Line 14 and Line 28 demonstrate the process of updating the table LastPos for each procedure. Thus we generate Directed LTL formulas along with tracking the set of traces they satisfy efficiently.

Practical Optimisations. The actual implementation of the algorithm refines the algorithms in certain places. For instance:

- Line 3: instead of considering all partial symbols, we restrict to those appearing in at least one positive trace.
- Line 13: some computations for $\varphi_{\geq j}$ can be made redundant; a finer data structure factorises the computations.
- Line 25: using a refined data structure, we only enumerate compatible directed formulas.

3.5.1.2 The Dual Point of View: Capturing the G operator

The Directed LTL fragment only consists of **F**, **X** and the $\wedge$-operator. In particular, it does not contain the **G** operator. To capture the same, we aim to capture formulas in a dual fragment to Directed LTL, which uses the **X** and **G**-operators, the **last**-predicate, as well as disjunctions and atomic propositions. Toward this, we make use of the following semantic relations:

$$\neg\, \mathbf{X}\, \varphi = \mathbf{last} \vee \mathbf{X} \neg\varphi \quad ; \quad \neg\, \mathbf{F}\, \varphi = \mathbf{G} \neg\varphi \quad ; \quad \neg(\varphi_1 \wedge \varphi_2) = \neg\varphi_1 \vee \neg\varphi_2.$$

In particular, we use the fact that if $t \models \neg\varphi$ if and only if $t \not\models \varphi$. Using this, it is easy to see that if φ is separating for a sample $\mathcal{S} = (P, N)$, then $\neg\varphi$ is separating for the dual sample $\mathcal{S}' = (N, P)$. We leverage this idea by using the same algorithm (Algorithm 2) to produce formulas in the dual fragment. This is done in two steps: we swap the positive and the negative traces in the sample $\mathcal{S}$ and produce separating Directed LTL formulas for the dual sample $\mathcal{S}'$. Then, we negate the formula and store $\neg\varphi$ as a separating formula for $\mathcal{S}$.

3.5.2 Boolean Combinations of Formulas

As explained in the previous section, at each iteration (ℓ, w) we can efficiently generate the set $\{\varphi, (\varphi_P, \varphi_N)\}$ where, φ is a formula in the fragment of Directed LTL or its dual with length ℓ and width d, and $\varphi_P \subseteq P$ and $\varphi_N \subseteq N$ are the sets of traces in $\mathcal{S}$ that satisfy φ. We aim to form a boolean combination of these formulas to construct separating formulas, as illustrated in the introduction of this chapter (see Section 3.1). To solve this, we propose two algorithms: i. *Boolean set cover* and ii. *Learning decision trees*.

3.5.2.1 Boolean Set Cover Problem

Given a sample $\mathcal{S}$ and a set of formulas, the *Boolean set cover* problem asks, does there exist a Boolean combination of some of the formulas that is separating for $\mathcal{S}$? We first go through an example illustrated in Figure 3.1.

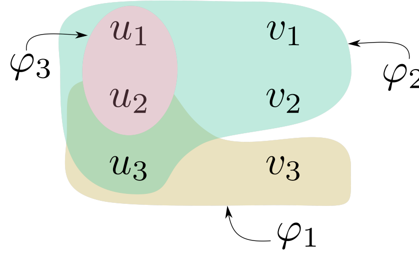


Figure 3.1: The Boolean set cover problem: the formulas φ_1, φ_2, and φ_3 satisfy the words encircled in the corresponding area; in this instance $(\varphi_1 \wedge \varphi_2) \vee \varphi_3$ is a separating formula.

In this example, we have $\mathcal{S} = (P = \{u_1, u_2, u_3\}, N = \{v_1, v_2, v_3\})$. We have three formulas φ_1, φ_2, and φ_3. Let the satisfaction set of a formula $SAT(\varphi)$ denotes the subset of $\mathcal{S}$ that satisfy the formula φ. The different coloured areas in the figure illustrate the satisfaction sets for each of the three formulas:

$$SAT(\varphi_1) = \{u_2, u_3, v_3\}; \qquad SAT(\varphi_2) = \{u_1, u_2, u_3, v_1, v_2\}; \qquad SAT(\varphi_3) = \{u_1, u_2\};$$

Inspecting the three subsets reveals that $(\varphi_1 \wedge \varphi_2) \vee \varphi_3$ is a separating formula. Note that, for any two formulas φ and ψ, $SAT(\varphi \wedge \psi) = SAT(\varphi) \cap SAT(\psi)$ and $SAT(\varphi \vee \psi) = SAT(\varphi) \cup SAT(\psi)$. A careful inspection will show that, $SAT\left((\varphi_1 \wedge \varphi_2) \vee \varphi_3\right) = \{u_1, u_2, u_3\}$ meaning that, it is separating for $\mathcal{S}$.

The Boolean set cover problem is a generalization of the well-known *set cover problem*, one of Karp's 21 NP-complete problems. Given $S_1, \ldots, S_m$ such that $S_i \subseteq [1, n]$ for all $1 \leq i \leq m$ and $\bigcup_i S_i = [1, n]$, the *set cover* problem is to identify the smallest subset $I \subseteq [1, m]$ such that, $\bigcup_{i \in I} S_i$ covers all of $[1, n]$ – such a set I is called a cover.The Boolean set cover problem reduces to the standard set cover problem if all formulas satisfy none of the negative traces: in that case, conjunctions are not useful, and we can ignore the negative traces.

The set cover problem is known to be NP-complete. However, there exists a polynomial-time $\log(n)$-approximation algorithm called the greedy algorithm: it is guaranteed to construct a cover that is at most $\log(n)$ times larger than the minimum cover. This approximation ratio is optimal in the following sense [48]: there is no polynomial time $(1 - o(1)) \log(n)$-approximation algorithm for subset cover unless P = NP. We describe a similar greedy approach to solve the Boolean set cover problem. Informally, the greedy algorithm for the subset cover problem does the following: it iteratively constructs a cover I by sequentially adding the most 'promising subset' to I, which is the subset S_i maximising how many more elements of $[1, n]$ are covered by adding i to I.

Algorithm 3 Greedy algorithm for the Boolean set cover problem

```
1: input: Sample S, and F a set of formulas
2: Fix a constant K ≤ |F|
3: procedure GREEDY(F)
4:     choose the K formulas φ_1, ..., φ_K in F with highest score
5:     for all ψ ∈ F do
6:         for all i ∈ [1, K] do
7:             construct ψ ∧ φ_i and ψ ∨ φ_i
8:             compute their scores
9:             if one of the two formulas is separating then
10:                return the separating formula
11:            end if
12:            let θ be the formula with highest score computed using ψ
13:            if θ has higher score than ψ then
14:                add θ to F
15:            end if
16:        end for
17:    end for
18: end procedure
```

Solving Boolean set cover. We introduce an extension of the greedy algorithm to the Boolean set cover problem. We first fix a score function to determine the 'promising formulas'. Intuitively a formula gets a higher score if it is close to being separating (accepts many positive traces and rejects many negative traces) and of a smaller size. Taking both these into account, we define the score function as follows:

$$Score(\varphi) = \frac{|\{t \in P \mid t \models \varphi\}| + |\{t \in N \mid t \not\models \varphi\}|}{\sqrt{|\varphi|} + 1}$$

The use of $\sqrt{\cdot}$ is empirical; it is used to mitigate the importance of size over being separating.

The algorithm is illustrated in Algorithm 3. The algorithm maintains a set of formulas F and adds new formulas to the set until finding a separating formula. At each iteration (ℓ, w) initially F contains the generated Directed LTL formulas of length ℓ and width w. Let us fix a constant K, and at each point in time, the algorithm chooses the K formulas $\varphi_1, \ldots, \varphi_K$ with the highest score in F and constructs all disjunctions and conjunctions of φ_i with formulas in F. For each i, we keep the disjunction or conjunction with a maximal score and add this formula to F if it has a higher score than φ_i. We repeat this procedure until we find a separating formula or no formula is added to F.

Practical Optimisations. For practical purposes, we impose the following heuristics in our implementation:

- Set the value of K to 5.
- Keep an upper bound on the size of a separating formula, which we use to cut off

computations that cannot lead to smaller formulas in the greedy algorithm for the Boolean subset cover problem.

3.5.2.2 Decision Tree

Another natural approach to solve the Boolean set cover problem is to use *decision trees*. Although decision trees produce LTLf formulas of bigger size, they can be helpful for practical purposes as they are easy to interpret (If-then-else formulas).

A decision tree for LTLf formulas is a structure that resembles a tree in which all nodes are tagged with LTLf formulas. The terminal nodes of the decision tree are labelled either $true$ or $false$, while the internal nodes are tagged with general LTLf formulas. Each node and its corresponding formula impose decisions made to classify a trace. Each node has two sub-trees as its children, with the left child connected by a solid edge and the right child connected by a dashed edge. An example of a decision tree representing an LTLf formula is depicted in Figure 3.2.

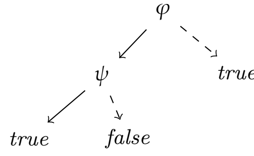


Figure 3.2: A decision tree for LTLf formula

A decision tree d tagged with LTLf formulas can be interpreted as an LTLf formula in the following way:

$$\varphi_d = \bigvee_{\rho \in \Pi} \bigwedge_{\varphi \in \rho} \varphi$$

where Π is the set of paths from the root node to a leaf node labelled $true$ and $\varphi \in \rho$ represents that it appears on path ρ (negated if it appeared on a dashed edge).

Learning decision trees. Intuitively, at each iteration, we take the generated Directed LTL formulas as input and try to learn a decision tree that represents their Boolean combinations encoding a separating formula for $\mathcal{S}$. Our decision tree learning algorithm follows the standard Top-Down Induction of Decision Trees (TDIDT) techniques [120]. These techniques construct the tree in a top-down fashion by finding appropriate features to partition the data and recursively proceed on the partitions.

We follow a similar technique with the set of generated Directed LTL formulas F. We assign a suitable Directed LTL formula φ from the set F for each node. After assigning the formula φ, next we split the sample $\mathcal{S}$ into two sub-samples $\mathcal{S}_1$ and $\mathcal{S}_2$ with respect to φ as follows: $\mathcal{S}_1 = \{t \in \mathcal{S} \mid t \models \varphi\}$, and $\mathcal{S}_2 = \{t \in \mathcal{S}_2 \mid t \not\models \varphi\}$. Then, we recursively apply the same technique to each resulting sub-samples. We terminate the algorithm when the sample $\mathcal{S}$ is perfectly classified and return the decision tree representing the separating formula. Otherwise, we reach a split which is not separable by any formulas from the set F, indicating that there is no separating Boolean combination. We proceed

to generate Directed LTL formulas at the next iteration. At each node, the aim is to choose suitable Directed LTL formulas for each node that will be of small size and also split the sample (sub-samples, respectively) efficiently. Towards this, we use the same score function similar to the one used in the Boolean set cover algorithm. Given a sample or sub-sample $\mathcal{S} = (P, N)$ and a formula φ, recall that the score function is defined as:

$$Score(\varphi, \mathcal{S}) = \frac{|\{t \in P \mid t \models \varphi\}| + |\{t \in N \mid t \not\models \varphi\}|}{\sqrt{|\varphi|} + 1},$$

Practical Optimisations. In practice, we do not intend for large LTLf formulas keeping interpretability at mind. Hence, we can fix an upper bound on the size of the decision tree learned at each iteration. This will make our search process reasonably faster.

We experimented with both approaches and found that the greedy algorithm is faster and yields smaller formulas, as DT-based methods produce formulas of a particular shape. This is why we do not report on the experimental results with this method.

3.5.3 Anytime Property

The anytime property of our algorithm is a consequence of storing the smallest formula seen so far (Line 7 and 14 of Algorithm 1). Once we find a separating formula, we can output it and continue the search for smaller separating formulas.

3.6 Extension to the Noisy Setting

Recall the LTLf noisy learning problem from Section 3.4. Given a threshold ε, we want to find an ε-separating LTLf formula.

Algorithm 1 can seamlessly be extended to the noisy setting by rewriting lines 6 and 13: instead of outputting only separating formulas, we output ε-separating formulas. Note that checking if a generated formula is ε-separating or not can be checked easily as we keep track of the set of traces from the sample that satisfy the formula.

3.7 Theoretical Guarantees

This section establishes the theoretical guarantees of our LTLf learning algorithm.

3.7.1 Correctness

The correctness of Algorithm 1 follows from lines 6 and 13. If the algorithm outputs a formula, it will be separating (ε-separating, respectively).

3.7.2 Termination

To show termination, we prove the following lemma:

Lemma 2 *For every informative sample $\mathcal{S}$, there exists an LTLf formula that is separating for $\mathcal{S}$.*

Proof of Lemma 2. Let $\mathcal{S} = (P, N)$. For every pair of traces $t_1 \in P$ and $t_2 \in N$, we create a Directed LTLf formula φ_{t_1,t_2} such that $t_1 \models \varphi_{t_1,t_2}$ and $t_2 \not\models \varphi_{t_1,t_2}$ that indicates the first symbol where t_1 and t_2 differ using a sequence of **X**-operators and an appropriate partial symbol. Let φ be defined as:

$$\varphi = \bigvee_{t_1 \in P} \bigwedge_{t_2 \in N} \varphi_{t_1,t_2}$$

Note that, as $\mathcal{S}$ is informative, φ is a valid formula, and it is separating for $\mathcal{S}$. □

Lemma 2 already gives an upper bound (the size of φ as described above) on the size of a separating formula that the algorithm searches for that guarantees the termination. It is easy to check that the size of the formula φ is $\mathcal{O}(t_l \cdot |\mathcal{S}|^2)$ where t_l is the maximum length of a trace appearing in the sample $\mathcal{S}$ and $|\mathcal{S}|$ denotes the total number of traces in $\mathcal{S}$. This also gives the intuition behind the NP upper bound for the LTL learning problem, as one can guess a formula of size $\leq |\varphi|$ (polynomial size) and check if it is separating or not in polynomial time. However, note that, for practical purposes, we fix a reasonable upper bound in search of a separating formula.

3.7.3 Completeness

To show the completeness of our algorithm, we show the expressiveness power of the Boolean combination of Directed LTL formulas. In particular, we prove the following theorem:

Theorem 3 *Every formula of LTLf $(\mathbf{F}, \mathbf{X}, \wedge, \vee)$ is equivalent to a Boolean combination of Directed LTL formulas. Equivalently, every formula of LTLf $(\mathbf{G}, \mathbf{X}, \wedge, \vee)$ is equivalent to a Boolean combination of formulas in the dual fragment of Directed LTL.*

To get an intuition, let us consider the formula $\mathbf{F}\, p \wedge \mathbf{F}\, q$, which is not a Directed LTL formula. However, it can be written as a Boolean combination of two Directed LTL formulas as follows:

$$\mathbf{F}\, p \wedge \mathbf{F}\, q = \mathbf{F}(p \wedge \mathbf{F}\, q) \vee \mathbf{F}(q \wedge \mathbf{F}\, p)$$

The second formulation has a disjunction over the possible orderings of p and q. It is worth noting that rewriting an LTLf formula as a Boolean combination of Directed LTL

formulas can result in an exponential size increase. This can be easily seen by generalising the previous example,

$$\bigwedge_{1\leq i\leq n} \mathbf{F}\, p_i = \bigvee_{\pi\in\Pi(n)} \mathbf{F}(p_{\pi(1)} \wedge \mathbf{F}(p_{\pi(2)} \wedge \mathbf{F}(\cdots \wedge \mathbf{F}\, p_{\pi(n)})))$$

where $\Pi(n)$ is the set of all permutations of the set $\{1, \ldots, n\}$.

The formal proof generalises this rewriting idea. For readability, in the following proofs, we refer to Directed LTL as dLTL and the Boolean combination of Directed LTL as dLTL$(\wedge, \vee)$.

We first prove a lemma necessary for the proof of the Theorem 3.

Lemma 3 *Let Δ_1, Δ_2 be two dLTL formulas. Then, $(\Delta_1 \wedge \Delta_2)$ can be written as a disjunction of formulas in dLTL.*

Proof of Lemma 3. To prove the lemma, we use induction over the structure of $\Delta_1 \wedge \Delta_2$ to show that it can be written as a disjunction of dLTL formulas. As induction hypothesis, we consider all formulas $\Delta_1' \wedge \Delta_2'$, where at least one of Δ_1' and Δ_2' is structurally smaller than Δ_1 and Δ_2 respectively, can be written as a disjunction of dLTL formulas.

The base case of the induction is when either Δ_1 or Δ_2 is a partial symbol. In this case, $\Delta_1 \wedge \Delta_2$ is itself a dLTL formula by definition of dLTL formulas.

The induction step proceeds via case analysis on the possible root operators of the formulas Δ_1 and Δ_2

- **(Case 1: either Δ_1 or Δ_2 is of the form $s \wedge \Delta$ for some partial symbol s.)** Without loss of generality, let us say $\Delta_1 = s \wedge \Delta$. In this case, $\Delta_1 \wedge \Delta_2 = (s \wedge \Delta) \wedge \Delta_2 = s \wedge (\Delta \wedge \Delta_2)$. By hypothesis, $\Delta \wedge \Delta_2 = \bigvee_i \Gamma_i$ for some Γ_i in dLTL. Thus, $\Delta_1 \wedge \Delta_2 = s \wedge \bigvee_i \Gamma_i = \bigvee_i (s \wedge \Gamma_i)$, which is a disjunction of dLTL formulas.
- **(Case 2: Δ_1 is of the form $\mathbf{X}\,\delta_1$ and Δ_2 is of the form $\mathbf{X}\,\delta_2$.)** In this case, $\Delta_1 \wedge \Delta_2 = \mathbf{X}(\delta_1 \wedge \delta_2)$. By hypothesis, $\delta_1 \wedge \delta_2 = \bigvee_i \gamma_i$ for some γ_i's in dLTL. Thus, $\Delta_1 \wedge \Delta_2 = \mathbf{X}(\bigvee_i \gamma_i) = \bigvee_i \mathbf{X}\,\gamma_i$, which is a disjunction of dLTL formulas.
- **(Case 3: Δ_1 is of the form $\mathbf{X}\,\delta_1$ and Δ_2 is of the form $\mathbf{F}\,\delta_2$.)** In this case, $\Delta_1 \wedge \Delta_2 = \mathbf{X}\,\delta_1 \wedge \mathbf{F}\,\delta_2 = (\mathbf{X}\,\delta 1 \wedge \delta_2) \vee (\mathbf{X}\,\delta_1 \wedge \mathbf{F}\,\mathbf{X}\,\delta_2) = (\mathbf{X}\,\delta 1 \wedge \delta_2) \vee \mathbf{X}(\delta_1 \wedge \mathbf{F}\,\delta_2)$. By hypothesis, both formulas $(\mathbf{X}\,\delta 1 \wedge \delta_2)$ and $(\delta_1 \wedge \mathbf{F}\,\delta_2)$ can be written as a disjunction of dLTL formulas. Thus, $\Delta_1 \wedge \Delta_2$ can also be written as a disjunction of dLTL formulas
- **(Case 4: Δ_1 is of the form $\mathbf{F}\,\delta_1$ and Δ_2 is of the form $\mathbf{F}\,\delta_2$.)** In this case, $\Delta_1 \wedge \Delta_2 = \mathbf{F}\,\delta_1 \wedge \mathbf{F}\,\delta_2 = \mathbf{F}(\delta_1 \wedge \mathbf{F}\,\delta_2) \vee \mathbf{F}(\delta_2 \wedge \mathbf{F}\,\delta_1)$. By hypothesis, both formulas $\delta 1 \wedge \mathbf{F}\,\delta_2$ and $\delta_2 \wedge \mathbf{F}\,\delta_1$ can be written as a disjunction of dLTL formulas. Thus, $\Delta_1 \wedge \Delta_2$ can also be written as a disjunction of dLTL formulas.

□

Now using Lemma 3, we will sketch the proof of Theorem 3.

Proof of Theorem 3. The proof proceeds via induction on the structure of formulas φ in LTLf $(\mathbf{F}, \mathbf{X}, \wedge, \vee)$. As induction hypothesis, we consider all formulas φ' structurally smaller than φ can be expressed in dLTL$(\wedge, \vee)$.

As the base case of the induction, we observe that formulas p for all $p \in \mathcal{P}$ are dLTL formulas and thus, in dLTL$(\wedge, \vee)$.

For the induction step, we perform a case analysis based on the root operator of φ.

- **(Case 1: $\varphi = \varphi_1 \vee \varphi_2$ or $\varphi = \varphi_1 \wedge \varphi_2$)**: By hypothesis, φ_1 is in dLTL$(\wedge, \vee)$ and φ_2 is in dLTL$(\wedge, \vee)$. Now, φ is in dLTL$(\wedge, \vee)$ since dLTL$(\wedge, \vee)$ is closed under positive boolean combinations.
- **(Case 2: $\varphi = \mathbf{X}\,\varphi_1$)**: By hypothesis, $\varphi_1 \in$ dLTL$(\wedge, \vee)$ and thus $\varphi_1 = \bigvee_j (\bigwedge_i \Delta_i)$. Now, $\varphi = \mathbf{X}(\bigvee_j (\bigwedge_i \Delta_i)) = \bigvee_j (\mathbf{X}(\bigwedge_i \Delta_i)) = \bigvee_j (\bigwedge_i \mathbf{X}\,\Delta_i) = \bigvee_i \bigwedge_i \Delta'_i$ ($\mathbf{X}\,\Delta_i$ is a dLTL formula). Thus, φ is in dLTL$(\wedge, \vee)$.
- **(Case 3: $\varphi = \mathbf{F}\,\varphi_1$)**: By hypothesis, $\varphi_1 \in$ dLTL$(\wedge, \vee)$ and thus $\varphi_1 = \bigvee_j (\bigwedge_i \Delta_i)$. Now $\varphi = \mathbf{F}\,\varphi_1 = \bigvee_j (\mathbf{F}(\bigwedge_i \Delta_i))$. Using lemma 3, we can re-write $\bigwedge_i \Delta_i$ as $\bigvee_i \Gamma_i$ for some Γ_i's in dLTL. As a result, $\varphi = \bigvee_j \bigvee_i \mathbf{F}(\Gamma_i)$. Thus, φ is in dLTL$(\wedge, \vee)$.

□

3.8 Experimental Evaluation

In this section, we answer the following research questions to assess the performance of our LTLf learning algorithm.

RQ1: How effective are we in learning concise LTLf formulas from samples?

RQ2: How much scalability do we achieve through our algorithm?

RQ3: What do we gain from the anytime property of our algorithm?

RQ4: How effective are we in the noisy learning setting?

3.8.1 Experimental Setup

To answer the questions above, we have implemented a prototype of our algorithm in Python 3 in a tool named SCARLET[1] (SCalable Anytime algoRithm for LEarning lTl). We run SCARLET on several benchmarks generated synthetically from LTLf formulas used in practice. To answer each research question precisely, we choose different sets of LTLf formulas. We discuss them in detail in the corresponding sections. Note that, however,

[1] https://github.com/rajarshi008/Scarlet

we did not consider any formulas with **U**-operator since SCARLET is not designed to find such formulas.

To assess the performance of SCARLET, we compare it against two state-of-the-art tools for learning logic formulas from examples:

1. FLIE[2], developed by [113], infers minimal LTLf formulas using a learning algorithm that is based on constraint solving (SAT solving).
2. SYSLITE[3], developed by [6], originally infers minimal past-time LTLf formulas using an enumerative algorithm implemented in a tool called CVC4SY [125]. For our comparisons, we use a version of SYSLITE that we modified (which we refer to as SYSLITE_L) to infer LTLf formulas rather than past-time LTLf formulas. Our modifications include changes to the syntactic constraints generated by SYSLITE_L as well as changing the semantics from past-time LTLf to ordinary LTL.

To obtain a fair comparison against SCARLET, in both the tools, we disabled the **U** operator. This is because allowing **U**-operator will only make the tools slower since they will have to search through all formulas containing **U**.

All the experiments are conducted on a single core of a Debian machine with an Intel Xeon E7-8857 CPU (at 3 GHz) using up to 6 GB of RAM. We set the timeout to 900 s for all experiments. We include scripts to reproduce all experimental results in a publicly available artifact [124].

Table 3.1: Common LTLf formulas used in practice

Absence:	$\mathbf{G}(\neg p)$, $\mathbf{G}(q \rightarrow \mathbf{G}(\neg p))$
Existence:	$\mathbf{F}(p)$, $\mathbf{G}(\neg p) \vee \mathbf{F}(p \wedge \mathbf{F}(q))$
Universality:	$\mathbf{G}(p)$, $\mathbf{G}(q \rightarrow \mathbf{G}(p))$
Disjunction of patterns:	$\mathbf{G}(\neg p) \vee \mathbf{F}(p \wedge \mathbf{F}(q) \vee \mathbf{G}(\neg s) \vee \mathbf{F}(r \wedge \mathbf{F}(s))$, $\mathbf{F}(r) \vee \mathbf{F}(p) \vee \mathbf{F}(q)$

3.8.1.1 Sample Generation

To provide a comparison among the learning tools, we follow the literature [113, 129] and use synthetic benchmarks generated from real-world LTLf formulas. For benchmark generation, earlier works rely on a fairly naive generation method. In this method, starting from a formula φ, a sample is generated by randomly drawing traces and categorising them into positive and negative examples depending on the satisfaction with respect to φ. This method, however, often results in samples that can be separated by formulas much smaller than φ. Moreover, it often requires a prohibitively large amount of time to generate samples (for instance, for $\mathbf{G}\, p$, where almost all traces satisfy a formula) and often does not terminate in a reasonable time.

[2] https://github.com/ivan-gavran/samples2LTL
[3] https://github.com/CLC-UIowa/SySLite

To alleviate the issues in the existing method, we have designed a novel generation method for the quick generation of large samples. The outline of the generation algorithm is presented in Algorithm 4. The crux of the algorithm is to convert the LTLf formula φ into its equivalent DFA $\mathcal{A}_\varphi$ and then extract random traces from the DFA to obtain a sample of desired length and size.

To convert φ into its equivalent DFA $\mathcal{A}_\varphi$ (Line 3), we rely on a python tool LTLf2DFA[4]. Essentially, this tool converts φ into its equivalent formula in FOL (first-order logic over finite linear order sequences) [63] and then obtains a minimal DFA from the formula using a tool named MONA [76].

For extracting random traces from the DFA (Line 5 and 9), we use a procedure suggested by [19]. The procedure involves generating words by choosing letters that are more likely to lead to an accepting state. This requires assigning appropriate probabilities to the transitions of the DFA. In this step, we add our modifications to the procedure. The main idea is that once a path has been explored to generate a trace, we reduce the probability of each transition appearing in the path by a small fraction $0 \leq \delta \leq 1$ and distribute δ proportionately to other transitions. This increases the likelihood of obtaining distinct traces at each iteration following different paths of the DFA.

Algorithm 4 Sample generation algorithm

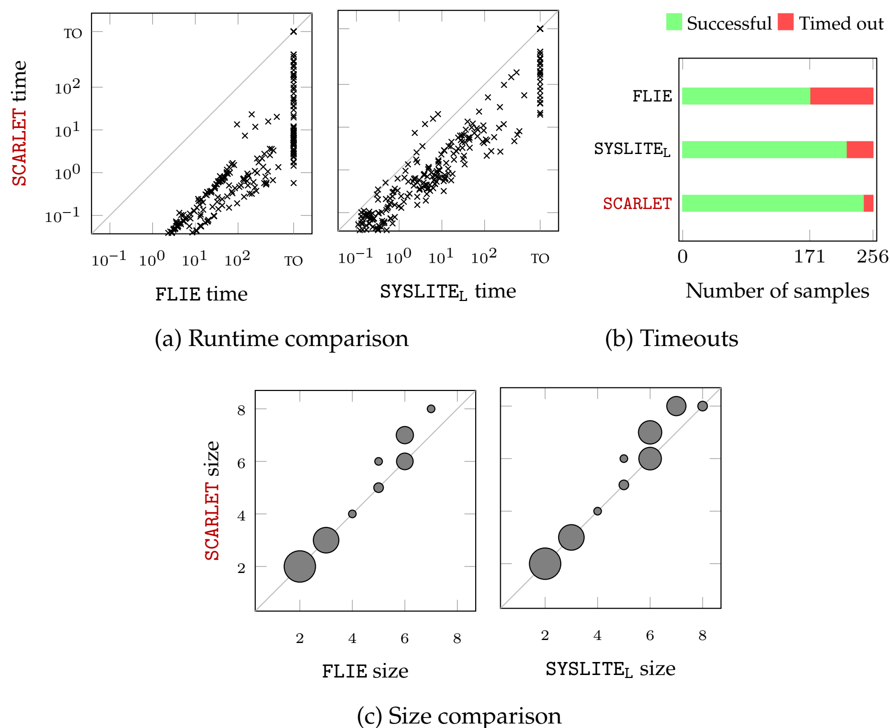
Input: Formula φ, length l, number of positive traces n_P, number of negative traces n_N.
1:
2: $P \leftarrow \{\}, N \leftarrow \{\}$
3: $\mathcal{A}_\varphi \leftarrow$ convert2DFA(φ)
4: **Loop** n_P times
5: $\quad w \leftarrow$ random accepted word of length l from $\mathcal{A}_\varphi$.
6: $\quad P \leftarrow P \cup \{w\}$
7: **end**
8: **Loop** n_N times
9: $\quad w \leftarrow$ random accepted word of length l from $\mathcal{A}^c_\varphi$.
10: $\quad N \leftarrow N \cup \{w\}$
11: **end**
12: **return** $S = (P, N)$

Unlike existing sample generation methods, our method does not create random traces and tries to classify them as positive or negative. This results in faster generation of large samples of better-quality (towards 'characteristic') samples in that the likelihood of getting φ as the smallest separating formula from the sample becomes higher.

3.8.2 RQ1: Performance Comparison

To address our first research question, we have compared all three tools on a synthetic benchmark suite generated from eight LTLf formulas. These formulas originate from a study by Dwyer et al. [51], who have collected a comprehensive set of LTLf formulas arising in real-world applications (see Table 3.1 for an excerpt). The selected LTLf

[4]https://github.com/whitemech/ltlf2DFA



(a) Runtime comparison (b) Timeouts

(c) Size comparison

Figure 3.3: Comparison of SCARLET, FLIE and $\texttt{SYSLITE}_\texttt{L}$ on synthetic benchmarks. In Figure 3.3a, all times are in seconds and 'TO' indicates timeouts. The size of the bubbles in the Figure 3.3c indicates the number of samples resulting into formulas of a particular size represented by each data point.

formulas have, in fact, also been used by FLIE for generating its benchmarks. While FLIE also considered formulas with **U**-operator, we did not consider them for generating our benchmarks due to reasons mentioned in the experimental setup.

Our benchmark suite consists of a total of 256 samples (32 for each of the eight LTLf formulas) generated using our generation method. The number of traces in the samples ranges from 50 to 2 000, while the length of traces ranges from 8 to 15.

Figure 3.3a presents the runtime comparison of FLIE, $\texttt{SYSLITE}_\texttt{L}$ and SCARLET on all 256 samples. The scatter plots show that SCARLET ran faster than FLIE on all samples. Likewise, SCARLET was faster than $\texttt{SYSLITE}_\texttt{L}$ on all but eight (out of 256) samples. SCARLET timed out on only 13 samples, while FLIE and $\texttt{SYSLITE}_\texttt{L}$ timed out on 85 and 36, respectively (see Figure 3.3b).

The good performance of SCARLET can be attributed to its efficient formula search technique. In particular, SCARLET only considers formulas that have a high potential of being a separating formula since it extracts Directed LTL formulas from the sample itself. FLIE and $\texttt{SYSLITE}_\texttt{L}$, on the other hand, search through arbitrary formulas (in order of increasing size), each time checking if the current one separates the sample.

Figure 3.3c presents the comparison of the size of the formulas inferred by each tool. On

170 out of the 256 samples, all tools terminated and returned an LTLf formula with size at most 7. In 150 out of these 170 samples, SCARLET, FLIE, and $\texttt{SYSLITE}_\text{L}$ inferred formulas of equal size, while in the remaining 20 samples, SCARLET inferred larger formulas. The latter observation indicates that SCARLET misses certain small, separating formulas, particularly those that are not a Boolean combination of directed formulas.

However, it is important to highlight that the formulas learned by SCARLET are, in most cases, not significantly larger than those learned by FLIE and $\texttt{SYSLITE}_\text{L}$. This can be seen from the fact that the average size of formulas inferred by SCARLET (on benchmarks in which none of the tools timed out) is 3.21, while the average size of formulas inferred by FLIE and $\texttt{SYSLITE}_\text{L}$ is 3.07.

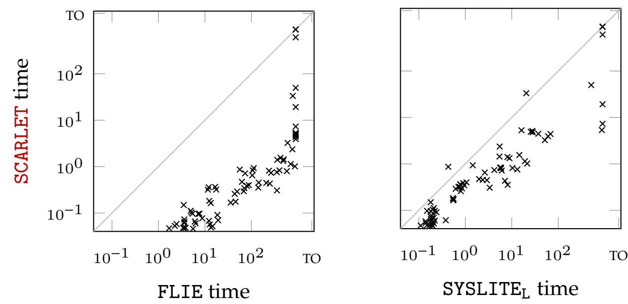
To ensure that SCARLET performs well, not only in our generated benchmarks, we compared the performance of the tools on an existing benchmark suite[5] [60]. The benchmark suite has been generated using a somewhat naive generation method from the same set of LTLf formulas listed in Table 3.1.

Figure 3.4a represents the runtime comparison of FLIE, $\texttt{SYSLITE}_\text{L}$ and SCARLET on 98 samples. From the scatter plots, we observe that SCARLET runs much faster than FLIE on all samples and than $\texttt{SYSLITE}_\text{L}$ on all but two samples. Also, SCARLET timed out only on 3 samples while $\texttt{SYSLITE}_\text{L}$ timed out on 6 samples and FLIE timed out on 15 samples.

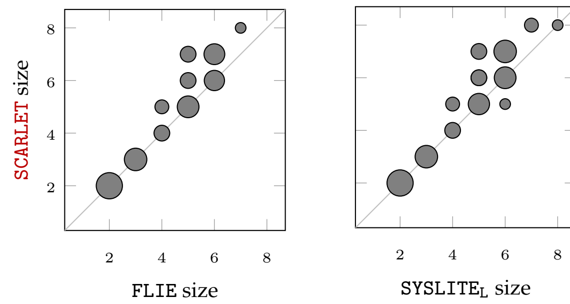
Figure 3.4b presents the comparison of formula size inferred by each tool. On 84 out of 98 samples, where none of the tools timed out, we observe that on 65 samples, SCARLET inferred formula size equal to the one inferred by $\texttt{SYSLITE}_\text{L}$ and FLIE. Further, the size gap is insignificant in the samples where SCARLET learns larger formulas than other tools. This is evident from the fact that the average formula size learned by SCARLET is 4.13, which is slightly higher than that by FLIE and $\texttt{SYSLITE}_\text{L}$, 3.84.

Overall, SCARLET displayed significant speed-up over both FLIE and $\texttt{SYSLITE}_\text{L}$ while learning a formula similar in size, answering question RQ1 in the positive.

[5] https://github.com/cryhot/samples2LTL

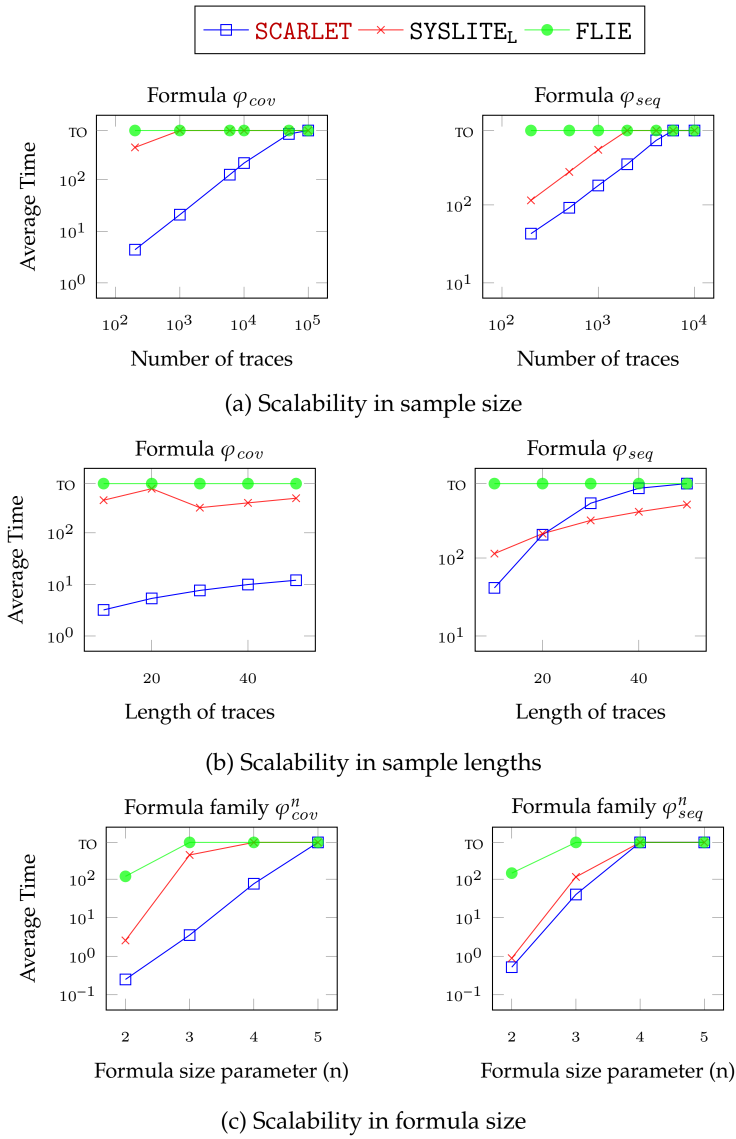


(a) Runtime comparison



(b) Size comparison

Figure 3.4: Comparison of SCARLET, FLIE and SYSLITE$_L$ on existing benchmarks. In Figure 3.4a, all times are in seconds and 'TO' indicates timeouts. The size of the bubbles indicates the number of samples for each data point.



(a) Scalability in sample size

(b) Scalability in sample lengths

(c) Scalability in formula size

Figure 3.5: Comparison of SCARLET, FLIE and SYSLITE$_L$ on synthetic benchmarks. In Figures 3.5a and 3.5b, all times are in seconds and 'TO' indicates timeouts.

3.8.3 RQ2: Scalability

To address the second research question, we investigate the scalability of SCARLET in two dimensions: the size of the sample and the size of the formula from which the samples are generated.

Scalability with respect to the size of the samples. For demonstrating the scalability with respect to the size of the samples, we consider two formulas $\varphi_{cov} = \mathbf{F}(a_1) \wedge \mathbf{F}(a_2) \wedge \mathbf{F}(a_3)$ and $\varphi_{seq} = \mathbf{F}(a_1 \wedge \mathbf{F}(a_2 \wedge \mathbf{F}\, a_3))$, both of which appear commonly in robotic motion planning [56]. While the formula φ_{cov} describes the property that a robot eventually visits (or covers) three regions a_1, a_2, and a_3 in arbitrary order, the formula φ_{seq} describes that the robot has to visit the regions in the specific order $a_1 a_2 a_3$.

We have generated two sets of benchmarks for both formulas, for which we varied the number of traces and their length, respectively. More precisely, the first benchmark set contains 90 samples of an increasing number of traces (5 samples for each number), ranging from 200 to 100 000, each consisting of traces of fixed length 10. On the other hand, the second benchmark set contains 90 samples of 200 traces, containing traces from length 10 to length 50.

Figure 3.5a shows the average runtime results of SCARLET, FLIE, and $\texttt{SYSLITE}_\text{L}$ on the first benchmark set. We observe that SCARLET substantially outperformed the other two tools on all samples. This is because both φ_{cov} and φ_{seq} are of size eight, and inferring formulas of such size is computationally challenging for FLIE and $\texttt{SYSLITE}_\text{L}$. In particular, FLIE and $\texttt{SYSLITE}_\text{L}$ need to search through all formulas of size up to eight to infer the formulas, while SCARLET, due to its efficient search order (using the length and width of a formula), infers them faster.

From Figure 3.5a, we further observe a significant difference between the run times of SCARLET on samples generated from formula φ_{cov} and from formula φ_{seq}. This is evident from the fact that SCARLET failed to infer formulas for samples of φ_{seq} starting at a size of 6 000, while it could infer formulas for samples of φ_{cov} up to a size of 50 000. Such a result is again due to the search order used by SCARLET: while φ_{cov} is a Boolean combination of directed formulas of length 1 and width 1, φ_{seq} is a directed formula of length 3 and width 1.

Figure 3.5b depicts the results we obtained by running all the second benchmark set with varying trace lengths. Some trends we observe here are similar to the ones we observe in the first benchmark set. For instance, SCARLET performs better on the samples from φ_{cov} than it does on samples from φ_{seq}. The reason for this remains similar: it is easier to find a formula which is a boolean combination of length 1, width 1 dLTL, than a simple LTLf of length 3 and width 1.

Contrary to the results on the first benchmark set, we observe that the increase of runtime with the length of the sample is quadratic. This explains why on samples from φ_{seq} on large lengths such as 50, SCARLET faces time-out. However, for samples from φ_{cov}, SCARLET displays the ability to scale way beyond length 50.

Scalability with respect to the size of the formula. To demonstrate the scalability with respect to the size of the formula used to generate samples, we have extended φ_{cov} and φ_{seq} to families of formulas $(\varphi^n_{cov})_{n\in\mathbb{N}\setminus\{0\}}$ with $\varphi^n_{cov} = \mathbf{F}(a_1) \wedge \mathbf{F}(a_2) \wedge \ldots \wedge \mathbf{F}(a_n)$ and $(\varphi^n_{seq})_{n\in\mathbb{N}\setminus\{0\}}$ with $\varphi^n_{seq} = \mathbf{F}(a_1 \wedge \mathbf{F}(a_2 \wedge \mathbf{F}(\ldots \wedge \mathbf{F}\, a_n)))$, respectively. This family of formulas describe properties similar to that of φ_{cov} and φ_{seq}, but the number of regions is parameterized by $n \in \mathbb{N} \setminus \{0\}$. We consider formulas from the two families by varying n from 2 to 5 to generate a benchmark suite consisting of samples (5 samples for each formula) having 200 traces of length 10.

Figure 3.5c shows the average run time comparison of the tools for samples from increasing formula sizes. We observe a trend similar to Figure 3.5a: SCARLET performs better than the other two tools and infers formulas of the family φ^n_{cov} faster than that of φ^n_{seq}. However, contrary to the nearly linear increase of the runtime with the number of traces, we notice an almost exponential increase of the runtime with the formula size.

Overall, our experiments show better scalability with respect to sample and formula size compared to the other tools, answering RQ2 in the positive.

3.8.4 RQ3: Anytime Property

To answer RQ3, we list two advantages of the anytime property of our algorithm. We demonstrate these advantages by showing evidence from the runs of SCARLET on benchmarks used in RQ1 and RQ2.

First, in the instance of a time out, our algorithm may find a “concise” separating formula while the other tools will not. In our experiments, we observed that for all benchmarks used in RQ1 and RQ2, SCARLET obtained a formula even when it timed out. In fact, in the samples from φ^5_{cov} used in RQ2, SCARLET (see Figure 3.5c) obtained the exact original formula, that too within one second (0.7 seconds in average), although timed out later. The time out was because SCARLET continued to search for smaller formulas even after finding the original formula.

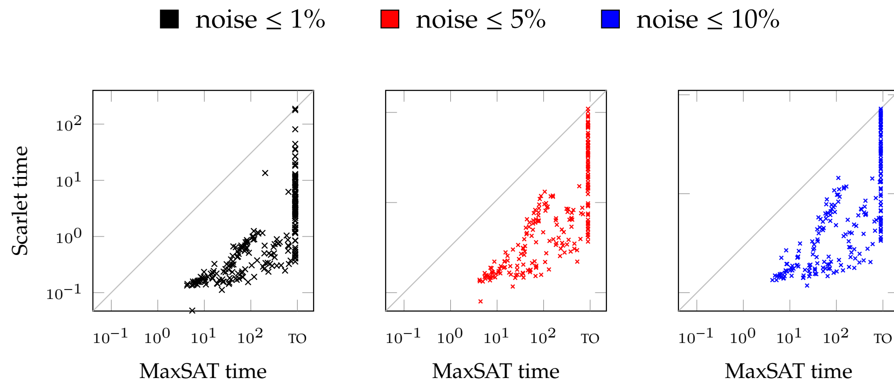
Second, our algorithm can actually output the final formula earlier than its termination. This is evident from the fact that, for the 243 samples in RQ1 where SCARLET does not time out, the average time required to find the final formula is 10.8 seconds, while the average termination time is 25.17 seconds. Thus, there is a chance that even if one stops the algorithm earlier than its termination, one can still obtain the final formula.

Our observations from the experiments indicate the advantages of anytime property to obtain a concise separating formula and, thus, answering RQ3 in the positive.

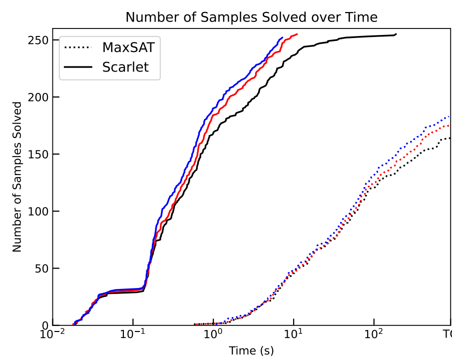
3.8.5 RQ4: Noisy Setting

To answer RQ4, we compared the noisy learning setting of SCARLET against a state-of-the-art tool [60] that uses MaxSAT based approach to infer LTL from noisy data. For this comparison, we ran both tools on the same synthetic benchmark suite introduced to answer RQ1 consisting of 256 samples. We tested for three noise thresholds (1%, 5%, and 10%) and presented the runtime comparison in Figure 3.6a.

This comparison clearly indicates the efficiency and better performance of our algorithm against the MaxSAT-based approach. Figure 3.6b represents a cactus plot that shows the cumulative number of samples from which both tools inferred LTL formulas over time for the same three noise levels. This clearly validates the scalability of our algorithm in the noisy setting.



(a) Runtime comparison



(b) Cactus plot

Figure 3.6: Comparison of SCARLET and MaxSAT on synthetic noisy benchmarks for different noise thresholds. All times are in seconds, and 'TO' indicates timeouts. The scatter plots in Figure 3.6a represent the comparison of time taken to solve each sample by both tools for three different noise levels. The cactus plot in Figure 3.6b represents the total number of samples solved within a given time-point by both the tools, and steeper lines represent better performance.

3.9 Conclusion

This chapter introduces a novel algorithm for learning LTLf formulas from observed system behaviours. We tackled the scalability problem of the existing algorithms in the literature by opting for an approximation algorithm. In addition to that, our algorithm is also, anytime, a useful property for practical purposes. We have also developed the

tool SCARLET to implement our algorithm and validate our claims by running suitable experiments.

Our algorithm targets a strict fragment of LTL, restricting its expressivity in two aspects: it does not include the **U**ntil operator, and we cannot nest the **F**inally and the **G**lobally operators. Incorporating the **U**-operator directly in our algorithm will not aid the scalability of the same as to keep track of 'promising formulas' having the **U**-operator, one may have to keep track of subsets of positions (at least exponentially larger) in a trace that satisfies each generated formula and our efficient dynamic programming tables in its current form cannot bypass that.

For nesting of **F** and **G** operators, first note that $\mathbf{F}\,\mathbf{G}\,\varphi$ and $\mathbf{G}\,\mathbf{F}\,\varphi$ can be rewritten as:

$$\mathbf{F}\,\mathbf{G}\,\varphi = \mathbf{G}\,\mathbf{F}\,\varphi = \mathbf{F}(\neg\,\mathbf{last} \vee \varphi)$$

However, arbitrary nesting of **F** and **G** can still be helpful, e.g. in $\mathbf{F}(a \wedge \mathbf{X}\,\mathbf{G}\,b)$ and it is not known if we can bound the number of nesting of **F** and **G** in LTLf. Esparza et al. in [55] shows that the nesting of **F** and **G** can be bounded by 2 for the general form of LTL, although it needs to be clarified to work. Therefore, we leave extending our algorithm to full LTL as an interesting future work.

Another interesting direction to explore will be to explore the characterization of DFAs that are equivalent to Directed LTL. In [63], the authors showed that full LTLf is equivalent to star-free regular languages and hence to counter-free automata. It will be interesting to see if the Directed LTL fragment results in any restricted class of these automata with nice algorithmic properties. In particular, these characterizations are helpful while considering the *reactive synthesis problems*.

One more important open question concerns the theoretical guarantees offered by the greedy algorithm for the Boolean set cover problem. It extends a well-known algorithm for the classic set cover problem, and this restriction has been proved to yield an optimal $\log(n)$-approximation. Do we have similar guarantees in our more general setting?

Chapter 4

Learning formulas in MTL

The previous chapter discusses learning concise specifications in LTL from observed system behaviour. In this chapter, we focus on learning specifications in Metric Temporal Logic (MTL) that extend LTL to describe the real-time properties of a system. We present an algorithm for learning MTL designed specifically to aid runtime verification. The content presented in this chapter is based on the work in [123].

4.1 Background

Runtime verification is a well-established method for ensuring the correctness of cyber-physical systems during runtime. Techniques in runtime verification are known to be more rigorous than conventional testing while not being as resource intensive as exhaustive formal verification [42]. In the field of runtime verification, among other techniques, monitoring system executions against formal specifications during runtime is a widely used one. Over the years, numerous monitoring techniques have been proposed for a variety of specification languages [75, 50, 47, 13].

In this work, we focus on Metric Temporal Logic (MTL) as the specification language. It is popularly employed for monitoring cyber-physical systems [77, 105]. Similar to LTL, MTL specifications are often easy to interpret due to their resemblance to natural language and, thus, also find applications in Artificial Intelligence [142]. Recall that, while there are many possible semantics of MTL (e.g., discrete, dense-time, pointwise, etc. [116]), we employ the dense-time continuous semantics as it is more natural and general than the counterparts [15, 9].

Virtually all verification techniques for MTL rely on the availability of a formal specification. However, as manually writing specifications is a tedious and error-prone task, different techniques to automatically learn concise specifications that precisely express the design requirements have been deployed. As we mentioned before, most of the existing works have targeted specification languages such as Linear Temporal Logic (LTL) [34, 113, 122] and Signal Temporal Logic (STL) [7, 102, 111, 133], with few works for MTL [78, 142]. In the context of explainability, these techniques mostly focus on the conciseness of the specification.

However, conciseness is not the only measure of interest for specifications, especially in the context of online monitoring. In online monitoring, specifically in *stream-based* runtime monitoring, a monitor reads an execution as a stream of data and verifies if a given specification is invariant (i.e., holds at all time points) in the execution. Many stream-based monitors [68, 82, 100] support MTL formulas. Typically, such monitors produce a stream of (Boolean) verdicts with some "latency", which depends on the lookahead of the formula. The lookahead required for an MTL formula is often formalized as its *future-reach* [77, 79], which is the amount of time required to determine its satisfaction at any time point.

With the aim of reducing the latency for efficient online monitoring, we focus on automatically learning MTL specifications based on two regularizers, size and future-reach. To this end, similar to the LTL learning setting in the previous chapter, we rely on a sample $\mathcal{S}$ of system executions observed for a finite duration, which is partitioned into a set P of positive executions and a set N of negative executions. We now formulate the central problem of the work as follows: given a sample $\mathcal{S} = (P, N)$ and a future-reach bound k, synthesize a minimal size MTL formula φ that (i) is *globally separating* for $\mathcal{S}$, in that, φ holds at all time points in the positive executions and does not hold at some time point in the negative executions[1], and (ii) the future-reach of φ is smaller than k.

Interestingly, without a future-reach bound, the most concise MTL formula that can be synthesized can have a large future-reach value, increasing the latency required for online monitoring. To illustrate this, we follow the running example Example 1 from Chapter 1 and observe some simulations of an autonomous vehicle in the following example.

Example 3 *During the simulations of an autonomous vehicle, we sample executions (shown below) of the vehicle every second for six seconds. We classify them as positive (denoted using u_i's) or negative (denoted using v_i's) based on whether the vehicle encountered a collision or not.*

	0	1	2	3	4	5
u_1:	$\{p,q\}$	$\{p\}$	$\{q\}$	$\{p,q\}$	$\{p\}$	$\{p\}$
u_2:	$\{q\}$	$\{\}$	$\{q\}$	$\{p\}$	$\{p\}$	$\{p,q\}$
v_1:	$\{p\}$	$\{q\}$	$\{\}$	$\{\}$	$\{\}$	$\{\}$
v_2:	$\{p\}$	$\{p,q\}$	$\{p\}$	$\{\}$	$\{p\}$	$\{\}$

In the executions, we use p to denote that there is no preceding vehicle within a particular unsafe distance ahead of the vehicle and q to denote that the vehicle's brake is triggered. Also, if p occurs at time point t, we interpret it as p holding during the entire interval $[t, t+1)$ to maintain the continuity of the executions.

In the sample, a minimal globally separating formula is $\varphi_1 = \mathbf{F}_{[0,3]}\, q$. This indicates that in all positive executions, the brake is triggered every three seconds (i.e., within the interval $[t, t+3]$ for every time point t), irrespective of whether there is an obstacle within the unsafe distance. The formula φ_1 has size two and a future-reach of three seconds, meaning that any online monitor requires a three second lookahead window to check the satisfaction of φ_1. There is another formula $\varphi_2 = p \vee \mathbf{F}_{[0,1]}\, q$ that is globally separating for the sample. This indicates that in all positive executions, for every time point t, if a preceding vehicle is within the unsafe distance,

[1]note that, this notion is different from the notion of "separating formula" in the previous chapter

then the brake is triggered within one second (i.e., within the interval $[t, t+1]$). Although of size four, φ_2 has future-reach of one second and will be typically preferred over φ_1 for monitoring in a safety-critical scenario.

To synthesize MTL formulas, we rely on a reduction to constraint satisfaction problems. In particular, following other works in formula inference [113, 129], our algorithm encodes the problem in a series of satisfiability problems in Linear Real Arithmetic (LRA). Further, we analyze the complexity of the decision version of the problem. While the exact complexity lower bounds are open, we show that the corresponding decision problem is in NP. We also show that the problem is already NP-hard for a fragment of MTL where we restrict the set of operators appearing in the formulas.

We implement our algorithm using a popular SMT solver in a tool called `TEAL`. We evaluate `TEAL` to synthesize MTL formulas in the application of a cyber-physical system. We also empirically study the interplay between the size and future-reach of a formula.

4.1.1 Related Works

To our knowledge, there are only a limited number of works for synthesizing MTL formulas. One of them [142] infers MTL formulas as decision trees for representing task knowledge in Reinforcement Learning. Some other works [78, 143] consider the parameter search problem for MTL where, given a parametric MTL formula (i.e., an MTL formula with missing temporal bounds), they infer the ranges of parameters where the formula holds/does not hold on a given system. Unlike our work, none of these works aims at synthesizing concise MTL specifications for monitoring tasks.

There are, nevertheless, numerous runtime monitoring procedures for MTL [136, 9, 49, 77, 16, 35, 83, 100], clearly indicating the need for efficiently monitorable MTL specifications. Many of them also rely on future-reach of a specification [77, 16] or other similar measures (e.g., horizon [49], worst-case propagation delay [83], etc.) to quantify the efficiency of their monitoring procedure.

Interestingly, several works focus on synthesizing formulas in STL, an extension of MTL to reason about real-valued signals. Bartocci et al. [14] provide a comprehensive survey of the existing works on inferring STL. Many of them [7, 89, 88] solve the parameter search for STL, while others [25, 24] learn decision trees over STL formulas, which typically do not result in concise formulas. There are few works [111, 114] that do prioritize the conciseness of formulas during inference. These works cannot be directly applied to solve our problem for two main reasons. First, these works assume inputs to be *piecewise-affine continuous* signals. While the above assumption is natural for synthesizing STL formulas inference from real-valued signals, in our setting, we must rely on the assumption that our inputs are *piecewise-constant* signals, which is natural for Boolean-valued signals. Second, these works do not employ any measure, apart from conciseness, that directly influences the efficiency of runtime monitoring. To the best of our knowledge, there is no work that uses an SMT-solver based approach for MTL. Our approach contains encoding interval operations in Linear Real Arithmetic which is partly similar to the approach in [22] where the authors use similar encoding of interval operations to solve a different problem.

Finally, there are works on synthesizing formulas in other temporal logic such as Linear Temporal Logic (LTL) [113, 126, 34, 122], Property Specification Language (PSL) [129], etc. which are not easily extensible to our setting.

4.2 Outline of the Chapter

We introduce the fragment of MTL we consider along with the learning problem we are interested in Section 4.3. We elaborate on our algorithm in Section 4.4. We prove all the theoretical results related to the algorithms in Section 4.5. Section 4.6 contains all the experimental results and evolution of our algorithms. We discuss possible adaptations and subtleties regarding our algorithm in Section 4.7. Finally, we present an extension of our algorithm that can capture STL specifications in Section 4.8.

4.3 The Problem Formulation

In this work, we aim to infer MTL specifications from finite system observations. In [116], the authors describe the two commonly adapted *dense-time* semantics of MTL on infinite executions: continuous and pointwise. In this study, we follow the continuous framework and extend it to finite observations inspired by [77]. Before introducing the syntax and semantics of MTL, we first define the concepts of *signal* and *observed prefixes* representing continuous behaviour of the system executions. In this section, we introduce the basic notations used throughout the paper.

4.3.1 Signals and Prefixes.

Recall that we represent continuous system executions as signals. A *signal* $\vec{x} : \mathbb{R}_{\geq 0} \to 2^{\mathcal{P}}$ over a set of propositions $\mathcal{P}$ is an infinite time series that describes relevant system events over time. A prefix of a signal $\vec{x}$ up to a time point T is a function $\vec{x}_T : [0, T) \to 2^{\mathcal{P}}$ such that $\vec{x}_T(t) = \vec{x}(t)$ for all $t \in [0, T)$.

To synthesize MTL formulas, we rely on finite observations that are sequences of the form $\Omega = \langle (t_i, \delta_i) \rangle_{i \leq n_{\vec{x}}}$, $n_{\vec{x}} \in \mathbb{N}$ such that,

- $t_0 = 0$,
- $t_{n_{\vec{x}}} < T$, and
- for all $i \leq n_{\vec{x}}$, $\delta_i \subseteq \mathcal{P}$ is the set of propositions that hold at time-point t_i.

To construct well-defined signal prefixes, we approximate each observation Ω as a *piecewise-constant* signal prefix $\vec{x}_T^{\Omega}$ using interpolation as:

– for all $i < n_{\vec{x}}$, for all $t \in [t_i, t_{i+1})$, $\vec{x}_T(t) = \delta_i$;

– for all $t \in [t_{n_{\vec{x}}}, T)$, $\vec{x}_T(t) = \delta_{n_{\vec{x}}}$.

For readability, we refer to signal prefixes simply as 'prefixes' when clear from the context. Now, we introduce the semantics and syntax of MTL that we opt for in this chapter.

4.3.2 Metric Temporal Logic

MTL is a logic formalism for specifying the real-time properties of a system. Recall the syntax of MTL from Section 2.2.3.2. For this work, we do not consider the timed-Until ($\mathbf{U}_I$) operator, as in the context of inferring temporal logic specifications, the operator is often hard to interpret [88, 122, 142]. While our algorithm is tailored to include the $\mathbf{U}_I$ operator, there are some technical subtleties that also affect the efficiency of the algorithm. For convenience, we mention the syntax of MTL we opt for in this chapter:

$$\varphi := p \in \mathcal{P} \mid \neg p \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \mathbf{F}_I\, \varphi \mid \mathbf{G}_I\, \varphi$$

where $p \in \mathcal{P}$ is a proposition, $\neg$ is the negation operator, $\wedge$ and $\vee$ are the conjunction and disjunction operators respectively, and $\mathbf{F}_I$ and $\mathbf{G}_I$ are the timed-Finally and timed-Globally operators respectively. Here, I is a closed interval of non-negative real numbers of the form $[a, b]$ where $0 \leq a \leq b$[2]. Note that the syntax is presented in *negation normal form*, which means that the $\neg$ operator can only appear in front of a proposition.

Recall that, as a syntactic representation of an MTL formula, we rely on *syntax-DAGs* and define the size $|\varphi|$ of an MTL formula φ as the number of nodes in its syntax-DAG, e.g., the size of $(p \wedge \mathbf{G}_I\, q) \vee (\mathbf{F}_I\, p)$ is six as its syntax-DAG has six nodes.

As mentioned above, we follow the continuous semantics of MTL formulas. For the standard continuous semantics ($\models$) of MTL over infinite signals, we refer to Section 2.2.3.2. In our context, we require the semantics of MTL to be defined over finite prefixes of signals such that the resulting formulas are "useful" for monitoring the signals over infinite sequences. We seek an optimistic semantics denoted by $\models_{\mathrm{f}}$ for an MTL formula φ over a prefix $\vec{x}_T$, such that $\vec{x}_T \models_{\mathrm{f}} \varphi$ holds if there exists an infinite signal extending $\vec{x}_T$ that satisfies φ. In other words, $\vec{x}_T$ carries no evidence against φ. To formalize this, we require the definition of $\models_{\mathrm{f}}$ to satisfy the following lemma.

Lemma 4 *Given a prefix $\vec{x}_T$, let $ext(\vec{x}_T) = \{\vec{x} \mid \vec{x}_T$ is a prefix of $\vec{x}\}$ be the set of all infinite extensions of $\vec{x}_T$. Then given an MTL formula φ, $\vec{x}_T \models_{\mathrm{f}} \varphi$ if there exists $\vec{x} \in ext(\vec{x}_T)$ such that $\vec{x} \models \varphi$.*

Towards this, we follow the idea of 'weak semantics' of MTL defined in [77]. In this work, following Eisner et al. [53], the authors have defined the weak semantics of MTL for the pointwise setting, which we adapt here for the continuous setting. Given a prefix $\vec{x}_T$, we inductively define when an MTL formula φ holds at time point $t < T$, i.e., $(\vec{x}_T, t) \models_{\mathrm{f}} \varphi$, as follows:

[2]Since we infer MTL formulas with bounded lookahead, we restrict I to be bounded.

$$
\begin{array}{lll}
(\vec{x}_T, t) \models_{\mathrm{f}} p & \iff & p \in \vec{x}_T(t); \\
(\vec{x}_T, t) \models_{\mathrm{f}} \neg p & \iff & p \notin \vec{x}_T(t); \\
(\vec{x}_T, t) \models_{\mathrm{f}} \varphi_1 \wedge \varphi_2 & \iff & (\vec{x}_T, t) \models_{\mathrm{f}} \varphi_1 \text{ and } (\vec{x}_T, t) \models_{\mathrm{f}} \varphi_2; \\
(\vec{x}_T, t) \models_{\mathrm{f}} \varphi_1 \vee \varphi_2 & \iff & (\vec{x}_T, t) \models_{\mathrm{f}} \varphi_1 \text{ or } (\vec{x}_T, t) \models_{\mathrm{f}} \varphi_2; \\
(\vec{x}_T, t) \models_{\mathrm{f}} \mathbf{F}_{[a,b]}\, \varphi & \iff & t + b \geq T \text{ or } \exists t' \in [t+a, t+b] \text{ s.t. } (\vec{x}_T, t') \models_{\mathrm{f}} \varphi; \\
(\vec{x}_T, t) \models_{\mathrm{f}} \mathbf{G}_{[a,b]}\, \varphi & \iff & t + a \geq T \text{ or} \\
 & & \begin{cases} \forall t' \in [t+a, t+b], (\vec{x}_T, t') \models_{\mathrm{f}} \varphi, & \text{if } t + b < T \\ \forall t' \in [t+a, T), (\vec{x}_T, t') \models_{\mathrm{f}} \varphi, & \text{otherwise.} \end{cases}
\end{array}
$$

We say that $\vec{x}_T$ satisfies φ if $(\vec{x}_T, 0) \models_{\mathrm{f}} \varphi$. Now we prove Lemma 4 that justifies the choice of our semantics.

Proof of Lemma 4. We need to prove that, given a prefix $\vec{x}_T$ and an MTL formula φ, $\vec{x}_T \models_f \varphi$ if there exists an extension $\vec{x} \in ext(\vec{x}_T)$ such that, $\vec{x} \models \varphi$. We prove this by induction on the structure of the MTL formula φ. Fix a prefix $\vec{x}_T$ and an MTL formula φ. In fact, we prove a stronger statement by induction:

For all $t \in [0, T)$, $(\vec{x}_T, t) \models_f \varphi$ if there exists a signal $\vec{x} \in ext(\vec{x}_T)$ such that $(\vec{x}, t) \models \varphi$.

For the base case, let $\varphi = p \in \mathcal{P}$. Then, for all $t \in [0, T)$, if there exists $\vec{x} \in ext(\vec{x}_T)$ such that, $(\vec{x}, t) \models p$, then $(\vec{x}_T, t) \models_f p$. It is easy to check that this extends easily for the $\neg$ operator and the boolean connectives $\wedge$ and $\vee$. Now let, $\varphi = \mathbf{F}_{[a,b]}\, \psi$ and fix a time point $t \in [0, T)$. We have to prove if there exists a signal $\vec{x} \in ext(\vec{x}_T)$ such that, $(\vec{x}, t) \models \mathbf{F}_{[a,b]}\, \psi$, then $(\vec{x}_T, t) \models_f \mathbf{F}_{[a,b]}\, \psi$. Now by definition of $\models$, $\exists t' \in [t+a, t+b]$ such that, $(\vec{x}, t') \models \psi$. Now there are two cases: (i) $t + b \geq T$: in this case, $(\vec{x}_T, t) \models_f \mathbf{F}_{[a,b]}\, \psi$, and (ii) $t + b < T$: then, $t' < T$ and by induction hypothesis, $(\vec{x}_T, t') \models_f \psi$ as $(\vec{x}, t) \models \psi$. Hence, $(\vec{x}_T, t) \models_f \varphi$. The case for $\varphi = \mathbf{G}_{[a,b]}\, \psi$ can be proved similarly. □

4.3.3 The Problem Formulation

Next, we formally introduce the various aspects of the central problem of the paper.

Sample. The input data consists of a set of labelled (piecewise-constant) prefixes. Formally, we rely on a sample $\mathcal{S} = (P, N)$ consisting of a set P of positive prefixes and a set N of negative prefixes. We assume that the sample is *informative*, i.e., $P \cap N = \emptyset$, indicating that a prefix cannot be both positive and negative. We say an MTL formula φ is *globally separating* (**G**-sep, for short) for $\mathcal{S}$ if it satisfies all the positive prefixes at each time-point and does not satisfy negative prefixes at some time point. We choose this notion as most stream-based monitors check if the specification holds at every time-point, and hence the outermost **G** in the desired specification is implicit [17]. Formally, given a sample $\mathcal{S}$, we define an MTL formula φ to be **G**-sep for $\mathcal{S}$ if (i) for all $\vec{x}_T \in P$ and for all $t \in [0, T)$, $(\vec{x}_T, t) \models_{\mathrm{f}} \varphi$; and (ii) for all $\vec{y}_T \in N$, there exists $t \in [0, T)$ such that $(\vec{y}_T, t) \not\models_{\mathrm{f}} \varphi$.

Future-Reach. Previously, we have explained how the 'lookahead' of an MTL formula has an impact on the efficiency of an online monitor monitoring the formula. To formalize the lookahead of an MTL formula φ, we rely on its future-reach $fr(\varphi)$, following [79, 77], which indicates how much of the future is required to determine the satisfaction of φ. It is defined inductively as follows:

$$fr(p) = fr(\neg p) = 0;$$
$$fr(\varphi_1 \wedge \varphi_2) = fr(\varphi_1 \vee \varphi_2) = \max(fr(\varphi_1), fr(\varphi_2));$$
$$fr(\mathbf{F}_I\,\varphi) = fr(\mathbf{G}_I\,\varphi) = \sup(I) + fr(\varphi).$$

To highlight that $fr(\varphi)$ quantifies the lookahead of φ, we observe the following lemma:

Lemma 5 *Let φ be an MTL formula such that $fr(\varphi) = k$. Then, $\vec{x} \models \varphi$ if and only if $\forall \vec{y} \in ext(\vec{x}_k)$, $\vec{y} \models \varphi$.*

Intuitively, the above lemma states that a formula with future-reach k cannot distinguish between two signals that are identical up to time k.

Problem Setting. We now formally introduce the problem of synthesizing an MTL formula to aid online monitoring. In the problem, we ensure that the MTL formula is efficient for monitoring by allowing the system designer to specify a future-reach bound.

Problem 3 (SynTL) *Given a sample $\mathcal{S} = (P, N)$ and a future-reach bound k, find an MTL formula φ such that*

- *φ is* $\mathbf{G}$-sep *for $\mathcal{S}$;*
- *$fr(\varphi) \leq k$;*
- *for every MTL formula φ' such that φ' is* $\mathbf{G}$-sep *for $\mathcal{S}$ and $fr(\varphi') \leq k$, $|\varphi| \leq |\varphi'|$*

Intuitively, the above optimization problem asks to synthesize a minimal size MTL formula that is $\mathbf{G}$-sep for $\mathcal{S}$ and has future-reach within the input bound. Note that this problem might not have a solution. For example, let $T > 1$ and the sample $\mathcal{S}$ consists of one positive prefix: $\vec{x} := (0, \{p\}), (1, \{q\})$ and one negative prefix: $\vec{y} := (0, \{q\})$. One can check that there does not exist any MTL formula, which is $\mathbf{G}$-sep for $\mathcal{S}$. This is because if any formula φ is $\mathbf{G}$-sep, then $(\vec{x}, 1) \models \varphi$ and $\forall t, (\vec{y}, t) \not\models \varphi$. But, any suffix of $\vec{y}$ and $\vec{x}[1:]$ (suffix of $\vec{x}$ starting from time point 1) is the same, and that leads to a contradiction.

Hence, to ensure that our algorithm terminates, we impose a reasonable size bound B. We search for a solution up to size B. Note that this is natural for our setting as we look for concise specifications and do not want to search for formulas beyond a certain size. Also, note that, in this problem, our goal is to find only one MTL formula, although there might be multiple solutions.

4.4 An SMT-based Algorithm

Our algorithm is based on a Satisfiability Modulo Theories (SMT)-based approach, which draws inspiration from the constraint satisfaction-based approaches that have been developed for synthesizing temporal logic formulas in the past [113, 34, 129, 6]. In essence, our algorithm generates a sequence of formulas in Linear Real Arithmetic (LRA) and then uses a highly optimized SMT solver to search for the desired solution. To provide a detailed description of our algorithm, we first introduce LRA to our readers in order to provide them with the necessary background knowledge.

4.4.1 Linear Real Arithmetic (LRA)

In LRA [12], given a set of real variables $\mathcal{Y}$, a *term*(t) is defined recursively by the following grammar:

$$t := c \mid y \in \mathcal{Y} \mid c \cdot t \mid t_1 \circ t_2$$

where $c \in \mathbb{R}$ is a constant, $y \in \mathcal{Y}$ is a real variable, $\cdot$ is the multiplication function that is only allowed with a constant, and $t_1 \circ t_2$ is a function application such that, $\circ \in \{+, -\}$ and t_1, t_2 are two terms. An *atomic formula* is of the form $t_1 \diamond t_2$ where $\diamond \in \{<, \leq, =, \geq, >\}$. Then, an *LRA formula*, defined recursively, is either an atomic formula, the negation $\neg\Phi$ of an LRA formula Φ, or the disjunction $\Phi_1 \vee \Phi_2$ of two formulas Φ_1, Φ_2. We additionally include standard Boolean constants true, and false and Boolean operators $\wedge$, $\rightarrow$ and $\leftrightarrow$.

To assign meaning to an LRA formula, we rely on a so-called *interpretation* function $\iota \colon \mathcal{Y} \rightarrow \mathbb{R}$ that maps real variables to constants in $\mathbb{R}$. An interpretation ι can easily be lifted to a term t in the usual way and is denoted as $\iota(t)$. Now, we define when ι *satisfies* a formula φ, denoted by $\iota \models \varphi$, recursively as follows:

$$\begin{aligned} \iota \models t_1 \diamond t_2 \text{ for } \diamond \in \{<, \leq, =, \geq, >\} &\iff \iota(t_1) \diamond \iota(t_2) \text{ is } \mathit{true} \\ \iota \models \neg\Phi &\iff \iota \not\models \Phi \\ \iota \models \Phi_1 \vee \Phi_2 &\iff \iota \models \Phi_1 \text{ or } \iota \models \Phi_2 \end{aligned}$$

We say that an LRA formula Φ is *satisfiable* if there exists an interpretation ι with $\iota \models \Phi$.

Despite being NP-complete, with the rise of the SAT/SMT revolution [107], checking the satisfiability of LRA formulas can be handled effectively by several highly-optimized SMT solvers [45, 39, 10].

4.4.2 Algorithm Overview

Our algorithm generates a sequence of Linear Real Arithmetic (LRA) formulas, denoted as $\langle \Phi^n_{\mathcal{S},k} \rangle_{n=1,2,\ldots}$, in order to facilitate the search for a suitable MTL formula. The formula $\Phi^n_{\mathcal{S},k}$ has the following properties:

1. $\Phi^n_{\mathcal{S},k}$ is satisfiable if and only if there exists an MTL formula φ of size n such that φ is $\mathbf{G}$-sep for $\mathcal{S}$ and $fr(\varphi) \leq k$.

Algorithm 5 Overview of our algorithm

Input: Sample $\mathcal{S}$, fr-bound k, Size bound B
1: **for** $n \in \{1, \dots, B\}$ **do**
2: Construct $\Phi^n_{\mathcal{S},k} := \Phi^{str}_{n,\mathcal{S},k} \wedge \Phi^{fr}_{n,\mathcal{S},k} \wedge \Phi^{sem}_{n,\mathcal{S},k}$
3: **if** $\Phi^n_{\mathcal{S},k}$ is SAT with a satisfying interpretation ι **then**,
4: Construct φ_ι from ι
5: **break**
6: **end if**
7: **end for**

2. from any satisfying interpretation ι of $\Phi^n_{\mathcal{S},k}$, one can construct an appropriate MTL formula φ_ι that is **G**-sep for $\mathcal{S}$.

We provide an outline of our algorithm in Algorithm 5. We check the satisfiability of the constructed LRA formula $\Phi^n_{\mathcal{S},k}$ for increasing values of the parameter n, starting from 1 and proceeding up to the size bound B. If the formula $\Phi^n_{\mathcal{S},k}$ is found to be satisfiable for some $n \in 1, \dots, B$, our algorithm constructs a **G**-sep MTL formula φ_ι from a satisfying interpretation ι returned by the SMT solver. On the other hand, if the formula $\Phi^n_{\mathcal{S},k}$ is not satisfiable for any $n \leq B$, our algorithm reports that no suitable **G**-sep MTL formula can be found within the given size bound of B.

The crux of our algorithm is the construction of the formula $\Phi^n_{\mathcal{S},k}$. Internally, $\Phi^n_{\mathcal{S},k} := \Phi^{str}_{n,\mathcal{S},k} \wedge \Phi^{fr}_{n,\mathcal{S},k} \wedge \Phi^{sem}_{n,\mathcal{S},k}$ is a conjunction of three subformulas, each with a distinct role. The subformula $\Phi^{str}_{n,\mathcal{S},k}$ encodes the structure of the prospective MTL formula. The subformula $\Phi^{fr}_{n,\mathcal{S},k}$ ensures that the future-reach of the prospective formula is less than or equal to k. Finally, the subformula $\Phi^{sem}_{n,\mathcal{S},k}$ ensures that the prospective formula is **G**-sep for $\mathcal{S}$. In what follows, we expand on the construction of each of the introduced subformula. We drop the subscripts n, $\mathcal{S}$, and k from the subformulas when they are clear from the context.

Structural Constraints. Following Neider and Gavran [113], we symbolically encode the syntax-DAG of the prospective MTL formula using the formula Φ^{str}. For this, we first fix a naming convention for the nodes of the syntax-DAG of an MTL formula. For a formula of size n, we assign to each of its nodes an identifier from $\{1, \dots, n\}$ such that the identifier of each node is larger than that of its children if it has any. Note that such a naming convention may not be unique. Based on these identifiers, we denote the subformula of φ rooted at Node i as $\varphi[i]$. In that case, $\varphi[n]$ is precisely the formula φ.

Next, to encode a syntax-DAG symbolically, we introduce the following variables[3]:

(i) Boolean variables $x_{i,\lambda}$ for $i \in \{1, \dots, n\}$ and $\lambda \in \mathcal{P} \cup \{\neg, \vee, \wedge, \mathbf{F}, \mathbf{G}\}$;

(ii) Boolean variables $l_{i,j}$ and $r_{i,j}$ for $i \in \{1, \dots, n\}$ and $j \in \{1, \dots, i\}$;

[3]We include Boolean variables in our LRA formulas since Boolean variables can always be simulated using real variables that are constrained to be either 0 or 1.

(iii) real variables a_i and b_i for $i \in \{1, \dots, n\}$.

The variable $x_{i,\lambda}$ tracks the operator labelled in Node i, meaning, $x_{i,\lambda}$ is set to true if and only if Node i is labelled with λ. The variable $l_{i,j}$ (resp., $r_{i,j}$) tracks the left (resp., right) child of Node i, meaning, $l_{i,j}$ (resp., $r_{i,j}$) is set to true if and only if the left (resp., right) child of Node i is Node j. Finally, the variable a_i (resp., b_i) tracks the lower (resp., upper) bound of the interval I of a temporal operator (i.e., operators $\mathbf{F}_I$ and $\mathbf{G}_I$), meaning that, if a_i (resp. b_i) is set to $a \in \mathbb{R}$ (resp., $b \in \mathbb{R}$), then the lower (resp., upper) bound of the interval of the operator in Node i is a (resp., b). While we introduce variables a_i and b_i for each node, they become relevant only for the nodes that are labelled with a temporal operator.

We now impose structural constraints on the introduced variables to ensure they encode valid MTL formulas. First, we define the following two formulas:

$$\texttt{one}-\texttt{left}(i) = \Big[\bigwedge_{1\leq i\leq n}\bigvee_{1\leq j\leq i} l_{i,j}\Big] \land \Big[\bigwedge_{1\leq i\leq n}\bigwedge_{1\leq j\leq j'\leq n} \neg l_{i,j} \lor \neg l_{i,j'}\Big], \text{and}$$

$$\texttt{one}-\texttt{right}(i) = \Big[\bigwedge_{1\leq i\leq n}\bigvee_{1\leq j\leq i} r_{i,j}\Big] \land \Big[\bigwedge_{1\leq i\leq n}\bigwedge_{1\leq j\leq j'\leq n} \neg r_{i,j} \lor \neg r_{i,j'}\Big]$$

that encodes that Node i contains exactly one left child (exactly one right child, respectively).

Now let $\Lambda = \mathcal{P} \cup U_\Lambda \cup B_\Lambda$, where U_Λ denotes the set of unary operators and B_Λ denotes the set of binary operators. Then the encoding of the structural constraints contains the following:

$$\Phi^{str} := \Big[\bigwedge_{1\leq i\leq n}\bigvee_{\lambda\in\Lambda} x_{i,\lambda}\Big] \land \Big[\bigwedge_{1\leq i\leq n}\bigwedge_{\lambda\neq\lambda'\in\Lambda} \neg x_{i,\lambda} \lor \neg x_{i,\lambda'}\Big] \land \tag{4.1}$$

$$\bigwedge_{1\leq i\leq n}\Big(\bigvee_{p\in\mathcal{P}} x_{i,p} \rightarrow \Big[\bigwedge_{1\leq j\leq n} \neg l_{i,j} \land \bigwedge_{1\leq j\leq n} \neg r_{i,j'}\Big]\Big) \land \tag{4.2}$$

$$\bigwedge_{1\leq i\leq n}\Big(\bigvee_{\lambda\in U_\Lambda} x_{i,\lambda} \rightarrow \Big[\texttt{one}-\texttt{left}(i) \land \bigwedge_{1\leq j\leq n} \neg r_{i,j'}\Big]\Big) \land \tag{4.3}$$

$$\bigwedge_{1\leq i\leq n}\Big(\bigvee_{\lambda\in B_\Lambda} x_{i,\lambda} \rightarrow \Big[\texttt{one}-\texttt{left}(i) \land \texttt{one}-\texttt{right}(i)\Big]\Big) \tag{4.4}$$

$$\bigwedge_{\substack{1\leq i\leq n\\ 1\leq j<i}}\Big(\big[x_{i,\neg} \land l_{i,j}\big] \rightarrow \bigvee_{p\in\mathcal{P}} x_{j,p}\Big) \tag{4.5}$$

Constraint 4.1 encodes the fact that each node only contains one operator or one proposition. Constraint 4.2 imposes that the nodes containing a proposition do not have

any child. Constraint 4.3 says that the nodes containing a unary operator contain exactly one child, while constraint 4.4 enforces that the nodes containing a binary operator contain exactly one left and exactly one right child. Finally, Constraint 4.5 imposes that the $\neg$ operator can occur only in front of propositions.

Let us consider a satisfying interpretation ι of Φ^{str}. From this interpretation, we can construct a syntax-DAG in a straightforward manner. Each node i in the DAG is labelled with a unique label λ, such that $\iota(x_{i,\lambda}) = 1$. The node n is designated as the root of the DAG. The arrangement of nodes in the DAG is determined uniquely by the values of $\iota(l_{i,j})$ and $\iota(r_{i,j})$. Additionally, we can derive an MTL formula from this syntax DAG, denoted as Φ^{str}_ι. It is important to note that at this stage, Φ^{str}_ι is not yet associated with the sample $\mathcal{S}$ or the future-reach bound k. To make Φ^{str}_ι follow the future-reach bound and be globally separating for the sample, we impose the next constraints as follows.

Future-reach Constraints. To symbolically compute the future-reach of the prospective formula φ, we encode the inductive definition of the future-reach, as described in Section 4.3.3 in an LRA formula. To this end, we introduce real variables f_i for $i \in \{1, \dots, n\}$ to encode the future-reach of the subformula $\varphi[i]$. Precisely, f_i is set to $f \in \mathbb{R}$ if and only if $fr(\varphi[i]) = f$.

To ensure the desired meaning of the f_i variables, we impose constraints as follows:

$$\bigwedge_{1 \le i \le n} x_{i,p} \rightarrow \big[f_i = 0\big] \wedge$$
$$\bigwedge_{\substack{1 \le i \le n \\ 1 \le j < i}} \big(x_{i,\neg} \wedge l_{i,j}\big) \rightarrow \big[f_i = f_j\big] \wedge$$
$$\bigwedge_{\substack{1 \le i \le n \\ 1 \le j, j' < i}} \big((x_{i,\vee} \vee x_{i,\wedge}) \wedge l_{i,j} \wedge r_{i,j'}\big) \rightarrow \big[f_i = \max(f_j, f'_j)\big] \wedge$$
$$\bigwedge_{\substack{1 \le i \le n \\ 1 \le j < i}} \big(x_{i,\mathbf{F}} \wedge l_{i,j}\big) \rightarrow \big[f_i = f_j + b_i\big] \wedge$$
$$\bigwedge_{\substack{1 \le i \le n \\ 1 \le j < i}} \big(x_{i,\mathbf{G}} \wedge l_{i,j}\big) \rightarrow \big[f_i = f_j + b_i\big] \wedge$$

These constraints express that the future-reach value at every node i, i.e., the future-reach of $\varphi[i]$, is calculated properly from the future-reach value of its children.

Finally, to enforce that the future-reach of the prospective MTL formula is within k, along with the constraints mentioned above, we have $f_n \le k$ in Φ^{fr}.

Note that any satisfying interpretation ι of $\Phi^{str} \wedge \Phi^{fr}$ results in a unique MTL formula whose future-reach is within the specified bound. Next, we have to introduce some constraints that make this formula globally separating.

Semantic Constraints. In order to assess whether the prospective formula is $\mathbf{G}$-sep, we need to encode the procedure of checking the satisfaction of an MTL formula into an LRA formula. For this purpose, we rely on a popular (offline) monitoring procedure introduced by Maler and Nickovic [105]. However, since our specific setting differs slightly, we will briefly outline our adaptation of the monitoring algorithm.

Given an MTL formula φ and a signal prefix $\vec{x}_T$, our monitoring algorithm computes a lexicographically ordered set $\mathcal{I}_\varphi(\vec{x}_T) = \{I_1, \ldots, I_\eta\}$ of *maximal disjoint time intervals* $I_1, \ldots, I_\eta$, where φ holds true on $\vec{x}_T$. In mathematical terms, the set $\mathcal{I}_\varphi(\vec{x}_T)$ we construct satisfies the following property:

Lemma 6 *Given an MTL formula φ and a prefix $\vec{x}_T$, for all $t \in [0, T)$, $(\vec{x}_T, t) \models_{\mathrm{f}} \varphi$ if and only if $t \in I$ for some $I \in \mathcal{I}_\varphi(\vec{x}_T)$.*

In our monitoring algorithm, we recursively compute the set $\mathcal{I}_\varphi(\vec{x}_T)$ based on the structure of the formula φ. To describe this recursive process, we utilize the notation $\mathcal{I}^\cup_\varphi(\vec{x}_T) = \bigcup_{I \in \mathcal{I}_\varphi(\vec{x}_T)} I$ to represent the union of intervals in $\mathcal{I}_\varphi(\vec{x}_T)$.

For the base case, we calculate $\mathcal{I}_p(\vec{x}_T)$ for each $p \in \mathcal{P}$ by aggregating the time points $t \in [0, T)$ where $(\vec{x}_T, t) \models_{\mathrm{f}} p$ into a collection of maximal disjoint time intervals. Recall that these intervals are of the form $[t_1, t_2)$ as the input prefixes are piecewise-constant.

In the inductive step, we make use of the relationships presented in Equation 4.5 for the various MTL operators. In the table, the notation $\ominus[a, b]$ is the Minkowski minus defined as, $[t_1, t_2) \ominus [a, b] = [t_1 - b, t_2 - a)$, and $\mathcal{I}^c = [0, T) - \mathcal{I}$ denotes the complement of the interval set $\mathcal{I}$.

$$
\begin{aligned}
\mathcal{I}^\cup_{\neg p}(\vec{x}_T) &= \left(\mathcal{I}^\cup_p(\vec{x}_T)\right)^c \\
\mathcal{I}^\cup_{\varphi_1 \vee \varphi_2}(\vec{x}_T) &= \mathcal{I}^\cup_{\varphi_1}(\vec{x}_T) \cup \mathcal{I}^\cup_{\varphi_2}(\vec{x}_T) \\
\mathcal{I}^\cup_{\varphi_1 \wedge \varphi_2}(\vec{x}_T) &= \mathcal{I}^\cup_{\varphi_1}(\vec{x}_T) \cap \mathcal{I}^\cup_{\varphi_2}(\vec{x}_T) \\
\mathcal{I}^\cup_{\mathbf{F}_{[a,b]}\varphi}(\vec{x}_T) &= \Big(\bigcup_{I \in \mathcal{I}_\varphi(\vec{x}_T)} I \ominus [a, b]\Big) \cup [T - b, T) \\
\mathcal{I}^\cup_{\mathbf{G}_{[a,b]}\varphi}(\vec{x}_T) &= \Big(\bigcup_{I \in (\mathcal{I}_\varphi(\vec{x}_T))^c} I \ominus [a, b]\Big)^c \cup [T - a, T)
\end{aligned}
$$

Equation 4.5: The relations for inductive computation of $\mathcal{I}^\cup_\varphi(\vec{x}_T)$

While Equation 4.5 presents the computation of $\mathcal{I}^\cup_\varphi(\vec{x}_T)$, we can obtain $\mathcal{I}_\varphi(\vec{x}_T)$ by simply partitioning $\mathcal{I}^\cup_\varphi(\vec{x}_T)$ into maximal disjoint intervals. Also note that these operations maintain the intervals to be of the form $[t_1, t_2)$ and thus are consistent with our formulation. Now we prove the correctness of Lemma 6 to establish the correctness of the computations of $\mathcal{I}^\cup_\varphi(\vec{x}_T)$.

Proof of Lemma 6. We prove that, for all $t \in [0, T)$, $(\vec{x}_T, t) \models_{\mathrm{f}} \varphi$ if and only if $t \in I$ for some $I \in \mathcal{I}_\varphi(\vec{x}_T)$. We prove both directions together by induction on the structure of the formula φ.

For the base case, one can check that for all $t \in [0, T)$, $t \in \mathcal{I}_p(\vec{x}_T)$ if and only if $t \in I$ for some $I \in \mathcal{I}_\varphi(\vec{x}_T)$ by construction. The proof for the neg operator and the boolean connectives $\wedge$ and $\vee$ follow from the correctness of the construction in the work of [105]. Here, we provide the proofs for two timed-temporal operators as their semantics differ from the work in [105].

Let $\varphi = \mathbf{F}_{[a,b]}\,\psi$. To show the forward direction, let $t \in I$ for some $I \in \mathcal{I}_\varphi(\vec{x}_T)$. We have to prove that, $(\vec{x}_T, t) \models_f \mathbf{F}_{[a,b]}\,\psi$. In particular, $t \in \mathcal{I}^\cup_\varphi(\vec{x}_T)$ by definition, i.e., $t \in \big(\bigcup_{I \in \mathcal{I}_\psi(\vec{x}_T)} I \ominus [a,b]\big) \cup [T-b, T)$. There are two cases: (i) $t \in [T-b, T)$: in this case, $t + b \geq T$ and by definition of $\models_f$, $(\vec{x}_T, t) \models_f \varphi$, or (ii) $t \in \big(\bigcup_{I \in \mathcal{I}_\psi(\vec{x}_T)} I \ominus [a,b]\big)$: Fix the interval $I' = [t_1, t_2) \in \mathcal{I}_\psi(\vec{x}_T)$ such that, $t \in (I' \ominus [a,b])$. By induction hypothesis, for all $t' \in I'$, $(\vec{x}_T, t') \models_f \psi$. Now, $t < t_2 - a \implies t + a < t_2$ and $t \geq t_1 - b \implies t + b \geq t_1$. Hence, $I' = [t_1, t_2) \supset [t+a, t+b]$. Hence, $\exists t' \in [t+a, t+b]$ such that, $(\vec{x}_T, t') \models_f \psi$ and henceforth, $(\vec{x}_T, t) \models_f \varphi$.

For the backward direction, we assume that, $(\vec{x}_T, t) \models_f \mathbf{F}_{[a,b]}\,\psi$ and prove that, $t \in I$ for some $I \in \mathcal{I}_\varphi(\vec{x}_T)$. In particular, we show that $t \in \mathcal{I}^\cup_\varphi(\vec{x}_T) = \big(\bigcup_{I \in \mathcal{I}_\psi(\vec{x}_T)} I \ominus [a,b]\big) \cup [T-b, T)$ and the rest of the argument follows from the fact that, $\mathcal{I}_\varphi(\vec{x}_T)$ is obtained by taking the maximal disjoint intervals of $\mathcal{I}^\cup_\varphi(\vec{x}_T)$. Now, by definition of $\models_f$, there are two possibilities: (i) $t + b \geq T$: then, $t \in [T-b, T)$ and hence, $t \in \mathcal{I}^\cup_\varphi(\vec{x}_T)$, or (ii) $\exists t' \in [t+a, t+b]$ such that, $(\vec{x}_T, t') \models_f \psi$. Now, by induction hypothesis, $t' \in I$ for some $I \in \mathcal{I}_\psi(\vec{x}_T)$. Let $I = [t_1, t_2)$. Now, $t_2 - a > t' - a \geq t$ and $t_1 - b \leq t' - b \leq t$. This implies that, $t \in [t_1 - b, t_2 - a) = (I \ominus [a,b])$ which proves that, $t \in \mathcal{I}^\cup_\varphi(\vec{x}_T)$.

The proof for the operator $\mathbf{G}_I$ can be derived similarly. □

We demonstrate the above computation with the running example.

Example 4 *For an illustration, we compute $\mathcal{I}_{\varphi_2}(u_1)$ from Example 3, where $u_1 = (0, \{p, q\}), (1, \{p\}), (2, \{q\}), (3, \{p, q\}), (4, \{p\}), (5, \{p\})$ is the first positive prefix, $\varphi_2 = p \vee \mathbf{F}_{[0,1]}\,q$, and $T = 6$. First, we have $\mathcal{I}_p(u_1) = \{[0,2), [3,6)\}$ and $\mathcal{I}_q(u_1) = \{[0,1), [2,4)\}$. Now, we can compute $\mathcal{I}_{\mathbf{F}_{[0,1]}\,q}(u_1) = \{[0,4), [5,6)\}$ and then $\mathcal{I}_{p \vee \mathbf{F}_{[0,1]}\,q}(u_1) = \{[0,6)\}$. We illustrate this in Figure 4.1.*

In the above computation, the number of maximal intervals required in $\mathcal{I}_\varphi(\vec{x}_T)$ is upper-bounded by $\mathcal{M} = n \cdot m$, where $n = |\varphi|$ and $m = \max(\{|\mathcal{I}_p(\vec{x}_T)| \mid p \in \mathcal{P}\})$, as also observed by Maler and Nickovic [105]. The computation of this bound can also be done inductively on the structure of the formula.

Now, in the subformula Φ^{sem}, we symbolically encode the set $\mathcal{I}_\varphi(\vec{x}_T)$ of our prospective MTL formula φ. In order to accomplish this, we introduce variables $t^l_{i,m,s}$ and $t^r_{i,m,s}$, where $i \in 1, \ldots, n$, $m \in 1, \ldots, \mathcal{M}$, and $s \in 1, \ldots, |\mathcal{S}|$, with s denoting the identifier for the sth prefix $\vec{x}^s_T$ in $\mathcal{S}$. The variables $t^l_{i,m,s}$ and $t^r_{i,m,s}$ encode the mth interval of $\mathcal{I}_{\varphi[i]}(\vec{x}^s_T)$ for the subformula $\varphi[i]$. In other words, $t^l_{i,m,s} = t_1$ and $t^r_{i,m,s} = t_2$ if and only if $[t_1, t_2)$ corresponds to the mth interval of $\mathcal{I}_{\varphi[i]}(\vec{x}^s_T)$.

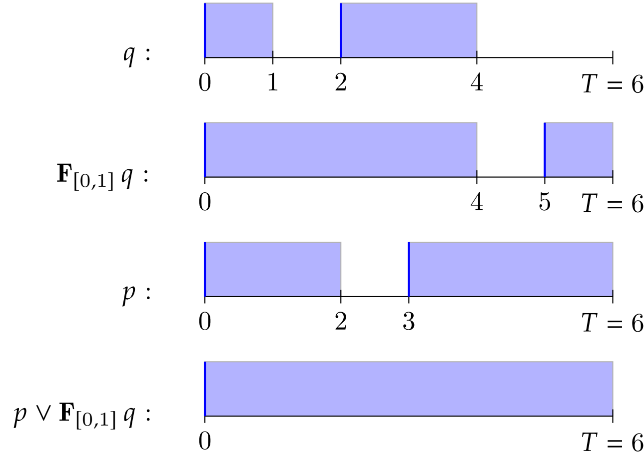


Figure 4.1: Computation of $\mathcal{I}_\varphi(u_1)$ on u_1 for every subformula φ of $\varphi_2 = p \vee \mathbf{F}_{[0,1]}\, q$

To ensure that the variables $t^l_{i,m,s}$ and $t^r_{i,m,s}$ retain their intended meaning, we establish constraints for each operator based on the relations specified in Equation 4.5. Presented below are the constraints for the specific MTL operators.

Constraints for the ¬ operator. For the ¬ operator, we have the following constraints:

$$\bigwedge_{\substack{1 \le i \le n \\ 1 \le j < i}} x_{i,\neg} \wedge l_{i,j} \rightarrow \Big[\bigwedge_{1 \le s \le |\mathcal{S}|} comp_s(i,j) \Big],$$

where, for every $\vec{x}^s_T$ in $\mathcal{S}$, $comp_s(i,j)$ encodes that the set $\mathcal{I}^{\cup}_{\varphi[i]}(\vec{x}^s_T)$ is the complement of the set $\mathcal{I}^{\cup}_{\varphi[j]}(\vec{x}^s_T)$. We construct $comp_s(i,j)$ as follows:

$$\texttt{ite}(t^l_{j,1,s} = 0, \tag{4.6}$$

$$\bigwedge_{1 \le m \le \mathcal{M}-1} t^l_{i,m,s} = t^r_{j,m,s} \wedge t^r_{i,m,s} = t^l_{j,m+1,s}, \tag{4.7}$$

$$t^l_{i,1,s} = 0 \wedge t^r_{i,1,s} = t^l_{j,1,s} \wedge \tag{4.8}$$

$$\bigwedge_{1 \le m \le \mathcal{M}-1} t^l_{i,m+1,s} = t^r_{j,m,s} \wedge t^r_{i,m+1,s} = t^l_{j,m+1,s}),$$

where `ite` is a syntactic sugar for the "if-then-else" construct over LRA formulas, which is standard in many SMT solvers. Here, Condition 4.6 checks whether the left bound of the first interval of $\mathcal{I}_{\varphi[j]}(\vec{x}^s_T)$, encoded by $t^l_{j,1,s}$, is 0. If that holds, as specified by Constraint 4.7, the left bound of the first interval of $\mathcal{I}_{\varphi[i]}(\vec{x}^s_T)$, encoded by $t^l_{1,i,s}$, will be

the right bound of the first interval of $\mathcal{I}_{\varphi[j]}(\vec{x}_T^s)$, encoded $t^r_{1,j,s}$ and so on. If Condition 4.6 does not hold, as specified by Constraint 4.8, the left bound of the first interval of $\mathcal{I}_{\varphi[i]}(\vec{x}_T^s)$ will start with 0, and so on.

As an example, for a prefix $\vec{x}_T^s$ and $T = 7$, let $\mathcal{I}_{\varphi[j]}(\vec{x}_T^s) = \{[0,4),[6,7)\}$. Then, Constraint 4.7 ensures that $\mathcal{I}_{\varphi[i]}(\vec{x}_T^s) = \{[4,6)\}$[4]. Conversely, if $\mathcal{I}_{\varphi[j]}(\vec{x}_T^s) = \{[1,4),[6,7)\}$, then Constraints 4.8 ensures that $\mathcal{I}_{\varphi[i]}(\vec{x}_T^s) = \{[0,1),[4,6)\}$.

For any interpretation ι, let $\mathcal{I}_i^\iota = \{[\iota(t^l_{i,1,s}), \iota(t^r_{i,1,s})), \dots, [\iota(t^l_{i,m,s}), \iota(t^r_{i,m,s}))\}$ denote the set of intervals constructed by the interpretation of the $t^l_{i,m,s}$ and $t^r_{i,m,s}$ variables. We define $\mathcal{I}_j^\iota$ similarly. Then, we can prove the following lemma

Lemma 7 *Let ι be a satisfying interpretation of $comp_s(i,j)$. Then, the set $\mathcal{I}_i^\iota$ consists of the maximal disjoint intervals of the complement of $\mathcal{I}_j^\iota$.*

Proof of Lemma 7. For simplicity of the proof, we name $\iota(t^\sigma_{\kappa,m})$ as $\tau^\sigma_{\kappa,m}$ for $\sigma \in \{l,r\}$ and $\kappa \in \{i,j,j'\}$, and $[\tau^l_{\kappa,m}, \tau^r_{\kappa,m})$ as $\Gamma_{\kappa,m}$ for $\kappa \in \{i,j,j'\}$. Note that we drop the identifier s representing the prefix since the prefix is fixed throughout the proof.

For the forward direction, we show that if $t \in \Gamma_{i,m}$ for some $\Gamma_{i,m} \in \mathcal{I}_i^\iota$ then $t \notin \Gamma_{j,m'}$ for any $\Gamma_{j,m'} \in \mathcal{I}_j^\iota$. First, let $m = 1$. Then, if $\tau^l_{j,1} = 0$, then Condition 4.6 gets triggered and $\tau^l_{i,1} = \tau^r_{j,1}$ and $\tau^r_{i,1} = \tau^r_{j,2}$. Hence, $\tau^r_{j,1} = \tau^l_{i,1} \leq t < \tau^r_{i,1} = \tau^l_{j,2}$. Also, if $\tau^l_{j,1} \neq 0$, then Condition 4.6 does not get triggered and $\tau^l_{i,1} = 0$ and $\tau^r_{i,1} = \tau^l_{j,1}$. Hence, $0 = \tau^l_{i,1} \leq t < \tau^r_{i,1} = \tau^l_{j,1}$. For $m \neq 1$, the reasoning works similarly.

For the other direction, we show that if $t \in \Gamma_{j,m}$ for some $\Gamma_{j,m} \in \mathcal{I}_j$ then $t \notin \Gamma_{i,m'}$ for any $\Gamma_{i,m'} \in \mathcal{I}_j$. The proof for this direction is almost identical to the proof for the forward direction and is a simple exercise. □

Constraints for the $\vee$ operator. For the $\vee$ operator, we have the following constraint:

$$\bigwedge_{\substack{1\leq i\leq n\\ 1\leq j,j'<i}} x_{i,\vee} \wedge l_{i,j} \wedge r_{i,j'} \rightarrow \Big[\bigwedge_{1\leq s\leq|\mathcal{S}|} union_s(i,j,j')\Big],$$

where, for every $\vec{x}_T^s$ in $\mathcal{S}$, $union_s(i,j,j')$ encodes that $\mathcal{I}_{\varphi[i]}(\vec{x}_T^s)$ consists of the maximal disjoint intervals obtained from the union of the intervals in $\mathcal{I}_{\varphi[j]}(\vec{x}_T^s)$ and $\mathcal{I}_{\varphi[j']}(\vec{x}_T^s)$. We

[4]The number of intervals in $\mathcal{I}_{\varphi[i]}(\vec{x}_T^s)$ may differ for different subformulas $\varphi[i]$, which we address at a later point.

construct $union_s(i,j,j')$ as follows:

$$\bigwedge_{1\leq m\leq \mathcal{M}} \left(t^l_{i,m,s} = \bigvee_m (t^l_{j,m,s} \vee t^l_{j',m,s}) \wedge t^r_{i,m,s} = \bigvee_m (t^r_{j,m,s} \vee t^r_{j',m,s}) \right) \wedge \tag{4.9}$$

$$\bigwedge_{\sigma\in[l,r]} \bigwedge_{1\leq m\leq \mathcal{M}} \left((t^\sigma_{j,m,s} = \bigvee_m t^\sigma_{i,m,s}) \iff (t^\sigma_{j,m,s} \notin \bigwedge_m I_{j',m,s}) \right) \wedge \tag{4.10}$$

$$\bigwedge_{\sigma\in[l,r]} \bigwedge_{1\leq m\leq \mathcal{M}} \left((t^\sigma_{j',m,s} = \bigvee_m t^\sigma_{i,m,s}) \iff (t^\sigma_{j',m,s} \notin \bigwedge_m I_{j,m,s}) \right), \tag{4.11}$$

where $I_{k,m,s}$ denotes the interval encoded by bounds $t^l_{k,m,s}$ and $t^r_{k,m,s}$[5]. Here, Constraint 4.9 states that the left (resp., right) bound of each interval of $\mathcal{I}_{\varphi[i]}(\vec{x}^s_T)$, encoded by $t^l_{i,m,s}$ (resp., $t^r_{i,m,s}$) corresponds to one of the left (resp., right) bounds of the intervals in $\mathcal{I}_{\varphi[j]}(\vec{x}^s_T)$ or in $\mathcal{I}_{\varphi[j']}(\vec{x}^s_T)$. Then, Constraint 4.10 states that for each interval I in $\mathcal{I}_{\varphi[j]}(\vec{x}^s_T)$, the left (resp., right) bound of I should appear as the left (resp., right) bound of some interval in $\mathcal{I}_{\varphi[i]}(\vec{x}^s_T)$ if and only if the left (resp., right) bound of I is not included in any of the intervals in $\mathcal{I}_{\varphi[j']}(\vec{x}^s_T)$. Constraint 4.11 mimics the statement made by Constraint 4.10 but for the bounds of the intervals in $\mathcal{I}_{\varphi[j']}(\vec{x}^s_T)$.

For an illustration, assume that $\mathcal{I}_{\varphi[j]}(\vec{x}^s_T) = \{[1,4),[6,7)\}$ and $\mathcal{I}_{\varphi[j']}(\vec{x}^s_T) = \{[3,5),[6,7)\}$ for a prefix $\vec{x}^s_T$ and $T=7$. Now, if $\varphi[i] = \varphi[j] \vee \varphi[j']$, then $\mathcal{I}_{\varphi[i]}(\vec{x}^s_T) = \{[1,5),[6,7)\}$ based on the relation for $\vee$-operator in Equation 4.5. Observe that all the bounds of the intervals in $\mathcal{I}_{\varphi[i]}(\vec{x}^s_T)$, i.e., 1, 5, 6, and 7, are present as the bounds of the intervals in either $\mathcal{I}_{\varphi[j]}(\vec{x}^s_T)$ or $\mathcal{I}_{\varphi[j']}(\vec{x}^s_T)$. This fact is in accordance with Constraint 4.9. Also, the right bound of $[1,4)$ in $\mathcal{I}_{\varphi[j]}(\vec{x}^s_T)$ does not appear as a bound of any intervals in $\mathcal{I}_{\varphi[i]}(\vec{x}^s_T)$, as it is included in an interval in $\mathcal{I}_{\varphi[j']}(\vec{x}^s_T)$, i.e., $4\in[3,5)$. This is in accordance with Constraint 4.10.

For an interpretation ι, we reuse all the naming conventions used in the proof of Lemma 7. Now, to prove the correctness of our encoding of $union_s(i,j,j')$, we can prove the following lemma

Lemma 8 *Let ι be a satisfying interpretation of $union_s(i,j,j')$. Then, the set $\mathcal{I}^\iota_i$ consists of the maximal disjoint intervals of the union of $\mathcal{I}^\iota_j$ and $\mathcal{I}^\iota_{j'}$.*

Proof of Lemma 8. We retain the naming convention of $\tau^\sigma_{\kappa,m}$ and $\Gamma_{\kappa,m}$ as before.

For the forward direction, we show that any time point $t\in\Gamma_{i,m}$ belongs to some $\Gamma_{j,m'}\in\mathcal{I}^\iota_j$ or some $\Gamma_{j',m''}\in\mathcal{I}^\iota_{j'}$. Towards contradiction, we assume that $t\notin\Gamma_{j,m'}$ for any $\Gamma_{j,m'}\in\mathcal{I}^\iota_j$ and $t\notin\Gamma_{j',m''}$ for any $\Gamma_{j',m''}\in\mathcal{I}^\iota_{j'}$. Now, based on Constraint 4.9, both $\tau^l_{i,m}$ and $\tau^r_{i,m}$ appear in some intervals in $\mathcal{I}_j$ and $\mathcal{I}^\iota_{j'}$ as left and right bound, respectively. We consider two cases based on where $\tau^l_{i,m}$ and $\tau^r_{i,m}$ appear. First, $\tau^l_{i,m}$ and $\tau^r_{i,m}$ both

[5] In LRA, $t\notin[t_1,t_2)$ can be encoded as $t<t_1 \vee t\geq t_2$.

appears, w.l.o.g, in $\mathcal{I}_j$. Now, let Γ_{j,m_1} and Γ_{j,m_1+1} be such that $\tau^r_{j,m_1} \le t < \tau^l_{j,m_1+1}$. Intuitively, this means that t lies in between (and is adjacent to) the intervals Γ_{j,m_1} and Γ_{j,m_1+1}. Note that both τ^r_{j,m_1} and τ^l_{j,m_1+1} is not included in $\mathcal{I}^\iota_i$ since $\mathcal{I}^\iota_i$ consists of maximal disjoint intervals and $[\tau^r_{j,m_1}, \tau^l_{j,m_1+1}] \subset \Gamma_{i,m}$. Now, based on Constraint 4.10, τ^r_{j,m_1} and τ^l_{j,m_1+1} are included in some intervals in $\mathcal{I}^\iota_{j'}$. Note that if they are included in the same interval, then that interval also contains t raising the contradiction to our assumption that $t \notin \Gamma_{j',m''}$ for any $\Gamma_{j',m''} \in \mathcal{I}^\iota_{j'}$. Then τ^r_{j,m_1} and τ^l_{j,m_1+1} are not included in the same interval in $\mathcal{I}^\iota_{j'}$. Then, there exists $\Gamma_{j',m_2} \in \mathcal{I}^\iota_{j'}$ and $\Gamma_{j',m_2+1} \in \mathcal{I}^\iota_{j'}$ such that,

$$\tau^r_{j,m_1} < \tau^r_{j',m_2} \le t < \tau^l_{j',m_2+1} < \tau^l_{j,m_1+1}$$

Now note that, τ^r_{j',m_2} and τ^l_{j',m_2+1} both are not included in any of the intervals in $\mathcal{I}_j$. Now, based on Constraint 4.11, both appear in $\mathcal{I}^\iota_i$. But that raises the contradiction to our assumption that $t \in \Gamma_{i,m}$.

For the other direction, we show that any time point, w.l.o.g, $t \in \Gamma_{j,m}$ belongs to some $\Gamma_{i,m'} \in \mathcal{I}^\iota_j$. For this, there can be three cases based on whether the bounds of $\Gamma_{j,m}$ appear as bounds in some interval $\Gamma_{i,m'} \in \mathcal{I}^\iota_i$ or not.

First, assume that both $\tau^l_{j,m}$ and $\tau^r_{j,m}$ appear as bounds τ^l_{i,m_1} and τ^r_{i,m_2} in $\mathcal{I}_k$ as stated by Constraint 4.9. We now claim that $m_1 = m_2$ meaning that τ^l_{i,m_1} and τ^r_{i,m_2} are bounds of the same intervals. Towards contradiction, let $m_1 + 1 \le m_2$. Then, τ^r_{i,m_1} belongs to the interval $\Gamma_{j,m}$, and based on Constraint 4.10, and cannot be one of the bounds of Γ_{i,m_1}. Then, we have $\tau^l_{j,m} = \tau^l_{i,m_1} \le t < \tau^r_{i,m_1} = \tau^r_{j,m}$

Second, assume that $\tau^l_{j,m}$ does not appear, while $\tau^r_{j,m}$ appears as bounds in $\mathcal{I}_k$. Now, based on Constraint 4.10, $\tau^l_{j,m}$ appears in one of the intervals $\Gamma_{j',m'}$ in $\mathcal{I}^\iota_{j'}$. Also, in that case, $\tau^l_{j',m'}$ appears as a left bound in $\mathcal{I}_k$, say $\mathcal{I}_{i,m_1}$. We now claim that $\tau^r_{i,m_1} > \tau^r_{j,m}$. Towards contradiction, we assume two cases. In the first case,

$$\tau^l_{j',m'} = \tau^l_{i,m_1} < \tau^r_{i,m_1} < \tau^l_{j,m} < \tau^r_{j',m}$$

contradicting Constraint 4.10. In the second case,

$$\tau^l_{j',m'} = \tau^l_{i,m_1} < \tau^l_{j,m} < \tau^r_{i,m_1} < \tau^r_{j,m}$$

contradicting Constraint 4.11. From the two cases, we conclude $\tau^r_{i,m_1} > \tau^r_{j,m}$ and hence, $\tau^l_{i,m_1} < \tau^l_{j,m} \le t < \tau^r_{j,m} < \tau^r_{i,m_1}$. The argument in the third case is similar to the arguments in the other two cases and can be seen easily. □

Constraints for the $\wedge$ operator. For the $\wedge$-operator, we have the following constraint:

$$\bigwedge_{\substack{1\le i\le n\\ 1\le j,j'<i}} x_{i,\wedge} \wedge l_{i,j} \wedge r_{i,j'} \rightarrow \Big[\bigwedge_{1\le s\le|\mathcal{S}|} comp_s(i,k) \wedge union_s(k,j_1,j'_1) \wedge comp_s(j_1,j) \wedge comp_s(j'_1,j') \Big]$$

This encodes the relation for the $\wedge$ operator as described in Equation 4.5. We introduce some intermediate set of intervals $\tilde{\mathcal{I}}_k$, $\tilde{\mathcal{I}}_{j_1}$ and $\tilde{\mathcal{I}}_{j'_1}$ such that $\tilde{\mathcal{I}}_{j_1}$ and $\tilde{\mathcal{I}}_{j'_1}$ are the

complement of $\mathcal{I}^{\cup}_{j}(\vec{x}^s_T)$ and $\mathcal{I}^{\cup}_{j'}(\vec{x}^s_T)$ respectively. Then, $\tilde{\mathcal{I}}_k$ contains the maximal disjoint intervals of union of $\tilde{\mathcal{I}}_{j_1}$ and $\tilde{\mathcal{I}}_{j'_1}$. Finally, $\mathcal{I}^{\cup}_{i}(\vec{x}^s_T)$ is the complement of the set of intervals in $\tilde{\mathcal{I}}_k$, making it the set containing maximal disjoint intervals of *intersection* of $\mathcal{I}^{\cup}_{\varphi[j]}(\vec{x}^s_T)$ and $\mathcal{I}^{\cup}_{\varphi[j']}(\vec{x}^s_T)$. The main idea is that we encode intersection in the semantics constraints for the $\land$ operator using union and complement following *De Morgan's laws* ($A \cap B = (A^c \cup B^c)^c$).

For an illustration, assume that $\mathcal{I}_{\varphi[j]}(\vec{x}^s_T) = \{[1,4),[6,7)\}$ and $\mathcal{I}_{\varphi[j']}(\vec{x}^s_T) = \{[3,5),[6,7)\}$ for a prefix $\vec{x}^s_T$ and $T = 7$. Now, if $\varphi[i] = \varphi[j] \land \varphi[j']$, then based on the relation for $\lor$-operator in Equation 4.5 $\mathcal{I}_{\varphi[i]}(\vec{x}^s_T) = \{[3,4),[6,7)\}$ based on the relation for $\land$ operator in Equation 4.5. One can check that a satisfying interpretation of the above encoding will encode $\tilde{\mathcal{I}}_{j_1} = \{[4,6)\}$ and $\tilde{\mathcal{I}}_{j'_1} = \{[1,3),[5,6)\}$. Then, correspondingly the set $\tilde{\mathcal{I}}_k$ will be encoded as their union, i.e., $\{[1,3),[4,6)\}$. Finally, $\mathcal{I}_{\varphi[i]}(\vec{x}^s_T)$ will be encoded as the complement of $\tilde{\mathcal{I}}_k$, which is $\{[3,4),[6,7)\}$.

Constraints for the $\mathbf{F}_I$ *operator.* Next, for the $\mathbf{F}_I$-operator where I is encoded using a_i and b_i, we have the following constraint:

$$\bigwedge_{\substack{1\leq i\leq n\\ 1\leq j<i}} x_{i,\mathbf{F}_I} \land l_{i,j} \rightarrow \Big[\bigwedge_{1\leq s\leq|\mathcal{S}|} union'_s(i,k,k) \land \ominus_s^{[a_i,b_i]}(k,j)\Big].$$

based on the relation for the $\mathbf{F}_{[a,b]}$ operator in Equation 4.5. We here rely on an intermediate set of intervals $\tilde{\mathcal{I}}_k$ encoded using some auxiliary variables $\tilde{t}^l_{k,m,s}$ and $\tilde{t}^r_{k,m,s}$ where $m \in \{1,\dots,\mathcal{M}\}$ and $s \in \{1,\dots,|\mathcal{S}|\}$. Also, we use the formula $\ominus_s^{[a_i,b_i]}(k,j)$ to encode that the intervals in $\tilde{\mathcal{I}}_k$ can be obtained by performing $I \ominus [a,b]$ to each interval I in $\mathcal{I}_{\varphi[j]}(\vec{x}^s_T)$, where $a_i = a$ and $b_i = b$. Finally, the formula $union'(i,k,k)$ encodes that $\mathcal{I}_{\varphi[i]}(\vec{x}^s_T)$ consists of the maximal disjoint intervals obtained from the union of the intervals in $\tilde{\mathcal{I}}_k$ and $\{[T-b,T)\}$.

The construction of $union'(i,k,k)$ is similar to that of $union(i,j,j')$ in that the constraints involved are similar to Constraints 4.9 to 4.11. For $\ominus_s^{[a_i,b_i]}(k,j)$, we have the following constraint:

$$\bigwedge_{1\leq m\leq\mathcal{M}-1} \Big[\tilde{t}^l_{k,m,s} = \max\{0,\Big(t^l_{j,m,s} - b_i\Big)\} \land \tilde{t}^r_{k,m,s} = \max\{0,\Big(t^r_{j,m,s} - a_i\Big)\}\Big] \tag{4.12}$$

As an example, consider $\mathcal{I}_{\varphi[j]}(\vec{x}^s_T) = \{[1,4),[6,7)\}$ for a prefix $\vec{x}^s_T$ and $T = 7$. Now, if $\varphi[i] = \mathbf{F}_{[1,4]}\,\varphi[j]$, then first we have $\tilde{\mathcal{I}}_k = \{[0,3),[2,6)\}$ based on Constraint 4.12. Note that, while the intervals in $\tilde{\mathcal{I}}_k$ may not be disjoint, $union'(i,k,k)$ ensures that $\mathcal{I}_{\varphi[i]}(\vec{x}^s_T)$ consists of only maximal disjoint intervals. Next, we have $\mathcal{I}_{\varphi[i]}(\vec{x}^s_T) = \{[0,7)\}$ which consists of the maximal disjoint intervals from the union of $\tilde{\mathcal{I}}_k = \{[0,3),[2,6)\}$ and $\{[T-b,T) = [3,7)\}$ using $union'(i,k,k)$.

The proof of correctness of the encoding for $\ominus_s^{[a_i,b_i]}(k,j)$ is straightforward and easy to check from the construction. In particular, we can prove the following lemma:

Lemma 9 *Let ι be a satisfying interpretation of $\ominus_s^{[a_i,b_i]}(k,j)$ such that $\iota(a_i) = a$ and $\iota(b_i) = b$. Then, the set $\mathcal{I}_i^\iota$ consists of the maximal disjoint intervals by applying $I \ominus [a,b]$ to the intervals I of $\mathcal{I}_j^\iota$.*

Constraints for the $\mathbf{G}_I$ operator. For the $\mathbf{G}_I$ operator where I is encoded using a_i, b_i we have the following constraint:

$$\bigwedge_{\substack{1\leq i\leq n\\1\leq j<i}} x_{i,\mathbf{G}_I} \wedge l_{i,j} \rightarrow \Big[\bigwedge_{1\leq s\leq|\mathcal{S}|} union''_s(i,k',k') \wedge comp_s(k',k) \wedge \ominus_s^{[a_i,b_i]}(k,j_1) \wedge comp_s(j_1,j)\Big].$$

based on the relation for the $\mathbf{G}_{[a,b]}$ operator in Equation 4.5. Similar to the encoding of $\mathbf{F}_{[a,b]}$ operator, we rely on intermediate set of intervals $\tilde{\mathcal{I}}_{j_1}$, $\tilde{\mathcal{I}}_k$ and $\tilde{\mathcal{I}}_{k'}$ encoded using some auxiliary variables. First we use the formula $comp_s(j_1,j)$ to encode that $\tilde{\mathcal{I}}_{j_1}$ is the complement of the set $\mathcal{I}_{\varphi[j]}(\vec{x}_T^s)$. Then we use the formula $\ominus_s^{[a_i,b_i]}(k,j_1)$ to encode that the intervals in $\tilde{\mathcal{I}}_k$ can be obtained by performing $I \ominus [a,b]$ to each interval I in $\tilde{\mathcal{I}}_{j_1}$, where $a_i = a$ and $b_i = b$. Then $comp_s(k',k)$ encodes that $\tilde{\mathcal{I}}_{k'}$ is the complement of $\tilde{\mathcal{I}}_k$. Finally, the formula $union''(i,k',k')$ encodes that $\mathcal{I}_{\varphi[i]}(\vec{x}_T^s)$ consists of the maximal disjoint intervals obtained by taking the union of the complement of $\mathcal{I}^{\cup}_{\varphi[j]}(\vec{x}_T^s)$ and $\{[T-a,T)\}$.

Similar to $union'$ in the semantic constraints for $\mathbf{F}_I$ operator, the construction of $union''(i,k,k)$ is similar to that of $union(i,j,j')$ in that the constraints involved are similar to Constraints 4.9 to 4.11.

As an example, consider $\mathcal{I}_{\varphi[j]}(\vec{x}_T^s) = \{[1,4),[6,7)\}$ for a prefix $\vec{x}_T^s$ and $T=7$. Now, if $\varphi[i] = \mathbf{G}_{[1,4]}\,\varphi[j]$, first we have $\tilde{\mathcal{I}}_{j_1} = \{[0,1),[4,6)\}$ using $comp_s(j_1,j)$. Then we have $\tilde{\mathcal{I}}_k = \{[0,5)\}$ based on Constraint 4.12 applied on $\tilde{\mathcal{I}}_{j_1}$. Now using the formula $comp_s(k',k)$, we have $\tilde{\mathcal{I}}_{k'} = \{[5,7)\}$. Next, we have $\mathcal{I}_{\varphi[i]}(\vec{x}_T^s) = \{[5,7)\}$ which consists of the maximal disjoint intervals from the union of $\tilde{\mathcal{I}}_{k'} = \{[5,7)\}$ and $\{[T-a,T) = [6,7)\}$ using $union''(i,k',k')$.

It is worth noting that although the number of intervals in $\mathcal{I}_{\varphi[i]}(\vec{x}_T^s)$ for each subformula $\varphi[i]$ is bounded by $\mathcal{M}$, it may not contain the same number of intervals. For instance, $\mathcal{I}_p(\vec{x}_T^s) = \{[0,1),[6,7)\}$ has two intervals, while, assuming $T=7$, $\mathcal{I}_{\neg p}(\vec{x}_T^s) = \{[1,6)\}$ has only one interval.

To circumvent this, we introduce some variables $num_{i,s}$ for $i \in \{1,\ldots,n\}$ and $s \in \{1,\ldots,|\mathcal{S}|\}$ to track of the number of intervals in $\mathcal{I}_{\varphi[i]}(\vec{x}_T^s)$ for each subformula $\varphi[i]$ for each prefix $\vec{x}_T^s$. We now impose the constraint

$$\bigwedge_{1\leq i\leq n, 1\leq m\leq\mathcal{M}} [m > num_{i,s}] \rightarrow [t^l_{i,m,s} = T \wedge t^r_{i,m,s} = T]$$

This ensures that all the unused variables $t^\sigma_{i,m,s}$ for each node i and prefix $\vec{x}_T^s$ in $\mathcal{S}$ are all set to T. We also use the $num_{i,s}$ variables in the constraints for easier computation of $\mathcal{I}_{\varphi[i]}(\vec{x}_T^s)$ for each operator. We include this in our implementation but omit it here for a simpler presentation.

Constraints for ensuring **G**-sep. Finally, to ensure that the prospective formula φ is **G**-sep for $\mathcal{S}$, we add:

$$\bigwedge_{\vec{x}^s_T \in P} \big[(t^l_{n,1,s} = 0) \wedge (t^r_{n,1,s} = T)\big] \wedge \bigwedge_{\vec{x}^s_T \in N} \big[(t^l_{n,1,s} \neq 0) \vee (t^r_{n,1,s} \neq T)\big]. \tag{4.13}$$

This constraint says that $\mathcal{I}_{\varphi[n]}(\vec{x}^s_T) = \{[0,T)\}$ for all the positive prefixes $\vec{x}^s_T$, while $\mathcal{I}_{\varphi[n]}(\vec{x}^s_T) \neq \{[0,T)\}$ for any negative prefixes $\vec{x}^s_T$.

4.5 Theoretical Guarantees

This section establishes the theoretical guarantees of our MTL learning algorithm.

4.5.1 Correctness

The correctness of our algorithm follows from the correctness of the inductive computation of $\mathcal{I}_\varphi(\vec{x}_T)$ in Lemma 6 and its encoding using the formulas described in Lemma 7, Lemma 8 and Lemma 9. We state the correctness result formally as follows:

Theorem 4 (Correctness) *Given a sample $\mathcal{S}$ and a future-reach bound k, if Algorithm 5 outputs an MTL formula φ, then φ is a minimal MTL formula such that φ is globally separating for $\mathcal{S}$ and $fr(\varphi) \leq k$.*

4.5.2 Termination

The termination of our algorithm is guaranteed by the choice of imposing a size bound B in Algorithm 5 on the search space of the MTL formulas as described in Section 4.3.3.

4.5.3 Complexity Results

Our synthesis algorithm solves the optimization problem SynTL by constructing formulas in LRA. We now analyze the computational hardness of SynTL and, thus, consider its *corresponding decision problem SynTL$_d$*: given a sample $\mathcal{S}$, a future-reach bound k and size bound B (in unary), does there exist an MTL formula φ such that φ is **G**-sep for $\mathcal{S}$, $fr(\varphi) \leq k$, and $|\varphi| \leq B$?

Following our algorithm, we can encode the SynTL$_d$ problem in an LRA formula $\Phi = \bigvee_{n \leq B} \Phi^n_{\mathcal{S},k}$, where $\Phi^n_{\mathcal{S},k}$ is as described in Algorithm 5. One can check that the size of Φ is $\mathcal{O}(|\mathcal{S}||k|B^3\mathcal{M}^3)$. Now, the fact that the satisfiability of an LRA formula is NP-complete [40] proves the following:

Theorem 5 *SynTL$_d$ is in NP.*

While the exact complexity lower bound for $\textsc{SynTL}_d$ is unknown, we conjecture that $\textsc{SynTL}_d$ is NP-hard. Our hypothesis stems from the fact that one can show $\textsc{SynTL}_d$ is NP-hard for a simple fragment MTL($\mathbf{G}_I, \vee, \neg$), which consists of only $\mathbf{G}_I$, $\vee$ and $\neg$ operators[6], following the techniques of Fijalkow and Lagarde [58].

Lemma 10 *$\textsc{SynTL}_d$ is NP-hard for MTL($\mathbf{G}_I, \vee, \neg$).*

Proof of Lemma 10. To prove the NP-hardness of $\textsc{SynTL}_d$ for MTL($\mathbf{G}_I, \vee, \neg$), we establish the NP-hardness of an easier problem where the future-reach bound is relaxed, which is the following: given a sample $\mathcal{S}$ and a size bound B, does there exist a formula φ in MTL($\mathbf{G}_I, \vee, \neg$) such that φ is $\mathbf{G}$-sep for $\mathcal{S}$, and $|\varphi| \leq B$? Towards this, we show a polynomial time reduction from the *hitting set* problem, a classical NP-complete problem in the literature. To this end, let us first define the hitting set decision problem: given $C_1, \ldots, C_n$ subsets of $[1, \ell]$ and $k \in \mathbb{N}$, does there exist H subset of $[1, \ell]$ of size at least k such that for every $j \in [1, n]$ we have $H \cap C_j \neq \emptyset$. In that case, we say that H is a hitting set.

We construct a reduction from the hitting set problem. Let $C_1, \ldots, C_n$ subsets of $[1, \ell]$ and $k \in \mathbb{N}$. Let us consider the set of propositions to be $\mathcal{P} = \{p_0, p_1, \ldots, p_\ell\}$. We construct the sample $\mathcal{S} = (P, N)$ with $T = l + 1$ as follows: for each $j \in [1, n]$ we let $[1, \ell] \setminus C_j = \{a_{j,1} < \cdots < a_{j,m_j}\}$, and define a positive signal prefix of the form

$$u_j = 0 : \{p_0\}; a_{j,1} : \{p_{a_{j,1}}\}; \ldots; a_{j,m_j} : \{p_{a_{j,m_j}}\}.$$

Let $P = \{u_1, \ldots, u_n\}$ be the set of all the positive signal prefixes. There is a single negative signal prefix, which is of the form

$$v = 0 : \{p_0\} ; 1 : \{p_1\} ; 2 : \{p_2\} ; \ldots ; \ell : \{p_\ell\}.$$

that means in v, only proposition p_i is true. We let N denote the singleton set containing v at time interval $[i, i + 1)$.

We claim that there exists a hitting set of size at most k if and only if there exists a formula in MTL($\mathbf{G}_I, \vee, \neg$) of size at most $3k - 1$ that is globally separating for $\mathcal{S}$, i.e., satisfies u_i's at all time-points and does not satisfy v at some time-point.

Let $H = \{c_1, \ldots, c_k\}$ be a hitting set of size k with $c_1 < c_2 < \cdots < c_k$, we construct the formula

$$\varphi = (\neg p_{c_1} \vee \mathbf{G}_{[0,\ell]}(\cdots \vee \mathbf{G}_{[0,\ell]} \neg p_{c_k}))$$

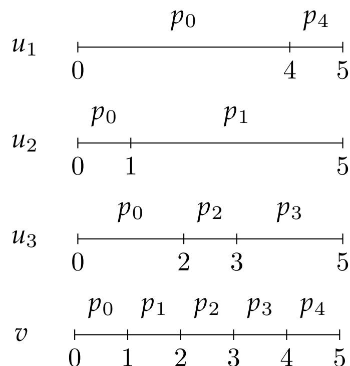
We argue that φ globally separates $u_1 \ldots u_n$ from v and has size $3k - 1$. Indeed, the fact that H is a hitting set means that for every $j \in [1, n]$, there exists i such that $c_i \in H$. This implies that u_j satisfies $\mathbf{G}_{[0,\ell]} \neg p_{c_i}$ globally, hence φ as well. Also, v does not satisfy φ at position c_1.

Conversely, let φ be a $\mathbf{G}$-sep formula in MTL($\mathbf{G}_I, \vee, \neg$) of size $3k - 1$. Following [58], we can assume that φ is of the form above. Because it does not satisfy v, we have $c_1 < c_2 < \cdots < c_k$. We let $H = \{c_1, \ldots, c_k\}$, and argue that H is a hitting set. To prove that H is a hitting set, we need to prove that, for every $j \in [1, n]$, we have $H \cap C_j \neq \emptyset$.

[6]we consider the fragment in negation normal form

Now as φ is $\mathbf{G}$-sep, for every $j \in [1, n]$, we have $(u_j, t) \models \varphi$ for all $t \in [0, \ell]$. Then there exists a c_i that does not appear in u_j, implying the fact that $c_i \in H \cap C_j$ by the construction of u_j.

We illustrate the above idea via an example. Let l be four and $C_1, C_2, C_3 \subseteq [1, 4]$ such that, $C_1 = \{1, 2, 3\}, C_2 = \{2, 3, 4\}$ and $C_3 = \{1, 4\}$. Then, we construct the sample $\mathcal{S}$ with $T = 5$ as follows:



where u_1, u_2, u_3 are positive prefixes and v is the negative prefix. The prefixes illustrated in the above figure are meant to be interpreted as a proposition written between two numbers $n_1, n_2 \in \mathbb{N}$ holds in the interval $[n_1, n_2)$. For example the prefix u_1 is such that, $\{p_0\}$ is $true$ in $[0, 4)$ and $\{p_4\}$ is $true$ in $[4, 5)$. Now note that a hitting set for C_1, C_2, C_3 is $H = \{2, 4\}$ such that, $|H| = 2$. Then the corresponding $\mathbf{G}$-sep formula for $\mathcal{S}$ is $\varphi = \neg p_2 \vee \mathbf{G}_{[0,4]} \neg p_4$. □

Note that this does not imply the NP-hardness of the SYNTL$_d$ problem for the full fragment, as formulas with more operators could be smaller. However, it does show that the problem is already combinatorially challenging in a very restricted setting. This complexity lower bound justifies our approach of using constraint programming. We leave it as an open problem to extend our complexity lower bound results to full MTL.

4.6 Experimental Evaluation

In this section, we answer the following research questions to assess the performance of our MTL learning algorithm.

RQ1: Can our algorithm synthesize concise formulas with small future-reach?

RQ2: How does lowering the future-reach bound affect the size of the formulas?

RQ3: How does our algorithm scale for different sample sizes?

4.6.1 Experimental Setup

To answer the research questions above, we have implemented a prototype of our algorithm in Python 3 using Z3 [44] as the SMT solver in a tool named `TEAL` (synThesizing Efficiently monitorAble mtL). To our knowledge, `TEAL` is the only tool for synthesizing minimal MTL formulas for monitoring purposes (see related works). In `TEAL`, we implement a heuristic on top of Algorithm 5. We initially set the maximum number of intervals $\mathcal{M}$ in sets $\mathcal{I}_{\varphi[i]}(\vec{x}_T)$ to be $\mu + 2$ where $\mu = \max(\{|\mathcal{I}_p(\vec{x}_T)| \mid p \in \mathcal{P}\})$. We iteratively increase the value of $\mathcal{M}$ until we find a solution. To ensure that the synthesized MTL formulas are correct, we implement a verifier based on the inductive computation of $\mathcal{I}_\varphi(\vec{x}_T)$ mentioned in Equation 4.5. The heuristic improves the runtime of `TEAL` significantly since most $\mathbf{G}$-sep formulas φ never require the worst-case upper bound[7] of $\mathcal{M} = \mu|\varphi|$.

We run `TEAL` on several benchmarks generated synthetically from MTL formulas. To obtain useful MTL formulas, we rely on specifications that describe the time-sensitive requirements of an electronically controlled steering (ECS) system, listed in Figure 5 of [90]. From the specifications listed, we identify three MTL patterns shown in Table 4.1. Note that we choose formulas of the form $G(\varphi)$ as we search for formulas that are globally separating. In our experiments, we generate MTL formulas from these patterns by replacing time interval $[t_1, t_2]$ with different values.

Table 4.1: Time-sensitive MTL requirements of ECS system

Bounded Recurrence:	$\mathbf{G}(\mathbf{F}_{[t_1,t_2]}\, p)$
Bounded Response:	$\mathbf{G}(p \rightarrow \mathbf{F}_{[t_1,t_2]}\, q)$
Bounded Invariance:	$\mathbf{G}(p \rightarrow \mathbf{G}_{[t_1,t_2]}\, q)$

To generate samples, starting with an MTL formula $\mathbf{G}(\varphi)$ from Table 4.1, we generated a set of random prefixes and then classified them into positive or negative depending on whether φ holds at all time-points of the prefix or not. All the experiments are conducted on a single core of a Debian machine with an Intel Xeon E7-8857 CPU (at 3GHz) using up to 6GB of RAM.

4.6.2 RQ1: Performance Evaluation

To address **RQ1**, we ran `TEAL` on a benchmark suite generated from nine MTL formulas obtained from the three MTL patterns in Table 4.1 by replacing t_1 with 0 and t_2 with 1,2, and 3. The suite consists of 36 samples for each pattern (12 samples for each formula), with the number of prefixes ranging from 10 to 40 and the length of prefixes (i.e., the number of sampled time points) ranging from 4 to 6. For each sample $\mathcal{S}$, we set the future-reach bound k to be $fr(\varphi)$ and size bound B to be $|\varphi|$, where $|\varphi|$ is the formula from which $\mathcal{S}$ was generated. The timeout chosen was 5400 secs.

[7]The operators $\mathbf{F}_I$, $\mathbf{G}_I$, $\wedge$, and $\neg$ increase the number of required intervals by at most one. Only the $\vee$ operator can double it in the worst-case.

Table 4.2: Summary of synthesized formulas.

Formula pattern	Num of Samples	Matched	Not Matched	Timed out
Bounded Recurrence	36	36	0	0
Bounded Response	36	24	2	10
Bounded Invariance	36	24	1	11

In this experiment, we noted the formulas synthesized by TEAL. We observed that TEAL synthesized formulas that matched the original MTL pattern in 96.4% of the cases in which it did not time out. We summarize these results in Table 4.2. Thus, to answer RQ1, TEAL demonstrates a good ability to synthesize a concise MTL formula with a small future-reach if one exists.

4.6.3 RQ2: Future-reach vs Size

To address **RQ2**, we investigate how the size of the synthesized formula changed over varying future-reach bounds. For this, we ran TEAL on the same benchmark suite from RQ1 but, this time, by varying the future-reach bound k from 1 to 4. We investigate the average size of the minimal formula we get over the generated 108 samples for each future-reach bound.

We observed that for future-reach bounds k of 1, 2, 3, and 4, the average size of the synthesized minimal formulas were 3.904, 3.734, 3.370, and 3.361, respectively. Thus, the trend is that with an increase in k, the average size of the minimal formula decreased. This is because an increase in k allows a bigger search space of formulas. One can, however, also notice that the decrease in the average size of the formulas with increasing future-reach bound is not vast. This highlights the advantage of using a future-reach bound for synthesizing formulas for online monitoring and confirms the efficacy of our algorithm.

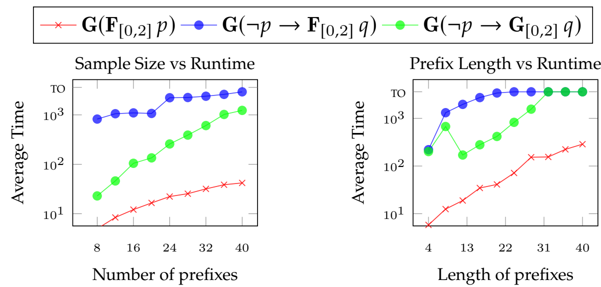


Figure 4.2: Runtime change with respect to the number of prefixes and prefix lengths

4.6.4 RQ3: Scalability

To address **RQ3**, we ran TEAL on a benchmark suite generated from three MTL formulas $\mathbf{G}(\mathbf{F}_{[0,2]}\, p)$, $\mathbf{G}(\neg p \rightarrow \mathbf{F}_{[0,2]}\, q)$, and $\mathbf{G}(\neg p \rightarrow \mathbf{G}_{[0,2]}\, q)$ which originate from the three MTL patterns in Table 4.1. The suite consists of 90 samples for each formula, with the number of prefixes varying from 8 to 40 and the length of prefixes varying from 4 to 40. We set the future-reach bound k to be two and size bound B to be $|\varphi|$ where φ is the original formula.

Figure 4.2 illustrates the runtime variation of TEAL in two cases: increasing the number of prefixes fixing the length of them and increasing the length of prefixes fixing the number of them. We observe that the increase in runtime is steeper relative to the prefix lengths than to the number of prefixes. This is natural because an increase in prefix length increases μ, which significantly affects the size of the encoding. We notice an anomalous drop in runtime for the third formula for prefix length 12. A plausible explanation is that the randomly generated samples with prefixes of length beyond 12 could not capture the property of the formula properly and allowed simpler formulas.

4.7 Discussion

We have presented a novel SMT-based algorithm for automatically learning concise and monitorable MTL specifications from finite system executions. Here we highlight two key points applicable to our approach:

- While our algorithm is tailored to synthesize globally separating formulas particularly useful for monitoring, we can adapt our algorithm easily to learn only *separating* formulas as in the standard temporal logic inference setting [113, 111]. In particular, to ensure that the learnt formula is $\mathbf{G}$-sep, we encode that, for every positive prefix, the set of intervals at the root node has to be $\{[0, T)\}$ and this should not be the case for any of the negative ones, as encoded in Constraint 4.13. To learn only separating formulas, we can modify the constraint as follows to check if only zero is included in the set of intervals at the root node for the positive prefixes and it is not the case for the negative ones:

$$\bigwedge_{\vec{x}^s_T \in P} (t^l_{n,1,s} = 0) \wedge \bigwedge_{\vec{x}^s_T \in N} (t^l_{n,1,s} \neq 0).$$

  This constraint says that $0 \in \mathcal{I}^{\cup}_{\varphi[n]}(\vec{x}^s_T)$ for all the positive prefixes $\vec{x}^s_T$, while $0 \notin \mathcal{I}^{\cup}_{\varphi[n]}(\vec{x}^s_T)$ for any negative prefixes $\vec{x}^s_T$.

- Moreover, while we excluded the $\mathbf{U}_I$ operator for interpretability purposes, we can still incorporate it using similar methods as the other operators. However, efficiently encoding the computation for $\mathbf{U}_I$ symbolically is challenging and could potentially have a notable impact on the algorithm's runtime.

- From a practical point of view, an interesting future direction will be to lift our techniques to automatically learn Signal Temporal Logic (STL) formulas for

verification. Our algorithm can be extended for STL learning in a seemingly naive way, as we discuss in the following section. However, this extension needs to be optimized or we need to search for better optimization and heuristics to make that algorithm efficient. In the following section, we give a sketch of how we can lift our techniques naively to accommodate STL.

4.8 Extension to STL Learning

In this section, we sketch an extension of our algorithm to learn efficiently monitorable STL specifications from system executions. Recall that STL is a temporal logic formalism that extends MTL by incorporating real-valued predicates, as introduced in Section 2.2.3.3. We consider the syntax of STL to be similar to that of MTL, as discussed in Section 4.3. The main difference is that instead of a set of propositions $\mathcal{P}$, we have a set of real-value predicates in the form $x_k \geq c$, where $x_k \in X$ is a real-value variable from a finite set of variables $X = x_0, \ldots, x_n$, and $c \in \mathbb{R}$. The extension is as follows:

- The *signals* and *prefixes* are continuous and defined in an analogous manner, but now they are real-valued. Specifically, they are of the form $\mathbb{R}_{\geq 0} \to \mathbb{R}^{|X|}$ ($[0, T) \to \mathbb{R}^{|X|}$, respectively);
- Each observation is represented as $\Omega = \langle (t_i, \delta_i) \rangle_{i \leq n_{\vec{x}}}$, where $n_{\vec{x}} \in \mathbb{N}$. The observations satisfy the following conditions: (i) $t_0 = 0$ and $t_{n_{\vec{x}}} < T$, (ii) for all $i \leq n_{\vec{x}}$, $\delta_i \in \mathbb{R}^{|X|}$ is an n-dimensional vector where the i-th coordinate indicates the value of the variable x_i at time point t_i. The observations are approximated using piecewise affine continuous prefixes $\vec{x}_T$ (instead of piecewise constant). Specifically, for all $i < n_{\vec{x}}$, $\vec{x}_T$ is continuous at t_i and affine on the interval $[t_i, t_{i+1})$.
- The sample $\mathcal{S}$ and the inputs are defined similarly, where each of the prefixes is real-valued.
- The future-reach value of an STL formula is analogous to that of an MTL formula.
- The problem definition is similar to SynTL, where given a set of positive and negative prefixes, the goal is to find the minimal STL formula within the given future-reach bound that is $\mathbf{G}$-sep for the sample $\mathcal{S}$.

Example 5 *First, we go through the running example presented in Example 3 in the context of MTL learning and motivate the need for STL learning. n this particular example, we assume that the system designer knows the specific measure of the unsafe distance, which is 20 meters. Consequently, the event "there is no preceding car within 20 meters" can be represented as a proposition p. The truth value of p can be evaluated at each observation in the simulations.*

However, in practical scenarios, the precise measure of the unsafe distance may not be known in advance. During simulations, the system designer can only observe the actual distance at each time point. In such cases, the objective is to synthesize similar specifications in the form of "for every time point t, if the preceding vehicle is within a distance of less than 20 meters from the vehicle, the brake should be triggered within one second." This requirement can be expressed as a globally separating STL formula: $(x \geq 20) \vee \mathbf{F}_{[0,1]}\, q$, where the variable x denotes the distance between the test vehicle and the preceding vehicle, and q indicates that the brake is triggered, similar to Example 3.

An immediate observation can be made based on the example mentioned above: if we have prior knowledge of the predicates (e.g., $x \geq 20$ in the example), we can convert the real-valued piecewise continuous affine signals into a piecewise-constant signal with respect to a set of propositions $\mathcal{P}$, which corresponds to the set of predicates. By doing so, we can directly apply our MTL learning algorithm presented in Algorithm 5 and extract the desired STL formulas using that set of predicates.

However, the most intriguing direction for extending our algorithm to STL learning lies in scenarios where the set of predicates is unknown and not provided as input. In the syntax-DAG of STL, the leaf nodes contain predicates of the form $x_k \geq c_k$ for $x_k \in X$ and $c_k \in \mathbb{R}$, instead of propositions as in MTL. Therefore, the key idea for extending our algorithm in this context is to utilize the LRA solver to guess these predicates at the leaf nodes and encode the interval computations properly according to that guess. This will ensure that the resulting STL formula is globally separating.

In the encoding of the semantics of MTL formulas, the base step in Algorithm 5 is as follows: given a proposition $p \in \mathcal{P}$ and a piecewise constant $\vec{x}_T$, we can easily calculate $\mathcal{I}_p(\vec{x}_T^s)$ that consists of the maximal disjoint set of intervals in $\vec{x}_T$ where p holds. Then we encode the interval computations on top of it as explained in Equations 4.5.

However, in the STL framework, the maximal disjoint intervals for leaf nodes are no longer fixed. Specifically, for a real-valued prefix $\vec{x}_T$, the set of intervals $\mathcal{I}_{x_k \geq c_k}(\vec{x}_T^s)$ for the predicate of the form $x_k \geq c_k$ depends on the valuation of c_i. Consequently, similar to semantics encodings for other operators in the MTL setting, we must encode the computation of $\mathcal{I}_{x_k \geq c_k}(\vec{x}_T^s)$. This can be achieved similarly by using variables $t^l_{i,m,s}$ and $t^r_{i,m,s}$ such that a satisfying interpretation that maps c_k to c will satisfy the fact that $\mathcal{I}_{x_k \geq c}(\vec{x}_T^s)$ contains the maximal disjoint intervals $I_1 \ldots I_{\mathcal{M}}$ such that, for all $1 \leq i \leq \mathcal{M}$, the value of $\vec{x}_T(t)(k)$, i.e., the value of the variable x_k in $\vec{x}_T$ at time point t is greater than or equal to c.

Let ι represent an interpretation. We denote $\iota(t^\sigma_{i,m,s})$ as τ^σ_m, indicating the value of the variable according to the interpretation. The subscripts i and s have been omitted for clarity as the context is clear. Regarding the predicate $x_k \geq c$, we define the parity of $\vec{x}_T(\tau^\sigma_{m_1})(k)$ as 1 if the value is greater than or equal to c, and 0 otherwise. We enforce the following constraints:

- For each $m \leq \mathcal{M} - 1$, the parity of $\vec{x}_T(\tau^\sigma_m)(k)$ must differ from the parity of $\vec{x}_T(\tau^\sigma_{m+1})(k)$.
- For each $m \leq \mathcal{M} - 1$, there should be no sampled time point $\tau^\sigma_m < t < \tau^\sigma_m$ in $\vec{x}_T$ where the parity of $\vec{x}_T(\tau^\sigma_m)(k)$ is different from the parity of $\vec{x}_T(t)(k)$.

These constraints ensure the desired conditions for the specified variable and predicate. It is easy to check that the above two constraints encode the fact that $\mathcal{I}_{x_k \geq c}(\vec{x}_T^s)$ contains the maximum disjoint intervals where $x_k \geq c$ in $\vec{x}_T$. It should also be noted that, given a time point t' and a variable x_k, the value of $\vec{x}_T(t')(k)$ can be easily computed through interpolation, based on the values at the sampled points. This is possible as $\vec{x}_T$ is assumed to be piecewise-affine continuous.

These constraints can be encoded in LRA and added, along with other semantic

constraints in Φ^{sem} (excluding the ones for propositions), to Algorithm 5. The Φ^{fr} remains unchanged, while the Φ^{str} is slightly modified to accommodate predicates instead of propositions in the leaf nodes. These modifications allow us to extend our techniques in a straightforward manner to learn monitorable and concise specifications in STL.

4.9 Conclusion

This chapter presents a novel SMT-based algorithm for automatic learning of MTL formulas from observed finite system behaviours. The algorithm aims to synthesize concise formulas with low future-reach, making them suitable for efficient monitoring. Benchmark experiments demonstrate that our algorithm successfully synthesizes concise formulas based on commonly used MTL patterns.

Furthermore, we extend our algorithm to support the synthesis of separating formulas, which are commonly used in standard temporal logic inference. We provide complexity results for the corresponding decision problems. As a future direction, it would be interesting to establish lower bounds for these inference problems in the full fragments.

Another avenue for future research is to enhance the algorithm to efficiently and optimally handle STL. While we have presented a straightforward extension to accommodate STL in our framework, we believe that a comprehensive analysis, along with the introduction of efficient optimizations and heuristics, can lead to a significantly more efficient STL learning algorithm than the one proposed.

Part II

Verification: Parameter Synthesis for One-Counter Automata

Chapter 5

Parameter Synthesis for One-Counter Automata with Parametric Updates and Tests

In this chapter, we discuss the parameter synthesis problems for one-counter automata, a classical model of computation in formal methods and verification. We will show a relationship between these problems to first-order logic and establish several complexity results. We will also briefly introduce a study on an extension of these models with Parikh constraints that were inspired by the above-mentioned connection to logic. However, from the point of view of formal verification, we first discuss the background and motivation behind the problems and the model chosen based on the related works in the literature. The content presented in this chapter is based on the works in [117, 31].

5.1 Background

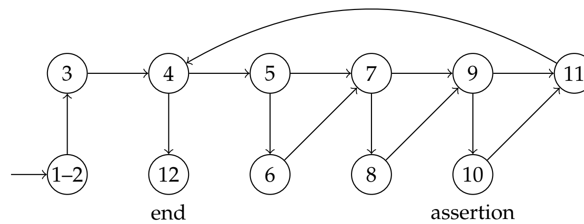
Our motivation for studying one-counter automata (OCA) with parameters stems from their usefulness in modelling programs whose control flow is determined by counter-variables. One such simple program is depicted in Figure 5.1.

```
1  def funprint(x):
2      i = 0
3      i += x
4      while i >= 0:
5          if i == 0:
6              print("Hello")
7          if i == 1:
8              print("world")
9          if i >= 2:
10             assert(False)
11         i -= 1
12     # end program
```

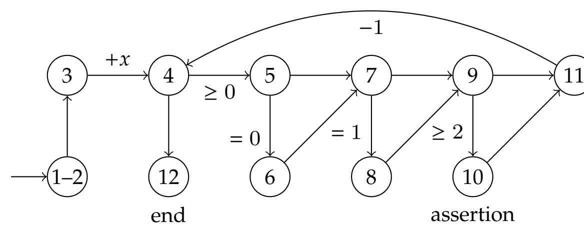
Figure 5.1: A program whose control flow is determined by the counter variable i

Indeed, the executions of such a program can be over-approximated by its control-flow graph (CFG) [1], where we encode every line of the program as a node, and the edges encode the reachability of one line of the program to the other. The CFG can be leveraged to get a *conservative response* to interesting questions about the program, such as: "Is there a value of x such that the false assertion is avoided?" Note that the CFG in Figure 5.2a abstracts away all variables and their values, and this introduces non-determinism. Hence, the question becomes: "Is it the case that all paths from the initial node avoid the one labelled with 10?" In this particular example, the abstraction is too *coarse*, and thus we obtain a false negative.

In such cases, the abstraction of the program should be refined [41]. A natural refinement of the CFG in this context is obtained by tracking the value of i (cf. program graphs in [8]) as illustrated in Figure 5.2b. This results in an OCA with parameters where there is a counter that can be updated or tested with parameters or integers along the edges. In these models, we can observe that: For $x \in \{0, 1\}$, it has no run that reaches the state labelled with 10. This is an instance of a *safety parameter synthesis problem* for which the answer is positive.



(a) CFG for program in Figure 5.1



(b) Extended CFG tracking variable i

Figure 5.2: At the top, the CFG with node labels corresponding to the source code line numbers; Below, the CFG extended by tracking the value of the variable i

In this chapter, we focus on the parameter synthesis problems for given OCA with parameters and do not consider the problem of obtaining such an OCA from a program (cf. [57]).

Counter automata [110] are classical models that extend finite-state automata with integer-valued counters. These have been shown to be useful in modelling complex systems, such as programs with lists and XML query evaluation algorithms [27, 36]. As previously mentioned, it is known that two counters suffice for counter automata to become Turing powerful, making all the interesting questions about them

undecidable [110]. In this chapter, we focus on a restriction of this model to one counter: OCA with integer-valued updates and tests. In particular, we study the universal parameter synthesis problems for this model, where we allow parameters to appear in the updates and the tests. Further note that the model we study has an asymmetric set of tests that can be applied to the counter: lower-bound tests and equality tests (both parametric and non-parametric). The primary reason for this is that adding upper-bound tests results in a model for which even the decidability of the (arguably simpler) existential reachability parameter synthesis problem is a long-standing open problem [30]. Namely, the resulting model corresponds to Ibarra's *simple programs* [80].

5.1.1 Related Works

The existential version of the parameter synthesis problems for OCA was considered by Göller et al. [67]. They ask whether there exist a valuation of the parameters and a run of the automaton which satisfies a given specification. In contrast, we consider the *universal* version of the problem where we quantify runs universally. This is required for the conservative-approximation use case described in the example above.

In both [67] and [23], the universal parameter synthesis problems for OCA with parameters were stated as open. Later, Lechner [93] gave an encoding for the complement of these problems into a one-alternation fragment of Presburger arithmetic with divisibility (PAD). Her encoding relies on the work by Haase et al. [73], which shows how to compute a linear-arithmetic representation of the reachability relation of OCA (see [99] for an implementation). In the same work, Haase et al. show that the same can be achieved for OCA with parameters using the divisibility predicate. In [93], Lechner goes on to consider the complexity of (validity of sentences in) the language corresponding to the one-alternation fragment her encoding targets. An earlier paper [29] by Bozga and Iosif argues that the fragment is decidable, and Lechner carefully repeats their argument while leveraging bounds on the bit-size of solutions of existential PAD formulas [95] to argue the complexity of the fragment is co2NEXP. For ω-regular properties given as a linear temporal logic (LTL) formula, her encoding is exponential in the formula, and thus it follows that the LTL synthesis problem is decidable and in 3NEXP.

5.1.2 Problems in the Literature

Recall that, *Presburger arithmetic* is the first-order theory of $\langle \mathbb{Z}, 0, 1, +, < \rangle$, as introduced in Section 2.2.2.1. *Presburger arithmetic with divisibility (PAD)* is the extension of PA obtained when we add a binary *divisibility predicate*. The resulting language in its full generality is undecidable [128]. In fact, a single quantifier alternation already allows encoding general multiplication, thus becoming undecidable [104]. However, the purely existential (Σ_0) and purely universal (Π_0) fragments have been shown to be decidable [18, 103].

The target of Lechner's encoding is $\forall\exists_R\mathrm{PAD}^+$, a subset of all sentences in the Π_1-fragment of PAD. Such sentences look as follows:
$\forall\vec{x}\exists\vec{y} \bigvee_{i\in I} \bigwedge_{j\in J_i} f_j(\vec{x}) \mid g_j(\vec{x},\vec{y}) \wedge \varphi_i(\vec{x},\vec{y})$ where φ is a quantifier-free PAD formula

without divisibility. Note that all divisibility constraints appear in positive form (hence the $^+$) and that, within divisibility constraints, the existentially-quantified variables y_i appear only on the right-hand side (hence the $\exists_R$). In [29], the authors give a quantifier-elimination procedure for sentences in a further restricted fragment we call the Bozga-Iosif-Lechner fragment (BIL) that is based on "symbolically applying" the generalized Chinese remainder theorem (CRT) [92]. Their procedure does not eliminate all quantifiers but rather yields a sentence in the Π_0-fragment of PAD. (Decidability of the BIL language would then follow from the result of Lipshitz [103].) Then, they *briefly argue* how the algorithm generalizes to $\forall\exists_R$PAD$^+$. There are two crucial problems in the argument from [29] that we have summarized here (and which were reproduced in Lechner's work):

- First, the quantifier-elimination procedure of Bozga and Iosif does not directly work for BIL. Indeed, not all BIL sentences satisfy the conditions required for the CRT to be applicable as used in their algorithm.
- Second, there is no way to generalize their algorithm to $\forall\exists_R$PAD$^+$ since the language is undecidable. Interestingly, undecidability follows directly from other results in [29, 93].

In Lechner's thesis [94], the result from [29] was stated as being under review. Correspondingly, the decidability of the universal parameter synthesis problems for OCA with parameters was only stated conditionally on $\forall\exists_R$PAD$^+$ being decidable.

5.1.3 Our Contribution

We fix the above-mentioned problems in the literature. Using developments from [29, 93], we first will argue that $\forall\exists_R$PAD$^+$ is undecidable. Then, we "fix" the definition of the BIL fragment by adding to it a necessary constraint so that the quantifier-elimination procedure from [29] works correctly. Using similar methods, we establish that the complexity of BIL is in co2NEXP. *To the best of our knowledge, BIL is the largest known decidable fragment of PAD in general.* Figure 5.3 illustrated the fragments of PAD and their decidability results.

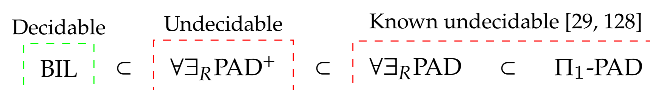


Figure 5.3: Syntactical fragments of PAD ordered w.r.t. their language (of sentences)

Then by careful analysis and reduction to BIL, we re-establish the decidability of universal parameter synthesis problems for various ω-regular specifications. To do so, we follow Lechner's original idea from [93] to encode them into $\forall\exists_R$PAD$^+$ sentences. However, to ensure we obtain a BIL sentence, several parts of her encoding have to be adapted. Finally, using the standard reduction from LTL to Büchi in literature, we establish that the problem is in 3NEXP for LTL specifications. Table 5.1 contains a summary of these complexity results.

	Lower bound	Upper bound
LTL	PSPACE-hard [134]	in 3NEXP (Thm. 10)
Reachability	coNP-hard (Thm 14, Chap 6)	in 2NEXP (Thm. 10)
Safety, Büchi, coBüchi	NP^{NP}-hard [93, 94]	

Table 5.1: Known and new complexity bounds for parameter synthesis problems

5.2 Outline of the Chapter

We introduce Presburger Arithmetic with Divisibility along with its various decidable and undecidable fragments in Section 5.3. Then, Section 5.4 introduces the one-counter automata model and the parameter synthesis problems we focus on in this chapter. We establish the complexity results of these problems in Section 5.5. Finally, we briefly introduce an extension of the model called Parikh One-counter Automata in Section 5.6.

5.3 Presburger Arithmetic with Divisibility

Presburger arithmetic (PA) is the first-order theory over $\langle \mathbb{Z}, 0, 1, +, < \rangle$ where $+$ and $<$ are the standard addition and ordering of integers. *Presburger arithmetic with divisibility (PAD)* is the extension of PA obtained when we add the binary divisibility predicate $|$, where for all $a, b \in \mathbb{Z}$ we have $a \mid b \iff \exists c \in \mathbb{Z} : b = ac$. In general, quantifier-free PAD formulas have the grammar:

$$\varphi ::= \varphi_1 \land \varphi_2 \mid \neg\varphi \mid f(\vec{x})\ P\ g(\vec{x}),$$

where P can be the order predicate $<$ or the divisibility predicate $|$, and f, g are linear polynomials. The *size* $|\varphi|$ of a PAD formula φ is defined by structural induction analogous to the size of a PA formula: For a linear polynomial $p(\vec{x})$ we define $|p(\vec{x})|$ as the number of symbols required to write it if the coefficients are given in binary. Then, we define $|\varphi_1 \land \varphi_2| \stackrel{\text{def}}{=} |\varphi_1| + |\varphi_2| + 1$, $|\neg\varphi| \stackrel{\text{def}}{=} |\exists x.\varphi| \stackrel{\text{def}}{=} |\varphi| + 1$, $|f(\vec{x})\ P\ g(\vec{x})| \stackrel{\text{def}}{=} |f(\vec{x})| + |g(\vec{x})| + 1$.

5.3.1 Allowing One Restricted Alternation

We define the language $\forall\exists_R$PAD of all PAD sentences allowing a universal quantification over some variables, followed by an existential quantification over variables that may not appear on the left-hand side of divisibility constraints. Formally, $\forall\exists_R$PAD is the set of all PAD sentences of the form: $\forall x_1 \ldots \forall x_n \exists y_1 \ldots \exists y_m \varphi(\vec{x}, \vec{y})$ where φ is a quantifier-free PAD formula and all its divisibility constraints are of the form $f(\vec{x}) \mid g(\vec{x}, \vec{y})$.

Positive-divisibility fragment. We denote by $\forall\exists_R\text{PAD}^+$ the subset of $\forall\exists_R$PAD sentences φ where the negation operator can only be applied to the order predicate $<$

and the only other Boolean operators allowed are conjunction and disjunction. In other words, $\forall\exists_R$PAD$^+$ is a restricted negation normal form in which divisibility predicates cannot be negated. Lechner showed in [93] that all $\forall\exists_R$PAD sentences can be translated into $\forall\exists_R$PAD$^+$ sentences.

Lemma 11 (Lechner's trick [93]) *For all φ_1 in $\forall\exists_R$PAD one can compute φ_2 in $\forall\exists_R$PAD$^+$ such that φ_1 is true if and only if φ_2 is true.*

Proof of Lemma 11. Consider a sentence Φ in $\forall\exists_R$PAD:

$$\forall x_1 \dots \forall x_n \exists y_1 \dots \exists y_m \varphi(\vec{x}, \vec{y}).$$

We observe Φ can always be brought into negation normal form so that negations are applied only to predicates [127]. Hence, it suffices to argue that we can remove negated divisibility predicates while staying within $\forall\exists_R$PAD.

The claim follows from the identity below since the newly introduced variables x', x'' are both existentially quantified and only appear on the right-hand side of divisibility constraints. For all $a, b \in \mathbb{Z}$ we have the following.

$$\neg(a \mid b) \iff (a = 0 \land b \neq 0) \lor \exists x' \exists x'' \Big(\big((b = x' + x'') \land (a \mid x') \land (0 < x'' < a)\big) \lor \big((b = -x' - x'') \land (a \mid x') \land (0 < x'' < -a)\big) \Big)$$

In other words, if $a = 0$ and $b \neq 0$, then $\neg(a \mid b)$. Further, if $a \neq 0$, there are integers $q, r \in \mathbb{Z}$ such that $b = qa + r$ and $0 < r < |a|$ if and only if $\neg(a \mid b)$. □

5.3.2 Undecidability of Both One-alternation Fragments

We will now prove that the language $\forall\exists_R$PAD$^+$ is undecidable; that is, to determine whether a given sentence from $\forall\exists_R$PAD$^+$ is true is an undecidable problem.

Theorem 6 *The language $\forall\exists_R$PAD$^+$ is undecidable.*

From Lemma 11, it follows that arguing $\forall\exists_R$PAD is undecidable suffices to prove the theorem. The latter was proven in [29]. More precisely, they show the complementary language is undecidable. Their argument consists in defining the least-common-multiple predicate, the squaring predicate, and, subsequently, integer multiplication. Undecidability thus follows from the MRDP theorem [106] which states that satisfiability for such equations (i.e. Hilbert's 10th problem) is undecidable. Hence, Theorem 6 is a direct consequence of the following result. We formalize and prove the undecidability result below:

Lemma 12 (From [29]) *The language $\forall\exists_R$PAD is undecidable.*

Proof of Lemma 12. We will show the language $\exists\forall_R$PAD of all sentences of the form $\neg\varphi$ such that $\varphi \in \forall\exists_R$PAD is undecidable.

We begin by recalling the definition of the $\mathrm{lcm}(\cdot,\cdot,\cdot)$ predicate. A common multiple of $a, b \in \mathbb{Z}$ is an integer $m \in \mathbb{Z}$ such that $a \mid m$ and $b \mid m$. Their least common multiple m is

minimal, that is, $m \mid m'$ for all common multiples m'. This leads to the following definition of $\mathrm{lcm}(a, b, m)$ for all $a, b, m \in \mathbb{Z}$.

$$\mathrm{lcm}(a, b, m) \iff \forall m' \left((a \mid m') \land (b \mid m')\right) \longleftrightarrow (m \mid m'))$$

Observe that the universally-quantified m' appears only on the right-hand side of the divisibility constraints. We thus have that $\exists\forall_R$PAD can be assumed to include a least-common-multiple predicate.[1] For convenience, we will write $\mathrm{lcm}(a, b) = m$ instead of $\mathrm{lcm}(a, b, m)$.

Now, once we have defined the $\mathrm{lcm}(\cdot, \cdot, \cdot)$ predicate, we can define the perfect square relation using the identity:

$$x > 0 \land x^2 = y \iff \mathrm{lcm}(x, x + 1) = y + x$$

and multiplication via:

$$4xy = (x + y)^2 - (x - y)^2.$$

Observe that we are now able to state Diophantine equations. Undecidability thus follows from the MRDP theorem [106], which states that satisfiability for such equations (i.e. Hilbert's 10th problem) is undecidable. □

5.3.3 The Bozga-Iosif-Lechner Fragment

In [29], Bozga and Iosif actually proved that determining the truth value of a given sentence from a fragment of $\forall\exists_R\mathrm{PAD}^+$ is decidable. Lechner provided a simplified argument and gave an improved complexity upper bound in [93]. While their generalization to the full $\forall\exists_R\mathrm{PAD}^+$ was based on a flawed argument, the decidable fragment, which we refer to as the Bozga-Iosif-Lechner fragment (or BIL for short), will be very useful in the rest of the chapter.

5.3.3.1 The Formulation

The Bozga-Iosif-Lechner (BIL) fragment is the set of all $\forall\exists_R\mathrm{PAD}^+$ sentences of the form:

$$\forall x_1 \dots \forall x_n \exists y_1 \dots \exists y_m (\bigvee_i x_i < 0) \lor \bigvee_{i \in I} \bigwedge_{j \in J_i} \left(f_j(\vec{x}) \mid g_j(\vec{x}, \vec{y}) \land f_j(\vec{x}) > 0\right) \land \varphi_i(\vec{x}) \land \vec{y} \geq \vec{0}$$

where $I, J_i \subseteq \mathbb{N}$ are all finite index sets, the f_j and g_j are linear polynomials and the $\varphi_i(\vec{x})$ are quantifier-free PA formulas over the variables $\vec{x}$. Note that, compared to $\forall\exists_R\mathrm{PAD}^+$, BIL sentences only constraint non-negative values of $\vec{x}$. (This technicality is necessary due to our second constraint below.) For readability, henceforth, we omit the disjunction that captures the case of x_i's being negative and only focus on the case where all x_i's take non-negative integer values. Additionally, it introduces the following three important constraints:

1. The $\vec{y}$ variables may only appear on the right-hand side of divisibility constraints.

[1] We remark that this definition of the least common multiple is oblivious to the sign of m, e.g. $\mathrm{lcm}(2, 3, -6)$ is true and $\mathrm{lcm}(a, b, m) \iff \mathrm{lcm}(a, b, -m)$ in general. This is not a problem since we can add $m \geq 0$ if desired.

2. All divisibility constraints $f_j(\vec{x}) \mid g_j(\vec{x}, \vec{y})$ are conjoined with $f_j(\vec{x}) > 0$.
3. The $\vec{y}$ variables are only allowed to take non-negative values.

It should be clear that the first constraint is necessary to avoid undecidability. Indeed, if the $\vec{y}$ variables were allowed in the PA formulas $\varphi_i(\vec{x})$, then we could circumvent the restrictions of where they appear in divisibilities by using equality constraints. The second constraint is similar in spirit. Note that if $a = 0$, then $a \mid b$ holds if and only if $b = 0$, so if the left-hand side of divisibility constraints is allowed to be 0, then we can encode PA formulas on $\vec{x}$ and $\vec{y}$ as before. Also, the latter (which was missing in [29, 93]) will streamline the application of the generalized Chinese remainder theorem in the algorithm described in the sequel.

5.3.3.2 The Flaw in the Previous Attempt

Now we identify a flow in the result of [29] that contains the flawed attempt of proving that $\forall\exists_R\mathrm{PAD}^+$ is decidable. First note that every $\forall\exists_R\mathrm{PAD}^+$ sentence φ can be assumed to be of the form $\Phi = \forall\vec{x}\varphi(\vec{x})$ with

$$\varphi(\vec{x}) = \exists y_1 \ldots \exists y_m \bigvee_{i\in I} \bigwedge_{j\in J_i} \big(f_j(\vec{x}) \mid g_j(\vec{x}, \vec{y})\big) \wedge \psi_i(\vec{x}, \vec{y})$$

where ψ_i are Presburger formulas with free variables $\vec{x}$ and $\vec{y}$. In their proposed algorithm, the first step claims that by substituting and renaming the existentially quantified variables, we can reduce φ to the following form:

$$\exists y_1 \ldots \exists y_m \bigvee_{i\in I} \bigwedge_{j\in J_i} \Big(f'_j(\vec{x}) \mid g'_j(\vec{x}, \vec{y})\Big) \wedge \psi'_i(\vec{x})$$

where the $\vec{y}$ variables can only appear on the right side of divisibility; intuitively, their algorithm proposes that we can remove all the existentially quantified variables from the Presburger constraints outside divisibility in the formula. Note that, intuitively, their process tries to reduce $\forall\exists_R\mathrm{PAD}^+$ to BIL. In the example below, we try to show that this is not the case, i.e., we start with a $\forall\exists_R\mathrm{PAD}^+$ formula and follow their proposed steps and show that it is not possible to remove all the existentially quantified variables[2].

[2]In the example, every step in intuitively simple and that is the reason we omit the full details of the algorithm in [29] and keep it at a high level in this chapter

$$\exists x_1 \exists x_2 (y \mid 5x_1 + 4x_2) \land (5x_1 + 6x_2 - y \leq 0) \land (5x_1 + 4x_2 - y \leq 0) \land (3y - 2x_2 \leq 0)$$

turning inequalities to equalities $\longrightarrow$

$$\exists \vec{x} \exists \vec{z} (y \mid 5x_1 + 4x_2) \land (5x_1 + 6x_2 - y + z_1 = 0) \land (5x_1 + 4x_2 - y + z_2 = 0) \land (3y - 2x_2 + z_3 = 0) \land (\vec{z} \geq 0)$$

removing x_1 $\downarrow$

$$\exists x_2 \exists \vec{z} (y \mid 2y - z_1 - 2x_2) \land (y - 2x_2 - z_1 + z_2 = 0) \land (3y - 2x_2 + z_3 = 0) \land (\vec{z} \geq 0)$$

removing x_2 $\longleftarrow$

$$\exists \vec{z} (y \mid y - z_2) \land (2y + z_1 - z_2 + z_3 = 0) \land (\vec{z} \geq 0)$$

Now the equation $(2y + z_1 - z_2 + z_3 = 0)$ cannot be reduced anymore as we cannot remove any of the $\vec{z}$ variables, and hence, in the end, we get existentially quantified variables outside divisibility. ∎

5.3.3.3 Decidability of BIL

In the rest of this section, we recall the decidability proof by Bozga and Iosif [29] and refine Lechner's analysis [93] to obtain the following complexity bound.

Theorem 7 *The BIL-fragment language is decidable in co2NEXP.*

The idea of the proof is as follows: We start from a BIL sentence. First, we use the *generalized Chinese remainder theorem* (CRT, for short) to replace all of the existentially quantified variables in it with a single universally quantified variable. We thus obtain a sentence in ∀PAD (i.e. the Π_0-fragment of PAD) and argue that the desired result follows from the bounds on the bit-size of satisfying assignments for existential PAD formulas [95].

Theorem 8 (Generalized Chinese remainder theorem [92]) *Let $m_i \in \mathbb{N}_{>0}$, $a_i, r_i \in \mathbb{Z}$ for $1 \leq i \leq n$. Then, there exists $x \in \mathbb{Z}$ such that $\bigwedge_{i=1}^{n} m_i \mid (a_i x - r_i)$ if and only if:*

$$\bigwedge_{1 \leq i,j \leq n} \gcd(a_i m_j, a_j m_i) \mid (a_i r_j - a_j r_i) \land \bigwedge_{i=1}^{n} \gcd(a_i, m_i) \mid r_i.$$

The solution for x is unique modulo $\operatorname{lcm}(m'_1, \ldots, m'_n)$, where $m'_i = \frac{m_i}{\gcd(a_i, m_i)}$.

From a BIL sentence, we apply the CRT to the rightmost existentially quantified variable and get a sentence with one less existentially quantified variable and with gcd-expressions. Observe that the second restriction we highlighted for the BIL

fragment (the conjunction with $f_j(\vec{x}) > 0$) is necessary for the correct application of the CRT. We will later argue that we can remove the gcd expressions to obtain a sentence in ∀PAD.

Example 6 *Consider the sentence:*

$$\forall x \exists y_1 \exists y_2 \bigvee_{i \in I} \bigwedge_{j \in J_i} \big(f_j(x) \mid g_j(x, \vec{y}) \land f_j(x) > 0\big) \land \varphi_i(x) \land \vec{y} \geq \vec{0}.$$

Let α_j denote the coefficient of y_2 in $g_j(x, \vec{y})$ and $r_j(x, y_1) \stackrel{def}{=} -(g_j(x, \vec{y}) - \alpha_j y_2)$. We can rewrite the above sentence as $\forall x \exists y_1 \bigvee_{i \in I} \psi_i(x, y_1) \land \varphi_i'(x) \land y_1 \geq 0$ where:

$$\psi_i(x, y_1) = \exists y_2 \bigwedge_{j \in J_i} (f_j(x) \mid (\alpha_j y_2 - r_j(x, y_1))) \land y_2 \geq 0, \text{ and}$$
$$\varphi_i'(x) = \varphi_i(x) \land \bigwedge_{j \in J_i} f_j(x) > 0.$$

Applying the CRT, $\psi_i(x, y_1)$ can equivalently be written as follows:

$$\bigwedge_{j,k \in J_i} \gcd(\alpha_k f_j(x), \alpha_j f_k(x)) \mid (\alpha_j r_k(x, y_1) - \alpha_k r_j(x, y_1)) \land \bigwedge_{j \in J_i} \gcd(\alpha_j, f_j(x)) \mid r_j(x, y_1).$$

Note that we have dropped the $y_2 \geq 0$ constraint without loss of generality since the CRT states that the set of solutions forms an arithmetic progression containing infinitely many positive (and negative) integers. This means the constraint will be trivially satisfied for any valuation of x and y_1 which satisfies $\psi_i(x, y_1) \land \varphi_i(x) \land y_1 \geq 0$ for some $i \in I$. Observe that y_1 only appears in polynomials on the right-hand side of divisibilities.

The process sketched in the example can be applied in general to BIL sentences sequentially starting from the rightmost quantified y_i. At each step, the size of the formula is at most squared. In what follows, it will be convenient to deal with a single polyadic gcd instead of nested binary ones. Thus, using associativity of gcd and pushing coefficients inwards — i.e. using the equivalence $a \cdot \gcd(x, y) \equiv \gcd(ax, ay)$ for $a \in \mathbb{N}$ — we finally obtain a sentence:

$$\forall x_1 \ldots \forall x_n \bigvee_{i \in I} \bigwedge_{j \in L_i} (\gcd(\{f_{j,k}'(\vec{x})\}_{k=1}^{K_j}) \mid g_j'(\vec{x})) \land \varphi_i'(\vec{x}) \tag{5.1}$$

where $|L_i|$, $|K_j|$, and the coefficients may all be doubly-exponential in the number m of removed variables due to iterated squaring.

Eliminating the gcd operator. In this next step, our goal is to obtain an ∀PAD sentence from Equation (5.1). Recall that ∀PAD "natively" allows for negated divisibility constraints. (That is, without having to encode them using Lechner's trick.) Hence, to remove expressions in terms of gcd from Equation (5.1), we can use the following identity:

$$\gcd(f_1(\vec{x}), \ldots, f_n(\vec{x})) \mid g(\vec{x}) \iff \forall d \left(\bigwedge_{i=1}^{n} d \mid f_i(\vec{x}) \right) \to d \mid g(\vec{x}).$$

This substitution results in a constant blowup of the size of the sentence. The above method gives us a sentence $\forall\vec{x}\forall d\psi(\vec{x}, d)$, where $\psi(\vec{x}, d)$ is a quantifier-free PAD formula. To summarize:

Lemma 13 *For any BIL sentence $\varphi = \forall x_1 \ldots \forall x_n \exists y_1 \ldots \exists y_m \bigvee_{i\in I} \varphi_i(\vec{x}, \vec{y})$ we can construct an ∀PAD sentence $\psi = \forall x_1 \ldots \forall x_n \forall d \bigvee_{i\in I} \psi_i(\vec{x}, d)$ such that: φ is true if and only if ψ is true and for all $i \in I$, $|\psi_i| \le |\varphi_i|^{2^m}$. The construction is realizable in time $O(|\varphi|^{2^m})$.*

To prove Theorem 7, the following small-model results for purely existential PAD formulas and BIL will be useful.

Theorem 9 ([95], Theorem 14) *Let $\varphi(x_1, \ldots, x_n)$ be a ∃PAD formula. If φ has a solution then it has a solution $(a_1, \ldots, a_n) \in \mathbb{Z}^n$ with the bitsize of each a_i bounded by $|\varphi|^{\mathrm{poly}(n)}$.*

Corollary 1 *Let $\forall x_1 \ldots \forall x_n \varphi(x_1, \ldots, x_n)$ be a BIL sentence. If $\neg\varphi$ has a solution then it has a solution $(a_1, \ldots, a_n) \in \mathbb{Z}^n$ with the bitsize of each a_i bounded by $|\varphi|^{2^m \mathrm{poly}(n+1)}$.*

Proof of Corollary 1. Using Lemma 13, we translate the BIL sentence to $\forall x_1 \ldots \forall x_n \forall d\psi(\vec{x}, d)$, where the latter is an ∀PAD sentence. Then, using Theorem 9, we get that the ∃PAD formula $\neg\psi(\vec{x}, d)$ admits a solution if and only if it has one with bit-size bounded by $|\psi|^{\mathrm{poly}(n+1)}$. Now, from Lemma 13 we have that $|\psi|$ is bounded by $|\varphi|^{2^m}$. Hence, we get that the bit-size of a the solution is bounded by: $|\varphi|^{2^m \mathrm{poly}(n+1)}$. □

We are now ready to prove the theorem.

Proof of Theorem 7. As in the proof of Corollary 1, we translate the BIL sentence to $\forall x_1 \ldots \forall x_n \forall d\psi(\vec{x}, d)$. Note that our algorithm thus far runs in time: $O\left(|\varphi|^{2^m}\right)$. By Corollary 1, if $\neg\psi(\vec{x}, d)$ has a solution, then it has one encodable in binary using a doubly exponential amount of bits with respect to the size of the input BIL sentence. The naive guess-and-check decision procedure applied to $\neg\psi(\vec{x}, d)$ gives us a $co2NEXP$ algorithm for BIL sentences. Indeed, after computing $\psi(\vec{x}, d)$ and guessing a valuation, checking it satisfies $\neg\psi$ takes polynomial time in the bit-size of the valuation and $|\psi|$, hence doubly exponential time in $|\varphi|$. □

5.4 Succinct One-Counter Automata with Parameters

In this section, we formally define OCA with parameters that extend finite-state automata with a single counter and a set of parameters against which the counter can be updated and tested. The concept and observations we introduce here are largely taken from [73] and the exposition in [94].

A *succinct parametric one-counter automaton (SOCAP)* is a tuple $\mathcal{A} = (Q, T, \Delta, X)$, where Q is a finite set of states, X is a finite set of parameters, $T \subseteq Q \times Q$ is a finite set of transitions and $\Delta : T \to Op$ is a function that associates an operation to every transition. The set $Op = CU \uplus PU \uplus ZT \uplus PT$ is the union of:

- *Constant Updates* $CU \stackrel{\text{def}}{=} \{+a : a \in \mathbb{Z}\}$,

- *Parametric Updates* $PU \stackrel{\text{def}}{=} \{+x, -x : x \in X\}$,
- *Zero Tests* $ZT \stackrel{\text{def}}{=} \{= 0\}$, and
- *Parametric Tests* $PT \stackrel{\text{def}}{=} \{= x, \geq x : x \in X\}$.

We denote by "$= 0$" or "$= x$" an *equality test* between the current value of the counter and zero or the value of x respectively and by "$\geq x$", a *lower-bound test* between the value of the counter and the value of x.

A *valuation* $V : X \longrightarrow \mathbb{N}$ assigns to every parameter a natural number. We assume CU are succinctly encoded in binary, hence the S in SOCAP. We omit "parametric" if $X = \emptyset$ (and call the non-parametric model as 'SOCA') and often write $q \xrightarrow{op} q'$ to denote $\Delta(q, q') = op$.

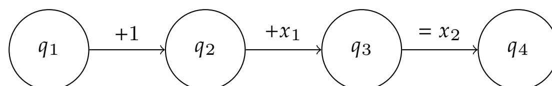


Figure 5.4: A one-counter automaton with parameters

Example 7 *Figure 5.4 demonstrates a one-counter automaton with parameters $X = \{x_1, x_2\}$ comprising a constant update $+1$ (from q_1 to q_2), a parametric update $+x_1$ (from State q_2 to State q_3) and a parametric test $= x_2$ (from q_3 to q_4).*

A *configuration* is a pair (q, c) where $q \in Q$ and $c \in \mathbb{N}$ is the *counter value*. Given a valuation $V : X \longrightarrow \mathbb{N}$ and a configuration (q_0, c_0), a *V-run from* (q_0, c_0) is a sequence $\rho = (q_0, c_0)(q_1, c_1) \ldots$ such that for all $i \geq 0$ the following hold: $q_i \xrightarrow{op_{i+1}} q_{i+1}$; $c_i = 0$, $c_i = V(x)$, and $c_i \geq V(x)$, if $\Delta(q_i, q_{i+1})$ is "$= 0$", "$= x$", and "$\geq x$", respectively; and c_{i+1} is obtained from c_i based on the counter operations. That is,

$$c_{i+1} = \begin{cases} c_i & \text{if } \Delta(q_i, q_{i+1}) \in (ZT \cup PT) \\ c_i + a & \text{if } \Delta(q_i, q_{i+1}) = +a \\ c_i + SV(x) & \text{if } \Delta(q_i, q_{i+1}) = Sx. \end{cases}$$

We say ρ *reaches* a state $q_f \in Q$ if there exists $j \in \mathbb{N}$, such that $q_j = q_f$. Also, ρ reaches or visits a set of states $F \subseteq Q$ iff ρ reaches a state $q_f \in F$. If V is clear from the context, we just write run instead of V-run. Note that, by definition, the counter-value is not allowed to go below zero in a valid run.

Convenient Syntactic Transformations. It is interesting to note that one can simulate any parametric test (equality or lower-bound) with only parametric updates and zero tests in a SOCAP. In particular, a transition labelled $\geq x$ can be simulated using two consecutive transitions labelled by $-x$ and $+x$. Here, we use the fact that the counter value can never go below zero in a one-counter automaton. Similarly, we can simulate equality tests of the form $= x$ by three consecutive transitions labelled with $-x$, $= 0$ and $+x$, respectively (similar to equality tests with constant). For example, in Figure 5.4, we can introduce two new states q_5 and q_6 and simulate the test $q_3 \xrightarrow{=x_2} q_4$ as

$q_3 \xrightarrow{-x_2} q_5 \xrightarrow{=0} q_6 \xrightarrow{+x_2} q_4$. One can also include constant lower-bound tests in the model as they can be simulated using constant updates and zero tests similarly. Note that upper-bound tests cannot be simulated in the above-mentioned ways. This is also natural as parameter synthesis problems for SOCAP with upper-bound tests correspond to a long-standing open problem, namely Ibarra's simple programs [80, 30].

The *underlying (directed) graph* of $\mathcal{A}$ is $G_{\mathcal{A}} = (Q, T)$. A V-run $\rho = (q_0, c_0)(q_1, c_1)\ldots$ in $\mathcal{A}$ *induces* a path $\pi = q_0 q_1 \ldots$ in $G_{\mathcal{A}}$. We assign weights to $G_{\mathcal{A}}$ as follows: For $t \in T$, weight(t) is 0 if $\Delta(t) \in ZT \cup PT$; a if $\Delta(t) = +a$; and $S \cdot V(x)$ if $\Delta(t) = Sx$. We extend the weight function to finite paths in a natural way. Namely, we set $\text{weight}(q_0 \ldots q_n) \stackrel{\text{def}}{=} \sum_{i=0}^{n-1} \text{weight}(q_i, q_{i+1})$.

5.4.1 Universal Parameter Synthesis problems for SOCAP

In this section, we introduce the universal parameter synthesis problems formally. Given a SOCAP $\mathcal{A}$, a specification p and a starting configuration (q_0, c), the problem asks whether there exists a valuation V such that all infinite V-runs from (q_0, c) satisfy p. Without loss of generality, we assume that the starting counter value $c = 0$, as we can always add an initial state with a $+c$ labelled transition to the original starting state q_0.

As specifications, we focus on LTL and some ω-regular properties that we describe below. Given a set of target states $F \subseteq Q$ and an infinite run $\rho = (q_0, c_0)(q_1, c_1)\ldots$, we say ρ satisfies:

- the *reachability* condition if $q_i \in F$ for some $i \in \mathbb{N}$;
- the *Büchi* condition if $q_i \in F$ for infinitely many $i \in \mathbb{N}$;
- the *coBüchi* condition if $q_i \in F$ for finitely many $i \in \mathbb{N}$ only;
- the *safety* condition if $q_i \notin F$ for all $i \in \mathbb{N}$;
- the *linear temporal logic* (LTL) formula φ over a set of atomic propositions $\mathcal{P}$ — and with respect to a labelling function $f : Q \to 2^{\mathcal{P}}$ — if $f(q_0)f(q_1)\ldots \models \varphi$.

Note that the existential version of these parameter synthesis problems have been discussed in [73], where instead of quantifying over all infinite runs, they ask for the existence of one possible run as a witness. As in this chapter, we focus on universal versions of these problems; we refer to them as p parameter synthesis problems (for specification of class p) unless otherwise mentioned. For example, if the specification is an LTL formula, we call it the LTL parameter synthesis problem. We decompose the above parameter synthesis problems into reachability sub-problems. Recall the concept of path-flow from Section 2.3.1 that establishes a connection between the reachability (witnesses) of graphs and their flows. We extensively use path-flow as a tool for our encoding in the following section.

5.5 Complexity of the Parameter Synthesis Problems

In this section, we prove that all the above-mentioned parameter synthesis problems are decidable. More precisely, we establish the following complexity upper bounds.

Theorem 10 *The reachability, Büchi, coBüchi, and safety synthesis problems for succinct one-counter automata with parameters are all decidable in 2NEXP. The LTL parameter synthesis problem is decidable in 3NEXP.*

The idea is as follows: we first focus on the coBüchi synthesis problem and reduce its complement to the truth value of a BIL sentence. To do so, we follow Lechner's encoding of the complement of the Büchi synthesis problem into $\forall\exists_R \mathrm{PAD}^+$ [93]. The encoding heavily relies on an encoding for (existential) reachability from [73]. We take extra care to obtain a BIL sentence instead of an $\forall\exists_R \mathrm{PAD}^+$ one as Lechner originally did.

The upper bound for the other synthesis problems stems from a reduction we show in the following subsection, where we reduce to the coBüchi one in polynomial time. The corresponding bounds thus follow from the one for coBüchi synthesis.

5.5.1 Reduction to coBüchi Synthesis Problem

In this section, we reduce the parameter synthesis problems for all the specifications to the coBüchi parameter synthesis problem for SOCAP. We formalize this as follows:

Lemma 14 *The reachability, safety and the Büchi synthesis problems can be reduced to the coBüchi synthesis problem in polynomial time. The LTL synthesis problem can be reduced to the coBüchi synthesis problem in exponential time.*

Proof. First, we show that the safety parameter synthesis problem can be reduced to the Büchi one by construction. Then, we reduce the Büchi synthesis problem to the coBüchi one. Next, we explain how the reachability synthesis problem can be reduced to the coBüchi one using similar ideas. Finally, we show an exponential reduction from the LTL synthesis problem to the coBüchi one using the connection between LTL and Büchi automata. The overview of the reductions is as follows:

Safety to Büchi. Consider a SOCAP $\mathcal{A} = (Q, T, \Delta, X)$, an initial configuration (q_0, c_0) and the set of target states F. Recall that the safety parameter synthesis problem for $\mathcal{A}$ asks if there exists a valuation of X such that all runs avoid F. We construct an automaton $\mathcal{B} = (Q', T', \Delta', X)$ which is disjoint union of two copies of $\mathcal{A}$: $\mathcal{B} \stackrel{\text{def}}{=} \mathcal{A}_1 \uplus \mathcal{A}_2$. We denote the states of $\mathcal{A}_1$ as Q_1 and states of $\mathcal{A}_2$ as Q_2, and the set of target states in $\mathcal{B}$ as F'. We will reduce the safety synthesis of $\mathcal{A}$ to Büchi synthesis of $\mathcal{B}$. Recall that the Büchi parameter synthesis problem for $\mathcal{B}$ asks if there exists a valuation of X such that all runs visit F' infinitely many times.

We take the initial configuration as (q_1^{in}, c_0) in $\mathcal{B}$ where $q_1^{\text{in}} \in Q_1$ is the copy of q_0 in $\mathcal{A}_1$. We "force" a move from the first copy to the second one via the target states (only), and

there is no way to come back to the first copy once we move to the second one. Formally, for every transition $(u, v) \in T$ such that $u \notin F$, we have $(u_1, v_1), (u_2, v_2) \in T'$ where $u_i, v_i \in Q_i$. For the transitions $(s, t) \in T$ such that $s \in F$, we have $(s_1, t_2), (s_2, t_2) \in T'$ where $s_i, t_i \in Q_i$. For all states $q \in Q_1$ and $q' \in Q_2$, $q \in F'$ and $q' \notin F'$.

Note that, for all valuations, there is an infinite run in $\mathcal{A}$ that visits F if and only if in $\mathcal{B}$ the corresponding run moves to $\mathcal{A}_2$ (and never comes back to the first copy) if and only if it visits F' only finitely many times. Hence, the answer to the safety synthesis problem in $\mathcal{A}$ is false if and only if the answer to the Büchi synthesis problem is false in $\mathcal{B}$.

Büchi to coBüchi. Consider the SOCAP $\mathcal{A}$ as defined above. Recall that the Büchi parameter synthesis for $\mathcal{A}$ asks if there exists a valuation of X such that all runs of $\mathcal{A}$ visits F infinitely many times. We will again construct another SOCAP $\mathcal{B} = (Q', T', \Delta', X)$ with a target set of states F' as above. Here, the construction of the automaton $\mathcal{B}$ is a bit different. Here we still construct $\mathcal{B}$ as a disjoint union of two copies of $\mathcal{A}$, but we remove the states in F from the copy $\mathcal{A}_2$. Also, for every $(u, v) \in T$, we have $(u_1, v_1), (u_1, v_2), (u_2, v_2) \in T'$. (Note that if $v \in F$ then $(u_1, v_2), (u_2, v_2) \notin T'$ as v_2 does not exist.) We set $F' \stackrel{\text{def}}{=} Q_2$. Recall that the coBüchi parameter synthesis problem for $\mathcal{B}$ asks if there exists a valuation of X such that all runs visit F' only finitely many times.

Now, for all valuations, there is an infinite run ρ in $\mathcal{A}$ that visits F only finitely many times if and only if there is an infinite run in $\mathcal{B}$ that follows ρ within $\mathcal{A}_1$ until it last visits a state from F and then moves to $\mathcal{A}_2$ so that it visits states from F' infinitely often. Hence, the answer to the Büchi synthesis problem in $\mathcal{A}$ is negative if and only if it is negative for the coBüchi synthesis problem in $\mathcal{B}$.

Reach to coBüchi. The reduction from the reachability parameter synthesis to the coBüchi one can be obtained following a similar construction as the one showed in the reduction from safety to Büchi. Intuitively, the idea is to reverse the target and non-target states in the construction, and the explanation is then straightforward following the similar explanation from the reduction from safety to Büchi.

LTL to coBüchi. Recall Theorem 2 from Section 2.3.2, which states that every LTL formula φ over a set of propositions $\mathcal{P}$ can be translated to a generalised Büchi automaton $\mathcal{A}$ of at most exponential size such that φ and $\mathcal{A}$ accept the same language. Recall also Lemma 1 from the same section, which states that every generalised Büchi automaton is equivalent to a Büchi automaton of polynomial size. Given an instance of LTL parameter synthesis problem for SOCAP $\mathcal{A}$ and formula φ, we can first create the exponential size Büchi automaton $\mathcal{B}_\varphi$ which accepts the same language as φ. Then it is easy to show that the LTL parameter synthesis problem for SOCAP $\mathcal{A}$ is positive if and only if the answer to the coBüchi parameter synthesis problem for SOCAP $\mathcal{A} \times \mathcal{B}_\varphi$ is positive, where $\mathcal{A} \times \mathcal{B}_\varphi$ is the product automaton. The details of the proof can be found in [94]. □

5.5.2 Encoding coBüchi Parameter Synthesis to BIL

Now the cornerstone of proving Theorem 10 is to show the decidability of coBüchi parameter synthesis problems. We will focus on a SOCAP $\mathcal{A} = (Q, T, \Delta, X)$ with $X = \{x_1, \ldots, x_n\}$ and often write $\vec{x}$ for $(x_1, \ldots, x_n)$. First, we illustrate with an example how reachability relation in a SOCAP can be seen as a combination of linear equalities, inequalities and divisibilities.

Example 8 *Consider a SOCAP $\mathcal{A}$ as illustrated in Figure 5.5. We want to determine if $(t, 0)$ is reachable from $(s, 0)$ in $\mathcal{A}$.*

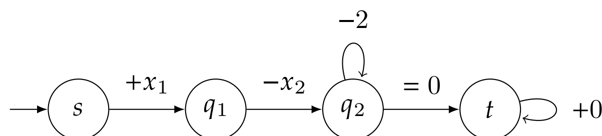


Figure 5.5: An example of SOCAP

Recall that the counter value always remains non-negative in a valid run. Hence, it is easy to see that the above reachability query can be encoded as the satisfiability query of the following formula:

$$\exists k\; x_1 \geq 0 \wedge x_1 \geq x_2 \wedge x_1 - x_2 - 2k = 0$$

The first two inequalities encode that the counter value remains non-negative after traversing the edges (s, q_1) and (q_1, q_2), respectively. The value of k determines how many times the run traverses the self-loop over q_3 such that $x_1 - x_2 - 2k = 0$ denotes the fact that the zero-test of (q_3, t) can be taken successfully. Then, the above formula can be written using divisibility as follows:

$$x_1 \geq 0 \wedge x_1 \geq x_2 \wedge 2|(x_1 - x_2)$$

Precisely, we will show a reduction from the complement of the coBüchi synthesis problem to the truth value of a BIL sentence that will give us the desired decidability result. To do this, we will use the following lemma that shows that the existence of a V-run from (q, c) to (q', c') can be encoded in a formula with a particular shape which is almost "BIL".

Lemma 15 *Given states q, q', one can construct in deterministic exponential time in $|\mathcal{A}|$ a PAD formula: $\varphi^{(q,q')}_{\text{reach}}(\vec{x}, a, b) = \exists \vec{y} \bigvee_{i \in I} \varphi_i(\vec{x}, \vec{y}) \wedge \psi_i(\vec{y}, a, b) \wedge \vec{y} \geq \vec{0}$ such that $\forall \vec{x} \exists \vec{y} \bigvee_{i \in I} \varphi_i(\vec{x}, \vec{y}) \wedge \vec{y} \geq 0$ is a BIL sentence, the $\psi_i(\vec{y}, a, b)$ are quantifier-free PA formulas, and additionally:*

- *a valuation V of $X \cup \{a, b\}$ satisfies $\varphi^{(q,q')}_{\text{reach}}$ iff there is a V-run from $(q, V(a))$ to $(q', V(b))$;*
- *the bit-size of constants in $\varphi^{(q,q')}_{\text{reach}}$ is of polynomial size in $|\mathcal{A}|$;*
- *$|\varphi^{(q,q')}_{\text{reach}}|$ is at most exponential with respect to $|\mathcal{A}|$; and*

- *the number of $\vec{y}$ variables is polynomial with respect to $|\mathcal{A}|$.*

Before we prove Lemma 15, we first show how we can prove Theorem 10 using this lemma.

First, we will argue that $\forall\vec{x}\exists a\exists b\varphi^{(q,q')}_{\text{reach}}(\vec{x},a,b)$ can be transformed into an equivalent BIL sentence. Note that for this to be the case, it suffices to remove the $\psi_i(\vec{y},a,b)$ subformulas. Intuitively, since these are quantifier-free PA formulas, their set of satisfying valuations is *semi-linear* (see, for instance, [72]). Our intention is to remove the $\psi_i(\vec{y},a,b)$ and replace the occurrences of $\vec{y},a,b$ in the rest of $\varphi^{(q,q')}_{\text{reach}}(\vec{x},a,b)$ with linear polynomials "generating" their set of solutions. We will use an *affine change of (existentially quantified) variables* to remove the ψ_i subformulas. This is formalized below.

Affine change of variables. Let $\vec{A} \in \mathbb{Z}^{m\times n}$ be an integer matrix of size $m \times n$ of rank r, and $\vec{b} \in \mathbb{Z}^m$. Let $\vec{C} \in \mathbb{Z}^{p\times n}$ be an integer matrix of size $p \times n$ such that $\begin{pmatrix}\vec{A}\\ \vec{C}\end{pmatrix}$ has rank s, and $\vec{d} \in \mathbb{Z}^p$. We write μ for the maximum absolute value of an $(s-1)\times(s-1)$ or $s \times s$ sub-determinant of the matrix $\begin{pmatrix}\vec{A} & \vec{b}\\ \vec{C} & \vec{d}\end{pmatrix}$ that incorporates at least r rows from $\begin{pmatrix}\vec{A} & \vec{b}\end{pmatrix}$.

Theorem 11 (From [139]) *Given integer matrices $\vec{A} \in \mathbb{Z}^{m\times n}$ and $\vec{C} \in \mathbb{Z}^{p\times n}$, integer vectors $\vec{b} \in \mathbb{Z}^m$ and $\vec{d} \in \mathbb{Z}^p$, and μ defined as above, there exists a finite set I, a collection of $n \times (n-r)$ matrices $\vec{E}^{(i)}$, and $n \times 1$ vectors $\vec{u}^{(i)}$, indexed by $i \in I$, all with integer entries bounded by $(n+1)\mu$ such that: $\{\vec{x} \in \mathbb{Z}^n : \vec{A}\vec{x} = \vec{b} \wedge \vec{C}\vec{x} \geq \vec{d}\} = \bigcup_{i\in I}\{\vec{E}^{(i)}\vec{y} + \vec{u}^{(i)} : \vec{y} \in \mathbb{Z}^{n-r}, \vec{y} \geq \vec{0}\}$.*

We are now ready to prove Theorem 10. In particular, using Lemma 15, we show that the complement of the coBüchi parameter synthesis problem can be encoded into BIL.

Proof of Theorem 10. Recall that the complement of the coBüchi synthesis problem asks: given a SOCAP $\mathcal{A}$ with parameters X, for all valuations, does there exist an infinite run from a given configuration $(q,0)$, that visits the target set F infinitely many times. Without loss of generality, we assume that the automaton has no parametric tests as they can be simulated using parametric updates and zero tests, as argued before.

The idea is to check if there exists a reachable "pumpable cycle" containing one of the target states. A pumpable cycle intuitively refers to a cycle that can be taken many times to reach any arbitrary counter value. Formally, given the starting configuration $(q,0)$, we want to check if we can reach a configuration (q_f,k), where $q_f \in F$ and $k \geq 0$ and then we want to reach q_f again via a pumpable cycle. This means that starting from (q_f,k), we reach the configuration (q_f,k) again, or we reach a configuration (q_f,k') with $k' \geq k$ without using zero-test transitions (this ensures that the cycle is pumpable). Note that reachability while avoiding zero tests is the same as reachability in the sub-automaton obtained after deleting all the zero-test transitions. We write $\varphi_{\text{reach-nt}}$ for the φ_{reach} formula constructed for that sub-automaton as per Lemma 15. The above constraints about the existence of a pumpable cycle can be encoded as a formula

$$\varphi_{\text{Büchi}}(\vec{x}) = \exists k\exists k' \bigvee_{q_f\in F} \zeta(\vec{x},k,k')$$

where the subformula ζ is of the form:

$$(k \leq k') \wedge \varphi_{\mathrm{reach}}^{(q,q_f)}(\vec{x}, 0, k) \wedge \left(\varphi_{\mathrm{reach-nt}}^{(q_f,q_f)}(\vec{x}, k, k') \vee \varphi_{\mathrm{reach}}^{(q_f,q_f)}(\vec{x}, k, k)\right).$$

Finally, the formula $\varphi_{\text{Büchi}}(\vec{x})$ will look as follows:

$$\exists \vec{y} \exists k \exists k' \bigvee_{i \in I} \bigwedge_{j \in J_i} \left(f_j(\vec{x}) \mid g_j(\vec{x}, \vec{y})\right) \wedge \varphi_i(\vec{x}) \wedge \psi_i(\vec{y}, k, k') \wedge \vec{y} \geq \vec{0}$$

where, by Lemma 15, the $\varphi_i(\vec{x})$ are quantifier-free PA formulas over $\vec{x}$ constructed by grouping all the quantifier-free PA formulas over $\vec{x}$. Similarly, we can construct $\psi_i(\vec{y}, k, k')$ by grouping all the quantifier free formulas over $\vec{y}$, k and k'. Now, we use the affine change of variables (see Theorem 11) to remove the formulas $\psi_i(\vec{y}, k, k')$. Technically, the free variables from the subformulas ψ_i will be replaced in all other subformulas by linear polynomials on newly introduced variables $\vec{z}$. Hence, the final formula $\varphi_{\text{Büchi}}(\vec{x})$ becomes:

$$\exists \vec{z} \bigvee_{i \in I'} \bigwedge_{j \in J_i} \left(f_j(\vec{x}) \mid g_j(\vec{x}, \vec{z})\right) \wedge \varphi_i(\vec{x}) \wedge \vec{z} \geq \vec{0}.$$

Note that, after using the affine change of variables, the number of $\vec{z}$ variables are bounded by the number of old existentially quantified variables $(\vec{y}, k, k')$. However, we have introduced exponentially many new disjuncts.[3]

By construction, for a valuation V there is an infinite V-run in $\mathcal{A}$ from $(q, 0)$ that visits the target states infinitely often iff $\varphi_{\text{Büchi}}(V(\vec{x}))$ is true. Hence, $\forall \vec{x}(\vec{x} < 0 \vee \varphi_{\text{Büchi}}(\vec{x}))$ precisely encodes the complement of the coBüchi parameter synthesis problem. Also, note that it is a BIL sentence since the subformulas (and, in particular, the divisibility constraints) come from our usage of Lemma 15. Now, the number of $\vec{z}$ variables, say m, is bounded by the number of $\vec{y}$ variables before the affine change of variables which is polynomial with respect to $|\mathcal{A}|$ from Lemma 15. Also, the bit-size of the constants in $\varphi_{\text{Büchi}}$ is polynomial in $|\mathcal{A}|$ though the size of the formula is exponential in $|\mathcal{A}|$. Now, using Lemma 13, we construct an $\forall$PAD sentence $\forall \vec{x} \forall d \psi(\vec{x}, d)$ from $\forall \vec{x}(\vec{x} < 0 \vee \varphi_{\text{Büchi}}(\vec{x}))$. By Corollary 1, $\neg\psi$ admits a solution of bit-size bounded by: $\exp(\ln(|\varphi_{\text{Büchi}}|) 2^m \mathrm{poly}(n+1)) = \exp(|\mathcal{A}| \cdot 2^{\mathrm{poly}(|\mathcal{A}|)} \mathrm{poly}(n+1))$, which is doubly exponential in the size of $|\mathcal{A}|$. As in the proof of Theorem 7, a guess-and-check algorithm for $\neg\psi$ gives us the desired 2NEXP complexity result for the coBüchi parameter synthesis problem. By Lemma 14, the other ω-regular parameter synthesis problems have the same complexity, and LTL parameter synthesis problem has one exponential top of it, i.e., in 3NEXP. □

Now the main goal of the chapter boils down to the proof of Lemma 15. Our proof relies on a symbolic representation of computations without zero tests in terms of *reachability certificates* developed in [73].

[3] Indeed, because of the bounds on the entries of the matrices and vectors, the cardinality of the set I is exponentially bounded.

5.5.2.1 Reachability Certificates

We presently recall the notion of reachability certificates from [73]. Fix a SOCAP $\mathcal{A}$ and a valuation V. A flow f in $G_{\mathcal{A}}$ is a *reachability certificate* for two configurations $(q, c), (q', c')$ in $\mathcal{A}$ if there is a V-run from (q, c) to (q', c') that induces a path π such that $f = f_\pi$ and one of the following holds:

- *(type 1)* f has no positive-weight cycles,
- *(type 2)* f has no negative-weight cycles, or
- *(type 3)* f has a positive-weight cycle that can be taken from (q, c) and a negative-weight cycle that can be taken to (q', c').

In the sequel, we will encode the conditions from the following lemma into a PAD formula so as to accommodate parameters. Intuitively, the proposition states that there is a run from (q, c) to (q', c') if and only if there is one of a special form: a decreasing prefix (type 1), a positive cycle leading to a plateau followed by a negative cycle (type 3), and an increasing suffix (type 2). Each one of the three sub-runs could, in fact, be an empty run.

Proposition 1 ([70], Lemma 4.1.14) *If (q', c') is reachable from (q, c) in a SOCAP with $X = \emptyset$ and without zero tests, then there is a run $\rho = \rho_1\rho_2\rho_3$ from (q, c) to (q', c'), where ρ_1, ρ_2, ρ_3, each has a polynomial-size reachability certificate of type 1, 3 and 2, respectively.*

Next, we show the idea of encoding these certificates in our desired logical fragment.

5.5.2.2 Encoding the Reachability Certificates

Now, we recall the encoding for the reachability certificates proposed by Lechner [93, 94]. Then, we highlight the changes necessary to obtain the required type of formula.

Consider a q_0–q_f path-flow f. To make sure that there is a path π such that $f = f_\pi$ lifts to a V-run from (q_0, c_0) to (q_f, c_f), we need to complement Proposition 1 with some way of ensuring the run does not reach negative counter values (by definition of SOCAP). Note that the existence of a path-flow is not enough to ensure that there exists a corresponding valid run in SOCAP. We demonstrate this with the following example.

Example 9 *Consider the SOCA presented in Figure 5.6. The figure also contains a path-flow f for the underlying weighted graph. The value of flow function f for each edge is described within braces. The operation associated with each edge is then mentioned, separated by a comma. For example, $f(s, q_1) = 1$ and $\Delta(s, q_1) = +5$. Note that the f is a valid path-flow by definition. Essentially the path that corresponds to the path-flow f is as follows:*

$$s \to (q_1 \to q_3 \to q_4)^2 \to t,$$

i.e., the path travels from s to q_1 and then traverse the loop $q_1q_3q_4$ twice then reaches t. Note that although this ensures that t is reachable from s in the underlying weighted graph, it does not

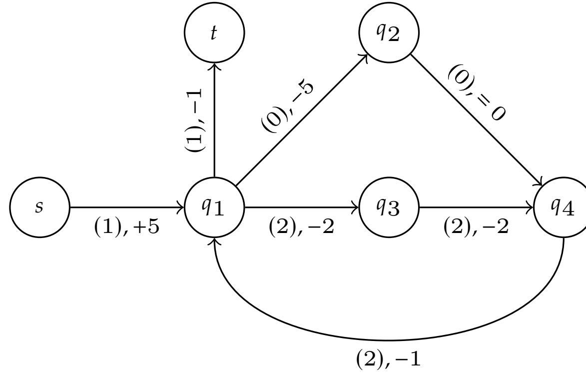


Figure 5.6: Existence of a path-flow is not enough as a witness of a run in SOCAP

ensure that t is reachable from $(s, 0)$ *in the SOCA. This is because after taking the loop* $q_1 q_3 q_4$ *once, the counter value reaches zero, and the transition* $q_1 \xrightarrow{-2} q_3$ *cannot be taken. This shows that the existence of a valid path-flow is not enough as a witness for the existence of a valid run in SOCA.*

Hence, the encoding of the certificates into PAD formulas from [73] relies on the following notion of *path-flow decomposition*.

Path-flow decomposition. Recall the definition of path-flow from Section 5.4.1. A decomposition of a q_0–q_f path-flow f consists of two parts. First, it includes an enumeration $q_1, \ldots, q_n$ of the set $\{q \in Q \mid f(p, q) > 0 \text{ for some } p \in Q\}$ of vertices in the subgraph induced by the support of f. Second, it includes a sequence $f_0, f_1, \ldots, f_{n-1}$ of flows such that:

- f_i is a path-flow from q_i to q_{i+1},
- $f(t) = \sum_{i=0}^{n-1} f_i(t)$ for all $t \in T$, and
- $f_j(p, q_i) = 0$ for all $i \leq j$ and all $(p, q_i) \in T$.

Each path-flow corresponds to an infix of the V-run, which ends at q_i and such that the suffix of the V-run henceforth never again reaches q_i. Furthermore, the original path is obtained as the concatenation of the individual paths.

Decomposing the path-flow into $q_0, \ldots, q_n$ and $f_0, \ldots, f_{n-1}$ allows Haase et al. [73] to state a linear number (with respect to $|\mathcal{A}|$) of constraints which suffice for this to hold in a specific situation: If f contains no positive-weight cycles, then every time the run reaches a state q from q itself, the value of the counter cannot have increased. Hence, for every state q, amongst all prefixes of π which end at q, the longest one must have minimal weight. This means the sufficient constraints are:

$$\bigwedge_{m=0}^{n-1} \sum_{i=0}^{m} \text{weight}(f_i) \geq -c_0. \tag{5.2}$$

That is, we verify the counter value is non-negative the last time each state is reached. This same idea also applies when a path-flow f contains no negative-weight cycles. However, in this case, it suffices to verify the non-negativity of the counter value the first time each state is visited. Instead of adapting our notion of decomposition, we follow [73] and use path-flows in the skew transpose of the induced graph of $\mathcal{A}$ as certificates for such scenarios.

In the sequel, we will focus on how to encode the existence of a path-flow f of every type into a PAD formula with $\vec{x}$ as free variables and with the properties required to prove Lemma 15.

For the detailed encoding of the three types of reachability certificates, we refer the readers to [94]. Before stating the precise technical lemma from the work, we first give an intuitive idea of how to encode type 1 certificates. The idea behind the encoding of the other two certificates is similar with some technical changes.

Let $\vec{f}$ be variables $(f_1, f_2, \dots)$ and q, q' states. The formula $\Phi_1^{(q,q')}(\vec{x}, a, b)$ we construct has the following form:

$$\bigvee_S \bigvee_D \exists \vec{f} \left(\varphi_{\text{flow}}(\vec{f}) \wedge \varphi_{\text{weight}}(\vec{f}, \vec{x}, a, b) \wedge \varphi_{\text{nopos}}(\vec{x}) \right), \tag{5.3}$$

where the subformulas are as follows:

- φ_{flow} is a quantifier-free Presburger arithmetic (PA) formula over $\vec{f}$ encoding the flow constraints. For any fixed flow decomposition, φ_{flow} has polynomial size and it encodes the constraints of flow conservation and Euler's theorem (Theorem 1);
- φ_{weight} is a quantifier-free PA formula over $\vec{f}, \vec{x}, a$ and b encoding the relation between the first and last configurations $(q, V(a)), (q', V(b))$ of the run, i.e. $b = \text{weight}(f) + a$, as well as the constraint from Equation (5.2);
- φ_{nopos} is an (exponential size) quantifier-free PA formula over $\vec{x}$ encoding the fact that f has no positive cycles. This can be achieved by enumerating all (exponentially many) simple cycles and the constraints required for them not to be positive.

In the formula, the first disjunction ranges over all supports S of path-flows; the second, over all decompositions D of flows with the chosen support. Note that these are both finite yet exponential in the size of $\mathcal{A}$. Similar to the formula $\Phi_1^{(q,q')}$, we can encode formulas of the form $\Phi_2^{(q,q')}$ (similar to $\Phi_1^{(q,q')}$, in skew-transpose of $G_{\mathcal{A}}$) and $\Phi_3^{(q,q')}$ corresponding to the other two types of reachability certificates. Notice that this does not suffice as the formula φ_{weight} contains quadratic polynomials of the form $w_i f_i(\vec{z}) + g_i(\vec{z})$ where $f_i(\vec{z})$ and $g(\vec{z})$ are linear polynomials. Intuitively, the variables w_i's correspond to the weights and the variables f_i's correspond to the decomposed flows. To get rid of these quadratic polynomials, we use the following helpful result from [73] that concerns a transformation from linear constraints over quadratic equations of a particular form to a quantifier-free formula in PAD.

From quadratic equations to divisibilities. Recall Theorem 11 that depicts the idea of the affine change of variables. Consider $\vec{A}, \vec{C}, \vec{b}, \vec{d}$ similarly defined and write P for the set of integer vectors $\vec{x}$ satisfying $\vec{A}x = \vec{b}$ and $\vec{C}x \leq \vec{d}$. Now, let $\vec{w} = (w_1, \ldots, w_n)$ and $\vec{z} = (z_1, \ldots, z_k)$ be disjoint sets of integer variables. For $1 \leq i \leq n$ let $Q_i(w_i, \vec{z})$ denote the quadratic polynomial $w_i f_i(\vec{z}) + g_i(\vec{z})$ where $f_i(\vec{z})$ and $g(\vec{z})$ are linear polynomials.

Lemma 16 (From [73]) *Given $\vec{A}, \vec{C}, \vec{b}, \vec{d}$ and $Q_i(w_i, \vec{z})$ as above, we can construct a formula $\varphi(\vec{z})$ in $\exists PAD$ with:*

$$\{\vec{z} \in \mathbb{Z}^k : \varphi(\vec{z})\} = \{\vec{z} \in \mathbb{Z}^k : \exists \vec{w} \in \mathbb{N}^n \ (Q_1(w_1, \vec{z}), \ldots, Q_n(w_n, \vec{z})) \in P\}$$

and such that $\varphi(\vec{z})$ is an exponential disjunction of subformulas $\psi_\ell(\vec{z})$ whose size is polynomial in the size of the Q_i and $(n+1)\mu$.

Indeed, we seek integer valuations of $\vec{w}$ and $\vec{z}$ such that $\vec{w} \geq \vec{0}$ and $(Q_1, \ldots, Q_n) \in P$. After applying an affine change of variables $\vec{x} = \vec{E}^{(\ell)}\vec{y} + \vec{u}^\ell$ on the definition of P, this reduces to finding solutions of the following system, with $\vec{w} \geq \vec{0}$ and $\vec{y} \geq \vec{0}$.

$$\begin{aligned}
w_1 f_1(\vec{z}) + g_1(\vec{z}) &= E^{(\ell)}_{1,1} y_1 + \cdots + E^{(\ell)}_{1,n-r} y_{n-r} + u^{(\ell)}_i \\
w_2 f_2(\vec{z}) + g_2(\vec{z}) &= E^{(\ell)}_{2,1} y_1 + \cdots + E^{(\ell)}_{2,n-r} y_{n-r} + u^{(\ell)}_2 \\
&\vdots \\
w_n f_n(\vec{z}) + g_n(\vec{z}) &= E^{(\ell)}_{n,1} y_1 + \cdots + E^{(\ell)}_{n,n-r} y_{n-r} + u^{(\ell)}_n
\end{aligned} \tag{5.4}$$

In turn, this is equivalent to finding a solution to the following system of divisibilities, again with $\vec{y} \geq \vec{0}$, if we ensure that both sides of each divisibility are either negative or non-negative (since the $\vec{w}$ could not take negative values).

$$\begin{aligned}
f_1(\vec{z}) &\mid E^{(\ell)}_{1,1} y_1 + \cdots + E^{(\ell)}_{1,n-r} y_{n-r} + u^{(\ell)}_i - g_1(\vec{z}) \\
f_2(\vec{z}) &\mid E^{(\ell)}_{2,1} y_1 + \cdots + E^{(\ell)}_{2,n-r} y_{n-r} + u^{(\ell)}_2 - g_2(\vec{z}) \\
&\vdots \\
f_n(\vec{z}) &\mid E^{(\ell)}_{n,1} y_1 + \cdots + E^{(\ell)}_{n,n-r} y_{n-r} + u^{(\ell)}_n - g_n(\vec{z})
\end{aligned}$$

The final existential PAD formula $\varphi(\vec{z})$ from Lemma 16 is as shown below.

$$\exists y_1 \ldots \exists y_{n-r} \bigvee_{\ell \in L} \bigvee_{j \in J} \bigwedge_{i=1}^{n} f_i(\vec{z}) \mid h_{\ell,i}(\vec{y}) - g_i(\vec{z}) \land \varphi_j(\vec{z}) \land \psi_j(\vec{y}, \vec{z}) \land \vec{y} \geq \vec{0}$$

Above, the set J indexes all possible choices of sign for both the left and right-hand sides of the divisibilities (either negative or non-negative). Then, the φ_j and ψ_j subformulas are quantifier-free PAD formulas without divisibilities to enforce said order.

5.5.2.3 Putting Everything Together

In this section, we combine the results from the previous subsection to construct φ_{reach} for Lemma 15.

Proof of Lemma 15. We first define the formula $\varphi^{(q,q')}_{\text{reach-nt}}(\vec{x},a,b)$ that is satisfied by a valuation V of $X \cup \{a,b\}$ if and only if there is a V-run from $(q,V(a))$ to $(q',V(b))$ without any zero-test transitions. By Proposition 1, there is such a V-run if and only if there is a V-run ρ from $(q,V(a))$ to $(q',V(b))$ without zero-test transitions and such that:

- there exists a configuration (u,k) such that, there is a run ρ_1 from $(q,V(a))$ to (u,k) that has a type-1 reachability certificate;
- there exists a configuration (v,k') such that, there is a run ρ_2 from (u,k) to (v,k') that has a type-3 reachability certificate;
- there is a run ρ_3 from (v,k') to $(q',V(b))$ that has a type-2 reachability certificate; and
- $\rho = \rho_1 \cdot \rho_2 \cdot \rho_3$.

We will construct formulas for the sub-automaton obtained by removing from $\mathcal{A}$ all zero-test transitions. Now using Lemma 16, each certificate can be encoded as a formula in the following form:

$$\exists y_1 \ldots \exists y_m \bigvee_{j\in J} \bigwedge_{i=1}^{n} f_i(\vec{x}) \mid h_i(\vec{y}) - g_i(\vec{x}) \wedge \varphi_j(\vec{x}) \wedge \psi_j(\vec{y}) \wedge \vec{y} \geq \vec{0} \tag{5.5}$$

Note that we have to take extra care to make sure that 0 does not appear on the left-hand side of the divisibilities while applying Lemma 16. This can be achieved by a finer analysis of the transformation of formulas using slack variables. Now all that remains is to obtain the formula $\varphi^{(q,q')}_{\text{reach-nt}}(\vec{x},a,b)$ in our desired form (Equation 5.5) is to use the fourth itemization from the previous properties of the witnessing reaching run and compose the encoding of the certificates to encode the run. The formula $\varphi^{(q,q')}_{\text{reach}}(\vec{x},a,b)$ expressing general reachability can then be defined by choosing an ordering on the zero tests. Formally, let ZT denote the set of all zero-test transitions. We write $1,\ldots,m \in ZT$ to denote an enumeration $(p_1,q_1),\ldots,(p_m,q_m)$ of a subset of zero-test transitions. We define $\varphi^{(q,q')}_{\text{reach}}(\vec{x},a,b)$ as:

$$\bigvee_{1,\ldots,m\in ZT} \exists k_0 \ldots \exists k_{m+1} \exists k'_0 \ldots \exists k'_{m+1} \Phi(\vec{x},\vec{k},\vec{k}')$$

where Φ is given by:

$$\Psi^{(q,p_1)}_{\text{reach-nt}}(\vec{x},k_0,k'_0,a,0) \wedge \Psi^{(q_m,q')}_{\text{reach-nt}}(\vec{x},k_{m+1},k'_{m+1},0,b) \wedge \bigwedge_{i=1}^{m-1} \Psi^{(q_i,p_{i+1})}_{\text{reach-nt}}(\vec{x},k_i,k'_i,0,0)$$

In words: for each enumeration of zero-test transitions, we take the conjunction of the intermediate $\varphi_{\text{reach-nt}}$ formulas as well as $\varphi_{\text{reach-nt}}$ formulas from the initial configuration to the final one.

Note that φ_{reach} has the required form as Equation 5.5. Indeed, the existentially quantified variables only appear in (the right-hand side of) divisibility constraints.

Also, we introduced an exponential number of disjunctions (over the enumeration of subsets of zero-test transitions), $2|T| + 4$ new variables (since $m \leq |T|$) and have not changed the bit-size length of constants after the construction of the Ψ subformulas. Thus, the bit-size of constants and the number of variables in φ_{reach} remain polynomial and $|\varphi_{\text{reach}}|$ is at most exponential in $|\mathcal{A}|$. □

5.6 Parikh One-counter Automata

In [73], the authors showed that the validity of ∃PAD is inter-reducible to the existential parameter synthesis problems for SOCAP. Our idea for studying an extension of OCA with parameters raised from the intuition of reproducing such an elegant connection to the fragment BIL. Note that although the universal parameter synthesis problem for SOCAP is reducible to BIL, it is still not clear if the validity of BIL sentences can be reduced to parameter synthesis problems for such models. In essence, following [73], one can easily establish a reduction to this effect for sentences of the form: $\forall\vec{x}\exists\vec{y} \bigvee_{i\in I} f_i(\vec{x}) \mid g(\vec{x},\vec{y}) \wedge f_i(\vec{x}) > 0 \wedge \varphi_i(\vec{x}) \wedge \vec{y} \geq \vec{0}$ but for the full fragment of BIL, it is still not clear. Towards this, we extend One-counter automata with *Parikh* constraints, a natural extension that allows any number of $\mathbb{Z}$-valued counters that are checked at the end of the run using a Presburger formula (on top of the existing counter that can be updated and tested for zero).[4] This can also be seen as a combination of OCA and *Parikh Automata* [85, 32, 33], another popular computational model in formal methods. We call it *Parikh One-counter Automata* (POCA). The model POCA is interesting in itself and has interesting properties from language-theoretic perspectives but this is out of the scope of this thesis, and we do not mention it here [31]. We are rather interested in extending POCA with parameters and studying the universal parameter synthesis problems.

Definition 3 *A* parametric POCA *is a tuple* $\mathcal{A} = (Q, T, \Delta, X, \varphi)$ *where:*

- *Q, T, Δ, and X are as in OCA with parameters,*
- *φ is an existential Presburger formula with $(|\Delta|)$ free variables.*

Given a run ρ, the Parikh Image *$\Phi(\rho)$ is a vector in $\mathbb{N}^T$, whose i-th component is the number of times transition $t_i \in \mathcal{T}$, appears in ρ. A run ρ in POCA is valid if it is valid as in OCA (maintain all the counter value constraints) and also it is* constraint-correct, *i.e., $\Phi(\rho)$ satisfies φ. Given a valuation $\mu\colon X \to \mathbb{N}$, $\mathcal{A}$ induces a POCA denoted by $\mathcal{A}_\mu$.*

In this section, we focus on the (universal) safety parameter synthesis problem for POCA with parameters. Given a parametric POCA $\mathcal{A}$ with parameter set X, a starting configuration $(q, 0)$ and a target state q_f, the *parametric universal nonemptiness problem*, Pune for short, asks whether it holds that, for all $\mu\colon X \to \mathbb{N}$, no run reaches q_f. Note that PUNE is the complement of the universal safety parameter synthesis problem.

Theorem 12 *The Pune problem for POCA with parametric updates is undecidable.*

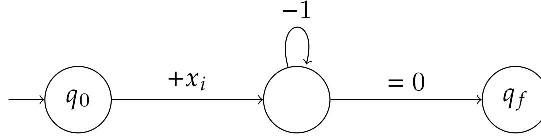
[4] We rely on a slightly different but equivalent definition, in which the Presburger formula actually specifies a relation on the number of times each transition is taken in the run. This explains the use of Rohit Parikh's name: a run is accepting if its Parikh image is accepted by the Presburger formula.

Proof. We present a reduction from Hilbert's Tenth Problem to the PUNE problem. Recall that Hilbert's Tenth Problem asks, given a polynomial with integer coefficients if it has a positive integer solution.

Let $P(x_1, \ldots, x_n)$ be such a polynomial and write $P = c_1 M_1 + \cdots + c_k M_k$ with each c_i in $\mathbb{Z}$ and each M_i a monomial with coefficient 1 (e.g., $x_1 x_2^2$). We construct a POCA $\mathcal{A}$ with parametric updates over the parameter set $\{x_1, \ldots, x_n\}$ that *evaluates* P. This is in the following sense: there are transitions $t_1, \ldots, t_k$ of $\mathcal{A}$ such that for any valuation μ of the parameters, there is a unique run ρ in $\mathcal{A}_\mu$ that reaches q_f, and, writing $|\rho|_{t_i}$ for the number of times t_i occurs in ρ:

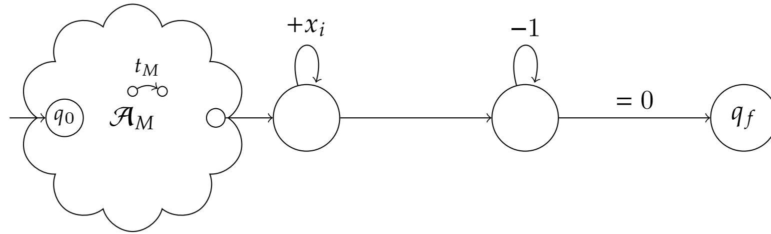
$$c_1|\rho|_{t_1} + \cdots + c_k|\rho|_{t_k} = P\left(\mu(x_1), \ldots, \mu(x_n)\right).$$

We start with the simplest case: $P = x_i$. Consider the following POCA:



Here, our transition t that evaluates to $P(\mu(x_i))$ is simply the self-loop: if the run is counter-correct, this loop must have been taken $\mu(x_i)$ times. Note that accepting runs end with a counter value of zero—these are properties we will keep throughout this construction.

Next, assume $P = Mx_i$ with M a monomial with coefficient 1. We assume that we have built a POCA $\mathcal{A}_M$ with a transition t_M that is taken $M(x_1, \ldots, x_n)$ times on accepting runs. We then build the following POCA for P:



As constraint, we combine the Parikh constraint of $\mathcal{A}_M$ with the statement that the $+x_i$ loop should be taken the same number of times as t_M; consequently, for any accepting run ρ of this POCA with valuation μ, the -1 loop is taken $|\rho|_{t_M}\mu(x_i)$ times, which is $M(\mu(x_1), \ldots, \mu(x_n))\mu(x_i)$ by hypothesis. This -1 loop is thus the transition that evaluates to $P(\mu(x_1), \ldots, \mu(x_n))$.

For the general case, we can chain together our POCA for each monomial one after the other and obtain our claimed POCA for any polynomial. The constraint formula can then compute the exact value of $P(\mu(x_1), \ldots, \mu(x_n))$ and accept if it is nonzero. Thus, there is no positive integer solution to P iff for all valuation μ, there is a run in $\mathcal{A}_\mu$ that reaches q_f. □

The undecidability result proved above is rather unfortunate. Indeed, we had originally expected that the PUNE problem for POCA with parametric updates would be a natural automata counterpart of the validity of sentences in BIL. Given the fairly simple shape of the automata built in the previous proof, we do not expect that natural restrictions of POCA will be able to play that role.

5.7 Conclusion

We have clarified the decidability status of universal parameter synthesis problems for OCA with parameters and show that, for several fixed ω-regular properties, they are in 2NEXP. For LTL, it is in 3NEXP, however for a fixed LTL specification, one can construct the corresponding Büchi automata and then solve it in 2NEXP. In the next chapter, we present a restriction of our model that will allow us to make these algorithms tractable and implementable.

Whether our new upper bounds are tight remains an open problem: the $\mathrm{NP}^{\mathrm{NP}}$ hardness result known [134, 93, 94] for other synthesis problems (see Table 5.1) does not match them.

We strongly believe that the BIL fragment will find uses beyond the synthesis problems for OCA with parameters: e.g. it might imply decidability of the software-verification problems that motivated the study of $\forall\exists_R\mathrm{PAD}^+$ in [29], or larger classes of quadratic string equations than the ones solvable by reduction to $\exists$PAD [101]. While we have shown BIL is decidable in co2NEXP, the best-known lower bound is the trivial coNP-hardness that follows from encoding the complement of the SUBSETSUM problem. (Note that BIL does not syntactically include the Π_1-fragment of PA so it does not inherit hardness from the results in [71].) Additionally, as mentioned in the previous section, it would be interesting to reduce the validity of BIL sentences to a synthesis problem. We also showed our attempt with a natural extension of One-counter automata with Parikh constraints, but unfortunately, that did not satisfy this goal. Hence a similar computation model for full BIL still evades us.

Chapter 6

Parameter Synthesis for One-Counter Automata with Parametric Tests Only

In the previous chapter, we established several complexity results for the parameter synthesis problems for One-counter automata against LTL specifications. We showed that for LTL specifications, it is decidable in 3NEXP, and for fixed LTL specifications, we can do it in 2NEXP by converting it to Büchi. In this chapter, we show that this problem is more tractable when we impose a restriction on the model. The content presented in this chapter is based on the work in [117].

6.1 Outline of the Chapter

Section 6.2 introduces the restricted model of one-counter automata we focus on in this chapter. Then, we introduce alternating two-way automata in Section 6.3, which is the backbone of the complexity results we provide here. Then, Section 6.4 introduces the reduction of parameter synthesis problems to the language emptiness problem of alternating two-way automata. We also show how we can improve upon the established complexity results for a particular class of parameter synthesis problems in Section 6.5.

6.2 One-Counter Automata with Parametric Tests

In this section, we introduce a subclass of general One-counter automata with parameters (SOCAP) where the restrictions are as follows:

- Parameters can only appear in tests;
- The updates are non-parametric and given in unary encoding.

Formally, *OCA with parametric tests (OCAPT)* allow for constant updates of the form $\{+a : a \in \{-1, 0, 1\}\}$ and zero and parametric tests. However, the set of parametric updates is empty, i.e., $PU = \emptyset$.

We consider the parameter synthesis problems for OCAPT. Our main result in this section is better complexity upper bounds for these synthesis problems compared to the ones for general SOCAP, as described in the previous chapter. Note that both the restrictions to the model considered above are essential to achieve this better complexity bound. In particular, we will prove the following theorem:

Theorem 13 *The coBüchi, Büchi and safety parameter-synthesis problems for OCAPT are in PSPACE; the reachability synthesis problem, in NP ^{coNP} = NP^{NP}. The LTL synthesis problem is in EXPSPACE.*

In the previous chapter, we opted for a first-order logic-based reduction in order to establish the decidability of the parameter-synthesis problems for the general model SOCAP. Here, we follow an automata-theoretic approach inspired by [23] to encode parameter valuations of OCAPT into words accepted by an *alternating two-way automaton*. First, we introduce the concept of alternating two-way automata.

6.3 Alternating Two-way Automata

Given a finite set Y, we denote by $\mathbb{B}^+(Y)$ the set of positive Boolean formulas over Y, including $true$ and $false$. A subset $Y' \subseteq Y$ satisfies $\beta \in \mathbb{B}^+(Y)$, written $Y' \models \beta$, if β is evaluated to $true$ when substituting $true$ for every element in Y', and $false$ for every element in $Y \setminus Y'$. In particular, we have $\emptyset \models true$. We define the size of β, $|\beta|$, inductively by setting $|\beta_1 \vee \beta_2| := |\beta_1 \wedge \beta_2| := |\beta_1| + |\beta_2| + 1$ and $|\beta| := 1$ for atomic formulas β.

Recall the definition of finite-state automata from Section 2.3.2. We can now define an *alternating two-way automaton* (A2A, for short) as an extension of finite-state automata as follows: Formally, an A2A is a tuple $\mathcal{T} = (Q, \Sigma, q_{in}, \Delta, F)$, where

- Q is a finite set of states,
- Σ is a finite alphabet,
- $q_{in} \in Q$ is the initial state,
- $F \subseteq Q$ is the set of final states, and
- $\Delta \subseteq Q \times (\Sigma \cup \{\text{first?}\}) \times \mathbb{B}^+(Q \times \{+1, 0, -1\})$ is the finite transition relation.

Intuitively, an A2A has a head over an input tape, and the A2A reads every letter based on the current position of the head on the tape. The $+1$ in Δ intuitively means that the head moves to the right; -1, that the head moves to the left; 0, that it stays at the current position. In particular, when $\Delta \subseteq Q \times \Sigma \times (Q \times \{+1\})$, then we deal with a classical one-way finite automaton. Furthermore, transitions are labelled by Boolean formulas over successors, which determine whether the current run branches off in a non-deterministic or a universal fashion or a combination thereof. The *size* of $\mathcal{T}$ is defined as $|\mathcal{T}| := |Q| + |\Sigma| + \sum_{(q,a,\beta)\in\Delta} |\beta|$.

Here we interpret alternating two-way automata on infinite words. A *run (tree)* γ of $\mathcal{T}$ on an infinite word $w = a_0 a_1 \cdots \in \Sigma^w$ is a (possibly infinite, but finitely branching)

rooted tree whose nodes are labelled with elements in $Q \times \mathbb{N}$ and such that it satisfies the following properties. The root of γ is labelled by $(q_{in}, 0)$. Moreover, for every node labelled by (s, m) with $k \in \mathbb{N}$ children labelled by $(q_1, n_1), \ldots, (q_k, n_k)$, there is a transition $(q, \sigma, \beta) \in \Delta$ such that, $\{(q_1, n_1 - m), \ldots, (q_k, n_k - m)\} \subseteq Q \times \{+1, 0, -1\}$ satisfies β. Further $\sigma = \text{first?}$ implies $m = 0$, and $\sigma \in \Sigma$ implies $a_m = \sigma$. In particular, all nodes (s, m) must have children unless there is a transition $(s, \cdot, \mathit{true}) \in \Delta$.

A run is *accepting* if all of its infinite branches contain infinitely many nodes with labels of the form $\{(q_f \times \mathbb{N}) \mid q_f \in F\}$.

The *language of* $\mathcal{T}$ is set of all words accepted by $\mathcal{T}$, i.e.,
$L(\mathcal{T}) \stackrel{\text{def}}{=} \{w \in \Sigma^\omega \mid \exists \text{ an accepting run of } \mathcal{T} \text{ on } w \text{ from } 0\}$. The *non-emptiness problem for A2As* asks, given an A2A $\mathcal{T}$ and $n \in \mathbb{N}$, whether $L(\mathcal{T}) \neq \emptyset$.

Lemma 17 (From [132]) *Language emptiness for A2As is in PSPACE.*

In what follows, from a given OCAPT $\mathcal{A}$ we will build an A2A $\mathcal{T}$ such that $\mathcal{T}$ accepts precisely those words which correspond to a valuation V of X under which all infinite runs satisfy the coBüchi condition. Hence, the corresponding synthesis problem for $\mathcal{A}$ reduces to checking non-emptiness of $\mathcal{T}$.

6.4 Transformation to Alternating Two-way Automata

In this section, we describe the construction of the A2A $\mathcal{T}$ in detail and elaborate on how we can encode the parameter synthesis problem of OCAPT to the language non-emptiness problem of an A2A.

6.4.1 Parameter Word: Encoding Valuation to Words

Following [23], we encode a valuation $V : X \to \mathbb{N}$ as an infinite *parameter word* $w = a_0 a_1 a_2 \ldots$ over the alphabet $\Sigma = X \cup \{\square\}$ such that $a_0 = \square$ and, for every $x \in X$, there is exactly one position $i \in \mathbb{N}$ such that $a_i = x$. We write $w(i)$ to denote its prefix $a_0 a_1 \ldots a_i$ up to the letter a_i. By $|w(i)|_\square$, we denote the number of occurrences of $\square$ in $a_1 \ldots a_i$. (Note that we ignore a_0.) Then, a parameter word w determines a valuation $V_w : x \mapsto |w(i)|_\square$ where $a_i = x$. We illustrate this with an example below.

Example 10 *Let $X = \{x_1, x_2\}$ be a set of parameters. Then, note that $w = \square\square\square x_1 x_2 \square^\omega$ is a parameter word that determines the valuation $V_w = \{x_1 \mapsto 2, x_2 \mapsto 2\}$. But $x_1 x_2 \square^\omega$ and $\square x_1 x_2 x_1 \square^\omega$ are not parameter words.*

Observe that for every valuation V, there is at least one parameter word w such that $V_w = V$. We denote the set of all parameter words as W_X.

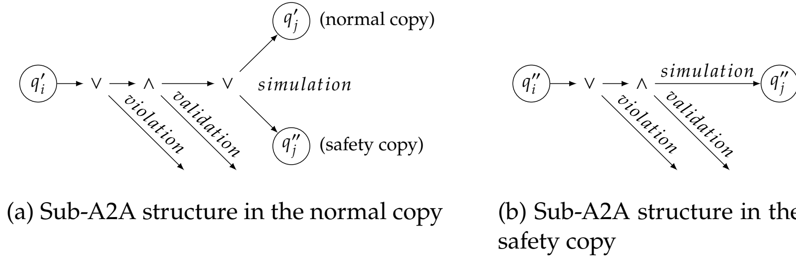
6.4.2 Overview of the Construction

From a given OCAPT $\mathcal{A} = (Q, T, \Delta, X)$, a starting configuration $(q_{in}, 0)$ and a set of target states F, we will now construct an A2A $\mathcal{T} = (S, \Sigma, s_{in}, \delta, S_f)$ that accepts words $w \in W_X$ such that, under the valuation $V = V_w$, all infinite runs from $(q_{in}, 0)$ visit F only finitely many times. Now we give the construction to prove the following statement.

Lemma 18 *For all OCAPT $\mathcal{A}$ there is an A2A $\mathcal{T}$ with $|\mathcal{T}| = |\mathcal{A}|^{O(1)}$ and $w \in L(\mathcal{T})$ if and only if all infinite V_w-runs of $\mathcal{A}$ starting from $(q_{in}, 0)$ visit F only finitely many times.*

The construction is based on the A2A framework presented in [23]. However, we employ the alternating semantics of the automaton more extensively. In order to capture the coBüchi condition, we simulate a "safety copy" of $\mathcal{A}$ within $\mathcal{T}$, where the final states in F are designated as "non-accepting sinks" (states with a self-loop and no other outgoing transitions) in $\mathcal{T}$. The accepting runs of $\mathcal{A}$ are simulated in $\mathcal{T}$ in a way that they have the option to enter this safety copy once they are certain that they will not visit F again and hence visiting them only finitely many times. Thus, for each state of $q \in Q$ in $\mathcal{A}$, we maintain two copies of the state in $\mathcal{T}$: $q' \in S$ simulating the normal version of q, and $q'' \in S$ simulating q from the safety copy.

Now, the key concept is to encode runs of $\mathcal{A}$ as branches of run trees of $\mathcal{T}$ on parameter words w. We achieve this by associating sub-trees t with root nodes labelled by (q', i) or (q'', i), both representing the configuration $(q, |w(i)|_\square)$ of $\mathcal{A}$. If a particular sub-tree t is accepting, it serves as evidence that all infinite runs of $\mathcal{A}$ starting from $(q, |w(i)|_\square)$ satisfy the coBüchi condition, i.e., visit F only finitely many times.



(a) Sub-A2A structure in the normal copy

(b) Sub-A2A structure in the safety copy

Figure 6.1: General Sub-A2A structure simulating (q_i, op, q_j)

Next, we provide the constructions of the sub-A2As for each kind of transition in $\mathcal{A}$. The overview of the construction is illustrated in Figure 6.1. The idea is that

- The constructed A2A $\mathcal{T}$ checks that the word it is reading is a valid parameter word that ensures it corresponds to a valuation of the parameters in $\mathcal{A}$. Otherwise, it rejects the word.
- $\mathcal{T}$ simulates every transition of $\mathcal{A}$ correctly. To do this, for all transitions $tr \in \Delta$ in $\mathcal{A}$, we create a sub-A2A $\mathcal{T}^{tr}_{\text{sub}}$ using copies of sub-A2As. For each such transition, one of two cases should hold:
  i. The transition cannot be simulated with the current counter value (because of a zero test or a decrement from value zero). Then we add a *violation branch* to check that it is indeed the case;

ii. The transition can indeed be simulated. Then, a *validation branch* checks the transition can be simulated, and a *simulation branch* reaches the next node in $\mathcal{T}$ with the updated counter value in $\mathcal{A}$.

- From a run sub-tree whose root is labelled with (q', i) or (q'', i), $\mathcal{T}$ verifies that all runs of $\mathcal{A}$ from $(q, |w(i)|_{\square})$ visit F only finitely many times. If the root node is of the form (q', i), then the *simulation branch* could reach a vertex labelled with r' or with r'' — with the idea being that $\mathcal{T}$ can choose to move to the safety copy or to stay in the "normal" copy of $\mathcal{A}$. If the root vertex is of the form (q'', i), the simulation branch can only reach the vertex labelled with r'' with the updated counter value that indicates that, once it enters the safety copy, it never comes out.
- We obtain the global A2A $\mathcal{T}$ by connecting all the sub-A2As. To ensure that all runs of $\mathcal{A}$ are simulated, we have the global transition relation δ in $\mathcal{T}$ be a conjunction of that of the sub-A2As which start at the same state $q \in \{p', p''\}$ for some $p \in Q$. For instance, let $tr_1 = (q, op_1, q_1)$ and $tr_2 = (q, op_2, q_2)$ be transitions of $\mathcal{A}$. The constructed sub-A2As $\mathcal{T}_{\text{sub}}^{tr_1}, \mathcal{T}_{\text{sub}}^{tr_2}$ will contain transitions $(q, \square, \beta_1) \in \delta_1$, and $(q, \square, \beta_2) \in \delta_2$ respectively. In $\mathcal{T}$, we make use of the conjunction to have $(q, \square, \beta_1 \wedge \beta_2) \in \delta$.
- Finally, the accepting states are chosen as follows: For every $q \in Q \setminus F$, we set $q'' \in S_f$ as accepting in $\mathcal{T}$. The idea is that if a run in $\mathcal{A}$ satisfies the coBüchi condition, then, after some point, it stops visiting target states. In $\mathcal{T}$, the simulated run can choose to move to the safety copy at that point and loop inside it forever, thus becoming an accepting branch. On the other hand, if a run does not satisfy the condition, its simulated version cannot stay within the safety copy. (Rather, it will reach the non-accepting sink states.) Also, the violation and the validation branches ensure that the operations along the runs have been simulated properly inside $\mathcal{T}$. It follows that $\mathcal{T}$ accepts precisely those words whose run-tree contains a simulation branch where states from F have been visited only finitely many times.

6.4.3 Construction of Sub-A2As

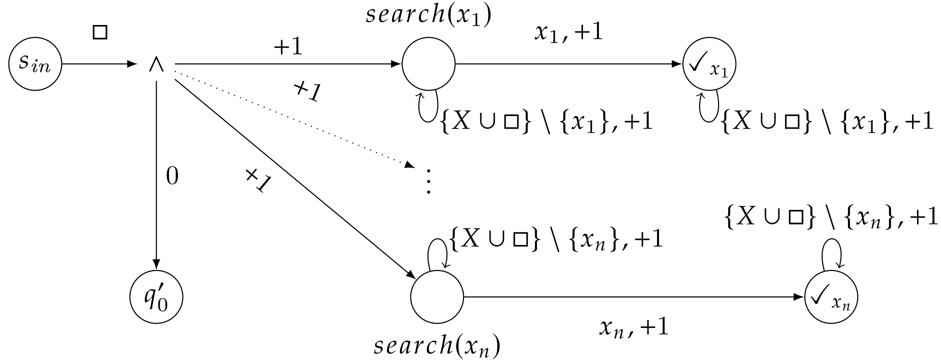
In the subsequent text, we will explain the construction of each sub-A2As in detail. For readability, for transitions of the form $(q_i, op, q_j) \in \Delta$, we will represent the $simulation$ branches as $s_i \rightsquigarrow s_j$ in δ, where s_i (similarly, s_j) represents q'_i or q''_i corresponding to the normal or the safety copy as described earlier.

Now we move forward to the detailed constructions for each operation.

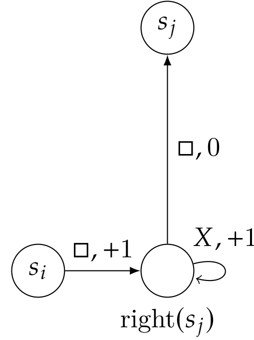
6.4.3.1 Verifying the Input Word

The sub-A2A $\mathcal{T}_{\text{sub}}^{\text{inp}}$ depicted in Figure 6.2a checks whether the given input is a valid parameter word. The states of the form $\checkmark_{x_i}$ represent that x_i has been found along the path. We let S_f consist of states $\checkmark_{x_i}$, one per $x_i \in X$.

Lemma 19 *It holds that* $L(\mathcal{T}_{\text{sub}}^{\text{inp}}) = W_X$.



(a) inp checking if the input is a valid parameter word



(b) inc encoding an increment

Figure 6.2: Sub-A2As for the word-validity check and to simulate increments of the form $(q_i, +1, q_j)$; we use $search(x)$, $\checkmark_x$, and $\mathrm{right}(q)$ as state names to make their function explicit

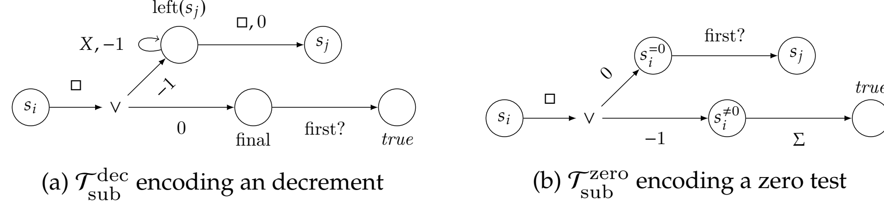
Proof. The A2A $\mathcal{T}_{\text{sub}}^{\text{inp}}$ consists of one deterministic one-way automata, per $x \in X$, whose language clearly corresponds to the set of words where x occurs exactly once. In $\mathcal{T}_{\text{sub}}^{\text{inp}}$, from the initial state and on the first letter □, a transition with a conjunction formula leads to all sub-automata for each x. The result follows. □

6.4.3.2 Encoding the Increment Operations

For every transition $(q_i, +1, q_j)$ in $\mathcal{A}$, we construct $\mathcal{T}_{\text{sub}}^{\text{inc}}$ (see Figure 6.2b). A run of this sub-A2A starts from s_i and some position c on the input word. Recall that c uniquely determines the current counter value in the simulated run of $\mathcal{A}$ (although, it should be noted c itself is not the counter value). Then, the run of $\mathcal{T}_{\text{sub}}^{\text{inc}}$ moves to the next occurrence of □ to the right of the current position and then goes to s_j accordingly. The correctness of this procedure follows from the construction and is straightforward.

6.4.3.3 Encoding the Decrement Operations

For transitions of the form $(q_i, -1, q_j)$ in $\mathcal{A}$, we construct $\mathcal{T}_{\text{sub}}^{\text{dec}}$ (see Figure 6.3a). In contrast to the increment sub-A2A, it also includes a *violation* branch in case the decrement would result in a negative counter value: On this branch, $\mathcal{T}_{\text{sub}}^{\text{dec}}$ attempts to read first? to determine if the position of the head corresponds to the the first letter of the word.



(a) $\mathcal{T}_{\text{sub}}^{\text{dec}}$ encoding an decrement (b) $\mathcal{T}_{\text{sub}}^{\text{zero}}$ encoding a zero test

Figure 6.3: Sub-A2As to simulate decrements and zero tests

Lemma 20 *Let $k, l \in \mathbb{N}$ and $w \in W_X$ with $\square$ the $(i+1)$-th letter of w. A run tree γ of $\mathcal{T}_{\text{sub}}^{\text{dec}}$ on w from k is accepting if and only if either $(s_i, k) \rightsquigarrow (s_j, l)$ is a part of γ and $|w(k)|_\square - 1 = |w(l)|_\square$, or $(s_i, 0) \rightsquigarrow (\text{final}, 0)$ is a part of γ and $k = 0$.*

Proof. Note that any accepting run γ of the sub-A2A must include at least one of the two finite branches from the claim. We further argue that each branch enforces the corresponding constraints if they appear in γ. Since these are mutually exclusive, it follows that γ includes exactly one of the branches.

If γ includes $(s_i, k) \rightsquigarrow (s_j, l)$ then $|w(k)|_\square - 1 = |w(l)|_\square$. The latter implies $k > l >= 0$ since, otherwise, the position of the head cannot be moved to the left. On the other hand, if γ includes $(s_i, n) \rightsquigarrow (\text{final}, n)$ then γ can only be accepting if $n = 0$. Hence, γ includes $(s_i, 0) \rightsquigarrow (\text{final}, 0)$. ◻

6.4.3.4 Encoding the Zero Tests

For every transition of the form $(q_i, = 0, q_j)$ in $\mathcal{A}$, we construct $\mathcal{T}_{\text{sub}}^{\text{zero}}$ (see Figure 6.3b) similarly to how we did for decrements. For the *validation* branch, it reads the letter first? to confirm the position of the head is at the beginning of the word. For the *violation* branch, it moves the head to the left to confirm that the head is not at the beginning.

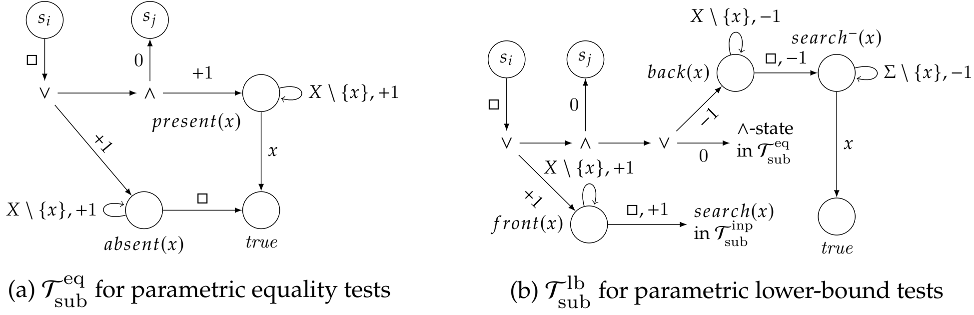
Lemma 21 *Let $k \in \mathbb{N}$ and $w \in W_X$ with $\square$ the $(k+1)$-th letter of w. A run tree γ of $\mathcal{T}_{\text{sub}}^{\text{zero}}$ on w from k is accepting if and only if either $(s_i, 0) \rightsquigarrow (s_j, 0)$ is a part of γ and $k = 0$, or $(s_i, k) \rightsquigarrow (s_i^{\neq 0}, k-1)$ is a part of γ and $|w(k)|_\square > 0$.*

Proof. We proceed as in the proof of Lemma 20.

If γ includes a branch with the state $s_i^{=0}$ then γ is accepting if and only if it reaches s_j. It can only reach s_j with the first? transition, i.e. when $k = 0$. Otherwise, it has to include a branch with $s_i^{\neq 0}$ and reading any letter, it reaches *true*. This is only possible if $k > 0$. Since the $(k+1)$-th letter of w is $\square$, the latter means $|w(k)|_\square > 0$. ◻

6.4.3.5 Encoding the Parametric Equality Tests

For every transition of the form $(q_i, = x, q_j)$ in $\mathcal{A}$, we construct $\mathcal{T}^{\text{eq}}_{\text{sub}}$ (see Figure 6.4a). For the $validation$ branch, it moves the head right, skipping over other variable symbols $X \setminus \{x\}$, while looking for k. For the $violation$ branch, it skips over other variable symbols while looking for the next $\square$.



(a) $\mathcal{T}^{\text{eq}}_{\text{sub}}$ for parametric equality tests (b) $\mathcal{T}^{\text{lb}}_{\text{sub}}$ for parametric lower-bound tests

Figure 6.4: Sub-A2As to simulate parametric tests

Lemma 22 *Let $k \in \mathbb{N}$ and $w \in W_X$ with $\square$ the $(k+1)$-th letter of w. A run tree γ of $\mathcal{T}^{\text{eq}}_{\text{sub}}$ on w from k is accepting if and only if either $(s_i, k) \rightsquigarrow (s_j, k)$ is part of γ and $V_w(x) = |w(k)|_\square$, or $(s_i, k) \rightsquigarrow (absent(x), k+1)$ is a part of γ and $V_w(x) \neq |w(k)|_\square$.*

Proof. Fix a word $w \in W_X$ with $\square$ as $(k+1)$-th letter. Consider any run tree γ of $\mathcal{T}^{\text{eq}}_{\text{sub}}$ on w. After reading the first $\square$, suppose γ has a branch leading to the state s_j. It must, therefore, also have a branch containing $present(x)$. Since, from there, it can only move to the state $true$ if it reads x before reading another $\square$ symbol to the right, we have $V(x) = |w(k)|_\square$.

If γ has a branch containing $absent(x_i)$, then it is accepting if and only if it reaches $true$ after reading another $\square$ before ever reading x. Hence, $V(x_i) \neq |w(k)|_\square$. $\square$

6.4.3.6 Encoding the Parametric Lower-bound Tests

For every transition of the form $(q_i, \geq x, q_j)$ in $\mathcal{A}$, we construct $\mathcal{T}^{\text{lb}}_{\text{sub}}$ (see Figure 6.4b). For the $validation$ branch, we check for equality to x or we check whether $> x$. We also create the corresponding $violation$ branches.

Lemma 23 *Let $k \in \mathbb{N}$ and $w \in W_X$ with $\square$ the $(k+1)$-th letter of w. A run tree γ of $\mathcal{T}^{\text{lb}}_{\text{sub}}$ on w from k is accepting if and only if either $(s_i, k) \rightsquigarrow (s_j, k)$ is part of γ and $|w(k)|_\square \geq V_w(x)$, or $(s_i, k) \rightsquigarrow (front(x), k+1)$ is a part of γ and $|w(k)|_\square < V_w(x)$.*

Proof. Fix a word $w \in W_X$ with $\square$ as $(k+1)$-th letter and consider any run tree γ of $\mathcal{T}^{\text{lb}}_{\text{sub}}$ on w. After reading the first $\square$, let us suppose it adds a branch checking $= x$ in $\mathcal{T}^{\text{eq}}_{\text{sub}}$. Then, γ is accepting if and only if it additionally contains a branch to (s_j, k) and $|w(k)|_\square = V_w(x)$. If it has the other sub-tree, i.e. it contains $back(x)$, γ is accepting if and only if it reaches the state $true$, which is possible only if there is a $\square$ to the left of the

current position and it reads an x to the left of that. It follows that it is accepting if and only if $|w(k)|_\square > V_w(x)$ and $(s_i, k) \rightsquigarrow (s_j, k)$ is part of γ.

If γ instead contains the branch with $front(x_i)$, it is accepting only if it can read x from $search(x)$ after having read a $\square$ from $front(x)$ to the right of the current position of the input. Hence, $|w(k)|_\square < V_w(x_i)$. $\square$

6.4.4 Correctness of the Encoding

In the previous subsection, we elaborated on the construction of every sub-A2As and proved the correctness of each construction in corresponding lemmas. Using those lemmas, it is straightforward to prove the correctness of our encoding, i.e., the correctness of Lemma 18. We present the proof formally below.

Proof of Lemma 18. We first start with the correctness of our encoding for the coBüchi parameter synthesis problem. In particular, we show that, $L(\mathcal{T}) = \{w \in W_X \mid$ all infinite V_w-runs of $\mathcal{A}$ visit F finitely many times from $(q_{in}, 0)\}$. We prove this in two parts:

$\supseteq$: Consider a word $w = a_0 a_1 a_2 \cdots \in W_X$, such that with valuation V_w all infinite V_w-runs of $\mathcal{A}$ visit F only finitely many times starting from $(q_{in}, 0)$. We have to show that w is accepted by $\mathcal{T}$, i.e., there exists an accepting run tree γ of w on $\mathcal{T}$. We will now grow an accepting run tree γ_{valid}. Since w is a valid parameter word we can add to γ_{valid} a sub-tree with root labelled by $(s_{in}, 0)$ and a branch extending to $(q'_0, 0)$ (see Lemma 19).

Consider now a valid infinite run ρ of $\mathcal{A}$ that visits F only finitely many times. Hence, ρ can be divided into $\rho = \rho_f \cdot \rho_{inf}$ such that ρ_f is a finite prefix and ρ_{inf} is the infinite suffix that never visits F. Let π be the path of the form $(q_{in}, op_1, q_1)(q_1, op_2, q_2)\ldots$ induced by ρ. We extend the division of π into $\pi = \pi_1 \cdot (q_{j-1}, op_j, q_j) \cdot \pi_2$ such that, $\pi_1 \cdot (q_{j-1}, op_j, q_j)$ is induced by ρ_f and π_2 is induced by ρ_{inf}. The idea is that the run ρ jumps to a "safety component" from the state q_j, after which it does not visit F at all as ρ satisfies the coBüchi condition.

Now, we further extend γ_{valid} by appending to it, from the $(q'_0, 0)$-labelled vertex, a sub-tree γ_{π_1} simulating the prefix π_1 as follows: for every transition of the form (q_i, op_{i+1}, q_{i+1}) where op_i is an increment or decrement, the corresponding $\mathcal{T}^{\text{inc}}_{\text{sub}}$ and $\mathcal{T}^{\text{dec}}_{\text{sub}}$ simulate the path from q'_i to q'_{i+1} correctly. Also, as every transition in π is valid in π_1 (i.e. does not result in negative counter values), using the first part of Lemmas 21, 22, and 23, we can take the $validation$ sub-trees of $\mathcal{T}^{\text{zero}}_{\text{sub}}$, $\mathcal{T}^{\text{eq}}_{\text{sub}}$, and $\mathcal{T}^{\text{lb}}_{\text{sub}}$, and append them to our run tree. For every $simulation$ branch, we stay at the normal copy, and we move from q'_i to q'_{i+1}. Now, for the transition (q_{j-1}, op_j, q_j), we do the same for the $violation$ and $validation$ branches but in the $simulation$ branch, we move to the *safety copy* and move to q''_j. Intuitively, this safety copy simulates the safety component of ρ as mentioned above. Now, with this, we append another sub-tree γ_{π_2}, which we create exactly in a similar way as γ_{π_1} but the $simulation$ branch stays in the safety copy, i.e., it moves from states of the form q''_i to q''_{i+1}. It is easy to see that, γ_{π_2} simulates the suffix ρ_{inf} correctly. Note that since π_2 does not visit F at all, the $simulation$ branch never reaches the non-accepting sink states in the safety copy, and it infinitely loops within the accepting states in the safety copy, making it accepting.

As ρ was chosen arbitrarily, we have that γ_ρ, for all infinite runs ρ, are accepting. To conclude, we need to deal with run trees arising from maximal finite runs–the runs that cannot be continued with any valid operation and hence, finite: We construct a sub-tree γ_{maxf} appending $\mathit{simulation}$ and $\mathit{validation}$ sub-trees for as long as possible. By definition of maximal finite runs, every such run reaches a point where all possible transitions are disabled. There, we append a $\mathit{violation}$ sub-tree which, using the second part of the mentioned lemmas, is accepting. Hence, γ_{valid} is accepting.

$\subseteq$: Consider a word $w \in L(\mathcal{T})$. We have to show that with valuation V_w, every infinite run of $\mathcal{A}$ visits F only finitely often from $(q_{in}, 0)$. We will prove the contrapositive of this statement: Let there exists a valuation V such that there is an infinite run of $\mathcal{A}$ that visits F infinitely often from $(q_{in}, 0)$, then for all words w with $V_w = V$, $w \notin L(\mathcal{T})$.

Let ρ be such an infinite run with valuation V_w. Now, ρ induces the path π which has the following form $(q_{in}, op_1, q_1)\ldots$, where for every i there exists a j such that $q_j \in F$. Recall that for every op_i, a run of $\mathcal{T}_{\mathrm{sub}}^{\mathrm{op_i}}$ has one $\mathit{simulation}$ branch, one or more $\mathit{validation}$ branches or a $\mathit{violation}$ branch. Now, as ρ is a valid infinite run of $\mathcal{A}$, every op_i can be taken, i.e., the counter value never becomes negative along the run. Hence, any $\mathit{violation}$ branch in any $\mathcal{T}_{\mathrm{sub}}^{\mathrm{op_i}}$ will be non-accepting already using the corresponding lemmas of the different operations. Hence, for every op_i appearing in π, let us consider the $\mathit{simulation}$ and $\mathit{validation}$ branches. Consider the global $\mathit{simulation}$ branch b in $\mathcal{T}$: $(s_{in}, 0) \rightsquigarrow s_0 \rightsquigarrow s_1 \ldots$, where each s_i in $\mathcal{T}$ represents q_i in $\mathcal{A}$ and is in the form q'_i or q''_i depending on whether it has jumped to the safety copy or not. If every s_i is of the form q'_i, then the infinite branch b has never moved to the safety copy and has not visited the accepting states at all. Hence, it is already non-accepting.

Now, for some l, let s_l be of the form q''_l representing q_l in $\mathcal{A}$, i.e., it has moved to the safety copy in $\mathcal{T}$. Note that if a branch in $\mathcal{T}$ moves to a safety copy, it can never escape; that is, for all $m \geq l$, s_m is of the form q''_m. Notice that from our assumption, there exists $n \geq l$, such that $q_n \in F$. Hence, s_n, representing q_n in the safety copy of $\mathcal{T}$, is a non-accepting sink establishing the fact that the branch b reaches a non-accepting sink making it non-accepting.

Note that b is a valid infinite branch in a run in A2A with no final states visited. Branch b will be present in every run of w in $\mathcal{T}$, resulting in no accepting run for w. □

6.5 Improving Upper Bound for Reachability Parameter Synthesis

Note that, Lemma 18 already gives an PSPACE upper-bound on the coBüchi parameter synthesis problem for OCAPT. Using the reductions from the previous chapter, we can establish that the reachability and safety parameter synthesis problems are also in PSPACE. In this section, we want to improve this upper bound for the reachability parameter synthesis specifically. Again following developments from [23], we now sketch a guess-and-check procedure using the fact that Lemma 18 implies a sufficient bound on valuations satisfying the reachability parameter synthesis problem. Recall that the reachability parameter synthesis problem asks whether all infinite runs of OCAPT reach a target state. Before improving the upper bound of the reachability

synthesis problem, we first establish the complexity of the universal reachability problem for non-parametric OCA. Note that this result is already interesting in itself and also crucially works as a building block for the reachability parameter synthesis problem.

6.5.1 Universal Reachability for Non-parametric OCA

Given a non-parametric OCA, the *universal reachability* problem asks, do all infinite runs from $(q_{in}, 0)$ reach the target state F or not? Note that the existential version of this problem has been proven to be NP-complete [73]. We first show that we can reduce an instance of the negation of the universal reachability problem to two instances of the existential reachability problem. Note that, in this section, we prove the result for general (succinct) OCA (SOCA) and it also works for OCA, where the updates are given in unary encoding.

Recall that, a *path* $\pi = q_0 q_1 \dots q_n$ in $G_{\mathcal{A}}$ is a *cycle* if $q_0 = q_n$. We say the cycle is *simple* if no state (besides q_0) is repeated. A cycle *starts from a zero test* if $\Delta(q_0, q_1)$ is "$= 0$". A *zero-test-free cycle* is a cycle where no $\Delta(q_i, q_{i+1})$ is a zero test. We define a *pumpable cycle* as being a zero-test-free cycle such that for all runs $\rho = (q_0, c_0) \dots (q_n, c_n)$ lifted from π we have $c_n \geq c_0$, i.e., the effect of the cycle is non-negative. The following result is inspired from [93]

Lemma 24 *Let $\mathcal{A}$ be a SOCA with an infinite run that does not reach F. Then, there is an infinite run of $\mathcal{A}$ which does not reach F such that it induces a path $\pi_0 \cdot \pi_1^{\omega}$, where π_1 either starts from a zero test or it is a simple pumpable cycle.*

Proof of Lemma 24. Let us call an infinite run of $\mathcal{A}$ a *safe run* if it does not reach F. Fix a safe run ρ. We denote q_{in} by q_0. Let $\pi = (q_0, op_1, q_1)(q_1, op_2, q_2) \dots$ be the path it induces. We denote by $\pi[i, j]$ the infix $(q_i, op_{i+1}, q_{i+1}) \dots (q_{j-1}, op_j, q_j)$ of π and by $\pi[i, \cdot]$ its infinite suffix $(q_i, op_{i+1}, q_{i+1}) \dots$ Suppose there are $0 \leq m < n \in \mathbb{N}$ such that $\pi[m, n]$ is a cycle that starts from a zero test. Note that if a cycle that starts from a zero test can be traversed once, it can be traversed infinitely many times. Then, the run lifted from the path $\pi[0, m] \cdot \pi[m, n]^{\omega}$ is our desired safe run. Now, let us assume that π has no cycles which start at a zero test. This means every zero test occurs at most once in π. Since the number of zero tests in $\mathcal{A}$ is finite, we have a finite $k \in \mathbb{N}$ such that there are no zero tests at all in $\pi[k, \cdot]$.

Now, consider $\pi[k, \cdot]$. Suppose it does not witness any non-negative effect cycle, i.e., every cycle in $\pi[k, \cdot]$ is negative. But we know π lifts to a valid infinite run which means the counter value cannot go below zero. This contradicts our assumption; Hence, there are $k \leq p < q$ such that $\pi[p, q]$ is a cycle with non-negative effect. It is easy to see that there must be r, s such that $p \leq r < s \leq q$ and $\pi[r, s]$ is a simple non-negative effect cycle. Also note that, $r \geq k$ which means that $\pi[r, s]$ does not have any zero tests. Hence, $\pi[r, s]$ is a simple pumpable cycle. Note that if a pumpable cycle can be traversed once, then it can be traversed infinitely many times. Using this fact, the run lifted from $\pi[0, r] \cdot \pi[r, s]^{\omega}$ is our desired safe run. □

Now we establish the complexity result of the universal reachability problem for SOCA using Lemma 24.

Theorem 14 *Checking whether all infinite runs from $(q_{in}, 0)$ reach a target state in a non-parametric one-counter automaton is coNP-complete.*

Proof of Theorem 14. We want to check whether all infinite runs starting from $(q_{in}, 0)$ reach F. Lemma 24 shows two conditions, one of which must hold if there is an infinite run that does not reach F. Note that both conditions are instances of existential reachability problems: a path to a cycle that starts from a zero test or to a simple pumpable cycle.

For the first condition, making the reachability-query instances concrete requires a configuration $(q, 0)$ and a state q' such that $\Delta(q, q')$ is a zero test. Both can be guessed and stored in polynomial time and space. For the other condition, we can assume that π_0 does not have any simple pumpable cycle. It follows that every cycle in π_0 has a zero test or has a negative effect. Let $W_{\max}$ be the sum of all the positive updates in $\mathcal{A}$. Note that the counter value cannot exceed $W_{\max}$ along any run lifted from π_0 starting from $(q_{in}, 0)$. Further, since π_1 is a simple cycle the same holds for $2W_{\max}$ for runs lifted from $\pi_0\pi_1$. Hence, we can guess and store in polynomial time and space the two configurations (q, c) and (q, c') required to make the reachability-query instances concrete.

Since the existential reachability problem for non-parametric SOCAP is in NP [73], we can guess which condition will hold and guess the polynomial-time verifiable certificates. This implies the problem is in coNP.

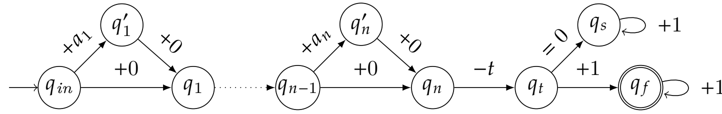


Figure 6.5: Reduction from non-SubsetSum to (universal) reachability for SOCA

For the lower bound, we give a reduction from the complement of the SubsetSum problem, which is NP-complete [61].

Given a set $S = \{a_1, a_2, \ldots a_n\} \subseteq \mathbb{N}$ and a target sum $t \in \mathbb{N}$, the Subset Sum problem asks whether there exists $S' \subseteq S$ such that $\sum_{a_i \in S'} a_i = t$. Given an instance of the Subset Sum problem with S and t, we create a SOCA $\mathcal{A}$ with initial configuration $(q_{in}, 0)$ and a single target state q_f as depicted in Figure 6.5. Note that, for every $1 \leq i \leq n$ there are two ways of reaching q_i from q_{i-1}: directly, with constant update $+0$; or via q'_i with total effect $+a_i$. Hence, for every subset $S' \subseteq S$, there exists a path from q_{in} to q_n with counter value $\sum_{a_i \in S'} a_i$. Clearly, if there exists S' such that $\sum_{a_i \in S'} a_i = t$ Subset Sum, then there exists an infinite run leading to q_s—not reaching the target state. On the other hand, if there is no such S', then all infinite runs reach q_f. Hence, the universal reachability in $\mathcal{A}$ is positive if and only if the answer to the SubsetSum problem is negative. □

6.5.2 Back to Parameter Synthesis for OCAPT: Proof of Theorem 13

In this section, we will establish the improved complexity upper bound for the reachability parameter synthesis problem of OCAPT. That will allow us to infer the

formal proof of Theorem 13, which is the main goal of this chapter. Towards this, we first state the following lemma.

Lemma 25 (Adapted from [23, Lemma 3.5]) *If there is a valuation V such that all infinite V-runs of $\mathcal{A}$ reach F, there is a valuation V' such that $V'(x) = exp(|\mathcal{A}|^{O(1)})$ for all $x \in X$ and all infinite V'-runs of $\mathcal{A}$ reach F.*

Proof. Using Lemma 18 for OCAPT $\mathcal{A}$, there is an A2A $\mathcal{T}$ of polynomial size (w.r.t. $\mathcal{A}$) such that, $L(\mathcal{T})$ is precisely the subset of W_X such that all infinite V_w-runs of $\mathcal{A}$ reach F. We then use the fact that there is a non-deterministic Büchi automaton such that $L(\mathcal{B}) = L(\mathcal{T})$ and $|\mathcal{B}| \in 2^{O(|\mathcal{T}|^2)}$ [137, 43].

Suppose for $\mathcal{A}$ the answer to the reachability parameter synthesis problem is positive, i.e. $L(\mathcal{B}) \neq \emptyset$. We know that the language of a Büchi automata is non-empty only if there is a "lasso" word which witnesses this. For all parameter words w accepted by a lasso there is a word $u \in \Sigma^*$ s.t. $|u| \leq |\mathcal{B}|$ and $w = u\square^\omega \in L(\mathcal{B})$. The result follows from our encoding of valuations. □

Now we can guess a valuation of the parameters of OCAPT with size bound in polynomial size (counter value in binary representation) (thus, of polynomial size). Now after guessing the valuation, we can check the universal reachability problem in the resulting non-parametric OCA in coNP (from Theorem 14). Hence, the improved upper bound for the reachability parameter synthesis problem is $\mathrm{NP}^{\mathbf{coNP}} = \mathrm{NP}^{\mathbf{NP}}$. This allows us to establish the proof of Theorem 13.

Proof of Theorem 13. In Lemma 18, we reduce the coBüchi synthesis problem to the non-emptiness problem for alternating two-way automata. Hence, we get the PSPACE upper bound. Note that, in the previous chapter, we reduced the Büchi and the safety parameter synthesis problems reduce to the coBüchi one (using Lemma 14) in polynomial time, and the reductions did not use any parametric updates in the automaton. Hence, they are valid for OCAPT, also resulting in the fact that these problems are also in PSPACE. Similarly, the reduction to LTL parameter synthesis to Büchi establishes the EXPSPACE upper bound for the problem. We have also improved the upper bound for the reachability parameter-synthesis problem to $\mathrm{NP}^{\mathbf{NP}}$. □

6.6 Conclusion

In this chapter, we have shown that the parameter synthesis problems are more tractable if we consider a restriction on the general model of One-counter automata with parameters. The restrictions are that the parameters are only allowed in tests, and the updates are given in unary encoding. It will be interesting to see if a more complicated parameter valuation encoding allows us to extend the automata-theoretic technique to reason about the full fragment, i.e., encoding the parametric updates. Note that, as argued in the previous chapter, there is still a gap between the upper-bound and lower-bound of these problems.

Chapter 7

Conclusion and Future Work

In this dissertation, we have presented our work on developing algorithms to learn temporal specifications from observed behaviours of a cyber-physical system and then using formal verification techniques to verify them to ensure safety and reliability. Every chapter contains open technical problems relevant to that chapter in the conclusion section. Here, we close our thesis with a brief summary of our results and a few possible general future directions in these fields.

For automatically learning specifications, we have focused on the framework of passive learning, where we have labelled system behaviours as input and learn concise specifications consistent with those inputs. As specifications, we have considered LTL, where the system behaviours are assumed to be collected as a discrete set of events and have proposed an approximation algorithm for learning that advances the existing learning algorithms for LTL in terms of scalability. Then, we have focused on more complex temporal specifications like MTL and STL that are able to capture the continuous evolution of a system over time. We have presented an SMT-based approach to learn concise and efficiently monitorable specifications in MTL and STL.

Regarding possible future directions towards this, it will be fascinating to investigate the learning problem of temporal specifications in an active learning setting. This setting assumes the presence of a teacher, where the learner tries to learn the desired temporal specification incrementally by asking the teacher two kinds of queries: membership and equivalence. This setting is interesting as an efficient active learning algorithm allows an engineer to incorporate newly observed system behaviours to update the already learned specification. Although there are substantial works on active learning settings for automata [4, 115], to the best of our knowledge, there are not many works on a similar setting for temporal logic. In [34], the authors explained how we can straightforwardly use a passive learning setting to devise an algorithm for active learning, but that is far from being efficient. The crux of the hardness towards this is that the equivalence query for LTL is computationally hard and more expensive than that of automata. Hence, it will be interesting to investigate fragments for which active learning can be efficient. We believe that Directed LTL, which we have introduced for our LTL learning algorithm, might be a good candidate towards this, but this requires more work and thorough analysis.

For the formal verification part, we focused on the classical model of one-counter automata that abstractly model systems with a possibly infinite state space, e.g.

programs controlled by one variable. We focused on the LTL parameter synthesis problem for these models, which asks whether a valuation of the parameters appearing in the model allows all infinite computations of the model to satisfy a desired LTL specification. We clarified the existing decidability and complexity results for this problem in the literature by exploiting a connection of this problem to the satisfiability of a fragment of Presburger arithmetic with divisibility (PAD). In this process, we have also introduced the largest known decidable fragment of one alternation PAD, named BIL.

Regarding future research direction in this field, the exact complexity of the decidability of the existential fragment of PAD (∃PAD) is still open. The current known complexity gap for the problem is that it is known to be in NEXP and NP-hard. There have been several related works around this problem containing the interreducibility of existential parameter synthesis of OCA [67] and the introduction of GCD operator [135], but the exact complexity of ∃PAD is still unknown. Note that the technique to determine the complexity of decidability of the fragment BIL in this dissertation also depends on the complexity of ∃PAD. It will also be interesting to generalise the automata-theoretic approach to solve the problems that can capture parametric updates. This can result in new insights into the complexity results for the parameter synthesis of OCA and, hence on, the exact complexity bounds on various fragments of PAD.

Another exciting direction will be to check these universal parameter synthesis problems for OCAs with specifications beyond LTL. Unlike the first part of the dissertation, we cannot consider MTL or STL to go beyond LTL, as OCAs are inherently discrete models. For Computation Tree Logic (CTL), the parameter synthesis problems are already undecidable in the existential framework. One possible direction will be for *flat freeze LTL*, which has been introduced to specify properties of data words [46]. The model-checking problem for flat Freeze LTL on classical one-counter automata is decidable [46, 94]. It will be interesting to investigate the parameter synthesis problems for these new specifications.

Bibliography

[1] Frances E Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970.

[2] Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. Parametric real-time reasoning. In S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal, editors, *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, pages 592–601. ACM, 1993.

[3] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, page 4–16, New York, NY, USA, 2002. Association for Computing Machinery.

[4] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[5] Mark A. Ardis. Formal methods for telecommunication system requirements: A survey of standardized languages. *Ann. Softw. Eng.*, 3:157–187, 1997.

[6] M. Fareed Arif, Daniel Larraz, Mitziu Echeverria, Andrew Reynolds, Omar Chowdhury, and Cesare Tinelli. SYSLITE: syntax-guided synthesis of PLTL formulas from finite traces. In *Formal Methods in Computer Aided Design, FMCAD*, 2020.

[7] Eugene Asarin, Alexandre Donzé, Oded Maler, and Dejan Nickovic. Parametric identification of temporal properties. In *Proceedings of the Second International Conference on Runtime Verification*, RV'11, page 147–160, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[9] Kevin Baldor and Jianwei Niu. Monitoring dense-time, continuous-semantics, metric temporal logic. In *RV*, volume 7687 of *Lecture Notes in Computer Science*, pages 245–259. Springer, 2012.

[10] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In *TACAS (1)*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[11] Luciano Baresi, Mohammad Mehdi Pourhashem Kallehbasti, and Matteo Rossi. Efficient scalable verification of LTL specifications. In *Icse (1)*, pages 711–721. IEEE Computer Society, 2015.

[12] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1267–1329. IOS Press, 2021.

[13] Ezio Bartocci, Jyotirmoy V. Deshmukh, Alexandre Donzé, Georgios Fainekos, Oded Maler, Dejan Nickovic, and Sriram Sankaranarayanan. Specification-based monitoring of cyber-physical systems: A survey on theory, tools and applications. In *Lectures on Runtime Verification*, volume 10457 of *Lecture Notes in Computer Science*, pages 135–175. Springer, 2018.

[14] Ezio Bartocci, Cristinel Mateis, Eleonora Nesterini, and Dejan Nickovic. Survey on mining signal temporal logic specifications. *Information and Computation*, 289:104957, 2022.

[15] David A. Basin, Felix Klaedtke, and Eugen Zalinescu. Algorithms for monitoring real-time properties. In Sarfraz Khurshid and Koushik Sen, editors, *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, volume 7186 of *Lecture Notes in Computer Science*, pages 260–275. Springer, 2011.

[16] David A. Basin, Srdan Krstic, and Dmitriy Traytel. Almost event-rate independent monitoring of metric dynamic logic. In *Rv*, volume 10548 of *Lecture Notes in Computer Science*, pages 85–102. Springer, 2017.

[17] David A. Basin, Srdjan Krstic, and Dmitriy Traytel. AERIAL: almost event-rate independent algorithms for monitoring metric regular properties. In *RV-CuBES*, volume 3 of *Kalpa Publications in Computing*, pages 29–36. EasyChair, 2017.

[18] A. P. Beltyukov. Decidability of the universal theory of natural numbers with addition and divisibility. *Journal of Soviet Mathematics*, 14(5):1436–1444, Nov 1980.

[19] O. Bernardi and Omer Giménez. A linear algorithm for the random sampling from regular languages. *Algorithmica*, 62:130–145, 2010.

[20] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, C-21(6):592–597, 1972.

[21] Dines Bjørner and Klaus Havelund. 40 years of formal methods - some obstacles and some possibilities? In *Fm*, volume 8442 of *Lecture Notes in Computer Science*, pages 42–61. Springer, 2014.

[22] Michael Blondin, Tim Leys, Filip Mazowiecki, Philip Offtermatt, and Guillermo A. Perez. Continuous one-counter automata. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13, 2021.

[23] Benedikt Bollig, Karin Quaas, and Arnaud Sangnier. The complexity of flat freeze LTL. *Logical Methods in Computer Science*, 15(3), 2019.

[24] Giuseppe Bombara and Calin Belta. Offline and online learning of signal temporal logic formulae using decision trees. *ACM Trans. Cyber-Phys. Syst.*, 5(3), mar 2021.

[25] Giuseppe Bombara, Cristian Ioan Vasile, Francisco Penedo Alvarez, Hirotoshi Yasuoka, and Calin Belta. A Decision Tree Approach to Data Classification using Signal Temporal Logic. In *Hybrid Systems: Computation and Control, HSCC*, Hscc '16, page 1–10, New York, NY, USA, 2016. Association for Computing Machinery.

[26] Silvia Bonfanti, Angelo Gargantini, and Atif Mashkoor. A systematic literature review of the use of formal methods in medical software systems. *Journal of Software: Evolution and Process*, 30(5):e1943, 2018.

[27] Ahmed Bouajjani, Marius Bozga, Peter Habermehl, Radu Iosif, Pierre Moro, and Tomás Vojnar. Programs with lists are counter automata. In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, pages 517–531, 2006.

[28] Jonathan P. Bowen and Victoria Stavridou. Safety-critical systems, formal methods and standards. *Softw. Eng. J.*, 8(4):189–209, 1993.

[29] Marius Bozga and Radu Iosif. On decidability within the arithmetic of addition and divisibility. In Vladimiro Sassone, editor, *Foundations of Software Science and Computational Structures, 8th International Conference, FOSSACS 2005*, volume 3441 of *Lecture Notes in Computer Science*, pages 425–439. Springer, 2005.

[30] Daniel Bundala and Joël Ouaknine. On parametric timed automata and one-counter machines. *Inf. Comput.*, 253:272–303, 2017.

[31] Michaël Cadilhac, Arka Ghosh, Guillermo A. Pérez, and Ritam Raha. Parikh One-Counter Automata. In Jérôme Leroux, Sylvain Lombardy, and David Peleg, editors, *48th International Symposium on Mathematical Foundations of Computer Science (MFCS 2023)*, volume 272 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:15, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[32] Michaël Cadilhac, Alain Finkel, and Pierre McKenzie. Affine Parikh automata. *RAIRO - Theoretical Informatics and Applications*, 46(04):511–545, 2012.

[33] Michaël Cadilhac, Andreas Krebs, and Pierre McKenzie. The algebraic theory of parikh automata. *Theory of Computing Systems*, 62(5):1241–1268, 2017.

[34] Alberto Camacho and Sheila A. McIlraith. Learning interpretable models expressed in linear temporal logic. *Proceedings of the International Conference on Automated Planning and Scheduling*, 29(1):621–630, May 2021.

[35] Agnishom Chattopadhyay and Konstantinos Mamouras. A verified online monitor for metric temporal logic with quantitative semantics. In *Rv*, volume 12399 of *Lecture Notes in Computer Science*, pages 383–403. Springer, 2020.

[36] Cristiana Chitic and Daniela Rosu. On validation of XML streams using finite state machines. In *Proceedings of the Seventh International Workshop on the Web and Databases, WebDB 2004, June 17-18, 2004, Maison de la Chimie, Paris, France, Colocated with ACM SIGMOD/PODS 2004*, pages 85–90, 2004.

[37] Glen Chou, Necmiye Ozay, and Dmitry Berenson. Explaining multi-stage tasks by learning temporal logic formulas from suboptimal demonstrations. In *Robotics: Science and Systems*, 2020.

[38] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[39] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 SMT solver. In *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2013.

[40] Barrett Clark and Tinelli Cesare. *Satisfiability Modulo Theories*, pages 305–343. Springer International Publishing, 2018.

[41] Patrick Cousot and Radhia Cousot. A gentle introduction to formal verification of computer systems by abstract interpretation. In Javier Esparza, Bernd Spanfelner, and Orna Grumberg, editors, *Logics and Languages for Reliability and Security*, volume 25 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 1–29. IOS Press, 2010.

[42] Thao Dang and Volker Stolz, editors. *Runtime Verification - 22nd International Conference, RV 2022, Tbilisi, Georgia, September 28-30, 2022, Proceedings*, volume 13498 of *Lecture Notes in Computer Science*. Springer, 2022.

[43] Christian Dax and Felix Klaedtke. Alternation elimination by complementation (extended abstract). In *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference, LPAR 2008, Doha, Qatar, November 22-27, 2008. Proceedings*, pages 214–229, 2008.

[44] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[45] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011.

[46] Stephane Demri and Ranko Lazic. Ltl with the freeze quantifier and register automata. *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 17–26, 2006.

[47] Jyotirmoy V. Deshmukh, Alexandre Donzé, Shromona Ghosh, Xiaoqing Jin, Garvit Juniwal, and Sanjit A. Seshia. Robust online monitoring of signal temporal logic. In *Rv*, volume 9333 of *Lecture Notes in Computer Science*, pages 55–70. Springer, 2015.

[48] Irit Dinur and David Steurer. Analytical approach to parallel repetition. In *Symposium on Theory of Computing, STOC*, pages 624–633, 2014.

[49] Adel Dokhanchi, Bardh Hoxha, and Georgios Fainekos. Online monitoring for temporal logic robustness. In *Rv*, volume 8734 of *Lecture Notes in Computer Science*, pages 231–246. Springer, 2014.

[50] Alexandre Donzé, Thomas Ferrère, and Oded Maler. Efficient robust monitoring for STL. In *Cav*, volume 8044 of *Lecture Notes in Computer Science*, pages 264–279. Springer, 2013.

[51] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *International Conference on Software Engineering, ICSE*, 1999.

[52] Rüdiger Ehlers, Ivan Gavran, and Daniel Neider. Learning properties in LTL $\cap$ ACTL from positive examples only. In *Formal Methods in Computer Aided Design, FMCAD*, pages 104–112, 2020.

[53] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with temporal logic on truncated paths. In Warren A. Hunt and Fabio Somenzi, editors, *Computer Aided Verification*, pages 27–39, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[54] @elonbachman. Tesla deaths, February 2020. Regularly updated at tesladeaths.com; version hosted on Zenodo will be updated periodically.

[55] Javier Esparza, Jan Kretínský, and Salomon Sickert. A unified translation of linear temporal logic to ω-automata. *J. Acm*, 67(6), 2020.

[56] Georgios E. Fainekos, Hadas Kress-Gazit, and George J. Pappas. Temporal logic motion planning for mobile robots. In *International Conference on Robotics and Automation, ICRA*, 2005.

[57] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Proofs that count. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 151–164. Acm, 2014.

[58] Nathanaël Fijalkow and Guillaume Lagarde. The complexity of learning linear temporal formulas from examples. In *International Conference on Grammatical Inference, ICGI*, 2021.

[59] Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2008.

[60] Jean-Raphaël Gaglione, Daniel Neider, Rajarshi Roy, Ufuk Topcu, and Zhe Xu. Learning linear temporal properties from noisy data: A maxsat-based approach. In Zhe Hou and Vijay Ganesh, editors, *Automated Technology for Verification and Analysis*. Springer International Publishing, 2021.

[61] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[62] Shalini Ghosh, Daniel Elenius, Wenchao Li, Patrick Lincoln, Natarajan Shankar, and Wilfried Steiner. ARSENAL: automatic requirements specification extraction from natural language. In *NASA Formal Methods, NFM*, 2016.

[63] Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *International Joint Conference on Artificial Intelligence, IJCAI*, 2013.

[64] Dimitra Giannakopoulou, Thomas Pressburger, Anastasia Mavridou, Julian Rhein, Johann Schumann, and Nija Shi. Formal requirements elicitation with FRET. In *International Conference on Requirements Engineering: Foundation for Software Quality, REFSQ*, 2020.

[65] E Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[66] E. Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.

[67] Stefan Göller, Christoph Haase, Joël Ouaknine, and James Worrell. Model checking succinct and parametric one-counter automata. In Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis, editors, *Automata, Languages and Programming, 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part II*, volume 6199 of *Lecture Notes in Computer Science*, pages 575–586. Springer, 2010.

[68] Felipe Gorostiaga and César Sánchez. Hlola: a very functional tool for extensible stream runtime verification. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021*, volume 12652 of *Lecture Notes in Computer Science*, pages 349–356. Springer, 2021.

[69] Kush Grover, Fernando S. Barbosa, Jana Tumova, and Jan Kretínský. Semantic abstraction-guided motion planning for scLTL missions in unknown environments. In *Robotics: Science and Systems XVII*, 2021.

[70] Christoph Haase. *On the complexity of model checking counter automata*. PhD thesis, University of Oxford, 2012.

[71] Christoph Haase. Subclasses of presburger arithmetic and the weak EXP hierarchy. In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 47:1–47:10. Acm, 2014.

[72] Christoph Haase. A survival guide to presburger arithmetic. *ACM SIGLOG News*, 5(3):67–82, 2018.

[73] Christoph Haase, Stephan Kreutzer, Joël Ouaknine, and James Worrell. Reachability in succinct and parametric one-counter automata. In *CONCUR 2009 - Concurrency Theory, 20th International Conference, CONCUR 2009, Bologna, Italy, September 1-4, 2009. Proceedings*, pages 369–383, 2009.

[74] Per Frederik Vilhelm Hasle and Peter Øhrstrøm. *The Significance of the Contributions of A.N. Prior and Jerzy Łoś in the Early History of Modern Temporal Logic*, volume II of *Logic and Philosophy of Time*, pages 31–40. Aalborg Universitetsforlag, 2019.

[75] Klaus Havelund and Doron Peled. Runtime verification: From propositional to first-order temporal logic. In *Rv*, volume 11237 of *Lecture Notes in Computer Science*, pages 90–112. Springer, 2018.

[76] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019*, 1995.

[77] Hsi-Ming Ho, Joël Ouaknine, and James Worrell. Online monitoring of metric temporal logic. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime*

Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings, volume 8734 of *Lecture Notes in Computer Science*, pages 178–192. Springer, 2014.

[78] Bardh Hoxha, Adel Dokhanchi, and Georgios Fainekos. Mining parametric temporal logic properties in model-based design for cyber-physical systems. *Int. J. Softw. Tools Technol. Transf.*, 20(1):79–93, 2018.

[79] Paul Hunter, Joël Ouaknine, and James Worrell. Expressive completeness for metric temporal logic. In *2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 349–357, 2013.

[80] Oscar H. Ibarra, Tao Jiang, Nicholas Q. Trân, and Hui Wang. New decidability results concerning two-way counter machines. *SIAM J. Comput.*, 24(1):123–137, 1995.

[81] Oscar H. Ibarra, Jianwen Su, Zhe Dang, Tevfik Bultan, and Richard A. Kemmerer. Counter machines and verification problems. *Theor. Comput. Sci.*, 289(1):165–189, 2002.

[82] Aaron Kane, Omar Chowdhury, Anupam Datta, and Philip Koopman. A case study on runtime monitoring of an autonomous research vehicle (ARV) system. In *Rv*, volume 9333 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2015.

[83] Brian Kempa, Pei Zhang, Phillip H. Jones, Joseph Zambreno, and Kristin Yvonne Rozier. Embedding online runtime verification for fault disambiguation on robonaut2. In *Formats*, volume 12288 of *Lecture Notes in Computer Science*, pages 196–214. Springer, 2020.

[84] Joseph Kim, Christian Muise, Ankit Shah, Shubham Agarwal, and Julie Shah. Bayesian inference of linear temporal logic specifications for contrastive explanations. In *International Joint Conference on Artificial Intelligence, IJCAI*, 2019.

[85] Felix Klaedtke and Harald Rueß. Monadic second-order logics with cardinalities. In *International Colloquium on Automata, Languages and Programming*, volume 2719 of *LNCS*, pages 681–696. Springer-Verlag, 2003.

[86] S. C. Kleene. *Representation of Events in Nerve Nets and Finite Automata*, pages 3–42. Princeton University Press, Princeton, 1956.

[87] Stephen Cole Kleene. Introduction to metamathematics, 1952.

[88] Zhaodan Kong, Austin Jones, and Calin Belta. Temporal logics for learning and detection of anomalous behavior. *IEEE Transactions on Automatic Control*, 62(3):1210–1222, 2017.

[89] Zhaodan Kong, Austin Jones, Ana Medina Ayala, Ebru Aydin Gol, and Calin Belta. Temporal logic inference for classification and prediction from data. In *Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control*, Hscc '14, page 273–282, New York, NY, USA, 2014. Association for Computing Machinery.

[90] Sascha Konrad and Betty H. C. Cheng. Real-time specification patterns. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 372–381. Acm, 2005.

[91] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real Time Syst.*, 2(4):255–299, 1990.

[92] Mahler Kurt. On the Chinese remainder theorem. *Mathematische Nachrichten*, 18(1-6):120–122, 1958.

[93] Antonia Lechner. Synthesis problems for one-counter automata. In Mikolaj Bojanczyk, Slawomir Lasota, and Igor Potapov, editors, *Reachability Problems - 9th International Workshop, RP 2015, Warsaw, Poland, September 21-23, 2015, Proceedings*, volume 9328 of *Lecture Notes in Computer Science*, pages 89–100. Springer, 2015.

[94] Antonia Lechner. *Extensions of Presburger arithmetic and model checking one-counter automata*. PhD thesis, University of Oxford, 2016.

[95] Antonia Lechner, Joël Ouaknine, and James Worrell. On the complexity of linear arithmetic with divisibility. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*, pages 667–676, 2015.

[96] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. General LTL specification mining. In *International Conference on Automated Software Engineering, ASE*, 2015.

[97] Nancy G. Leveson. Medical devices: the therac-25, 1985.

[98] Wenchao Li. *Specification Mining: New Formalisms, Algorithms and Applications*. PhD thesis, University of California, Berkeley, USA, 2013.

[99] Xie Li, Taolue Chen, Zhilin Wu, and Mingji Xia. Computing linear arithmetic representation of reachability relation of one-counter automata. In Jun Pang and Lijun Zhang, editors, *Dependable Software Engineering. Theories, Tools, and Applications - 6th International Symposium, SETTA 2020, Guangzhou, China, November 24-27, 2020, Proceedings*, volume 12153 of *Lecture Notes in Computer Science*, pages 89–107. Springer, 2020.

[100] Leonardo Lima, Andrei Herasimau, Martin Raszyk, Dmitriy Traytel, and Simon Yuan. Explainable online monitoring of metric temporal logic. In *TACAS (2)*, volume 13994 of *Lecture Notes in Computer Science*, pages 473–491. Springer, 2023.

[101] Anthony W. Lin and Rupak Majumdar. Quadratic word equations with length constraints, counter systems, and presburger arithmetic with divisibility. In Shuvendu K. Lahiri and Chao Wang, editors, *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, volume 11138 of *Lecture Notes in Computer Science*, pages 352–369. Springer, 2018.

[102] Alexis Linard and Jana Tumova. Active learning of signal temporal logic specifications. In *2020 IEEE 16th International Conference on Automation Science and Engineering (CASE)*, pages 779–785, 2020.

[103] Leonard Lipshitz. The diophantine problem for addition and divisibility. *Transactions of the American Mathematical Society*, pages 271–283, 1978.

[104] Leonard Lipshitz. Some remarks on the diophantine problem for addition and divisibility. *Bull. Soc. Math. Belg. Sér. B*, 33(1):41–52, 1981.

[105] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In Yassine Lakhnech and Sergio Yovine, editors, *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[106] Ju V Matijasevic. Enumerable sets are diophantine. In *Soviet Math. Dokl.*, volume 11, pages 354–358, 1970.

[107] Kuldeep S. Meel and Ofer Strichman, editors. *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*, volume 236 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[108] Norman D. Megill and David A. Wheeler. *Metamath: A Computer Language for Pure Mathematics*. Lulu Press, Morrisville, North Carolina, 2019.

[109] Steven P. Miller, Michael W. Whalen, and Darren D. Cofer. Software model checking takes off. *Commun. ACM*, 53(2):58–64, feb 2010.

[110] Marvin L. Minsky. Recursive unsolvability of post's problem of "tag" and other topics in theory of turing machines. *Annals of Mathematics*, 74(3):437–455, 1961.

[111] Sara Mohammadinejad, Jyotirmoy V. Deshmukh, Aniruddh Gopinath Puranic, Marcell Vazquez-Chanlatte, and Alexandre Donzé. Interpretable classification of time-series data using efficient enumerative techniques. In Aaron D. Ames, Sanjit A. Seshia, and Jyotirmoy Deshmukh, editors, *HSCC '20: 23rd ACM International Conference on Hybrid Systems: Computation and Control, Sydney, New South Wales, Australia, April 21-24, 2020*, pages 9:1–9:10. Acm, 2020.

[112] Christoph Molnar. Interpretable machine learning, 2022.

[113] Daniel Neider and Ivan Gavran. Learning linear temporal properties. In Nikolaj S. Bjørner and Arie Gurfinkel, editors, *Formal Methods in Computer Aided Design, FMCAD*, pages 1–10. Ieee, 2018.

[114] Laura Nenzi, Simone Silvetti, Ezio Bartocci, and Luca Bortolussi. A robust genetic algorithm for learning temporal specifications from data. In Annabelle McIver and Andras Horvath, editors, *Quantitative Evaluation of Systems*, pages 323–338, Cham, 2018. Springer International Publishing.

[115] J. Oncina and P. García. *Inferring Regular Languages in Polynomial Updated Time*. 1992.

[116] Joël Ouaknine and James Worrell. Some recent results in metric temporal logic. In Franck Cassez and Claude Jard, editors, *Formal Modeling and Analysis of Timed Systems*, pages 1–13, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[117] Guillermo A. Pérez and Ritam Raha. Revisiting Parameter Synthesis for One-Counter Automata. In Florin Manea and Alex Simpson, editors, *30th EACSL Annual Conference on Computer Science Logic (CSL 2022)*, volume 216 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 33:1–33:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

[118] Amir Pnueli. The temporal logic of programs. In *Symposium on Foundations of Computer Science, SFCS*, pages 46–57, 1977.

[119] M. Presburger. *Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt*. publisher not identified, 1931.

[120] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, Mar 1986.

[121] Michael O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969.

[122] Ritam Raha, Rajarshi Roy, Nathanaël Fijalkow, and Daniel Neider. Scalable anytime algorithms for learning fragments of linear temporal logic. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 263–280, Cham, 2022. Springer International Publishing.

[123] Ritam Raha, Rajarshi Roy, Nathanaël Fijalkow, Daniel Neider, and Guillermo Peréz. Synthesizing efficiently monitorable formulas in metric temporal logic. In Submission, 2023.

[124] Ritam Raha, Rajarshi Roy, Nathanaël Fijalkow, and Daniel Neider. SCARLET: Scalable Anytime Algorithm for Learning LTL, January 2022.

[125] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In *Computer-Aided Verification, CAV*, 2019.

[126] Heinz Riener. Exact synthesis of LTL properties from traces. In *Fdl*, pages 1–6. Ieee, 2019.

[127] Alan J.A. Robinson and Andrei Voronkov. *Handbook of automated reasoning*, volume 1. Gulf Professional Publishing, 2001.

[128] Julia Robinson. Definability and decision problems in arithmetic. *The Journal of Symbolic Logic*, 14(2):98–114, 1949.

[129] Rajarshi Roy, Dana Fisman, and Daniel Neider. Learning interpretable models in the property specification language. In *International Joint Conference on Artificial Intelligence, IJCAI*, pages 2213–2219. ijcai.org, 2020.

[130] Kristin Yvonne Rozier. Specification: The biggest bottleneck in formal methods and autonomy. In *Verified Software. Theories, Tools, and Experiments, VSTTE*, 2016.

[131] Eigo Segawa, Morito Shiohara, Shigeru Sasaki, Norio Hashiguchi, Tomonobu Takashima, and Masatoshi Tohno. Preceding vehicle detection using stereo images and non-scanning millimeter-wave radar. *IEICE Trans. Inf. Syst.*, 89-d:2101–2108, 2006.

[132] Olivier Serre. Parity games played on transition graphs of one-counter processes. In Luca Aceto and Anna Ingólfsdóttir, editors, *Foundations of Software Science and Computation Structures, 9th International Conference, FOSSACS 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25-31, 2006, Proceedings*, volume 3921 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 2006.

[133] Simone Silvetti, Laura Nenzi, Luca Bortolussi, and Ezio Bartocci. A robust genetic algorithm for learning temporal specifications from data. *CoRR*, 2017.

[134] A. Prasad Sistla and Edmund M. Clarke. The complexity of propositional linear temporal logics. *J. Acm*, 32(3):733–749, 1985.

[135] Mikhail R. Starchak. *Quasi-Quantifier Elimination Algorithms and Definability Problems in Arithmetics with Divisibility*. PhD thesis, Saint Petersburg State University, 2022.

[136] Prasanna Thati and Grigore Rosu. Monitoring algorithms for metric temporal logic specifications. In Klaus Havelund and Grigore Rosu, editors, *Proceedings of the Fourth Workshop on Runtime Verification, RV@ETAPS 2004, Barcelona, Spain, April 3, 2004*, volume 113 of *Electronic Notes in Theoretical Computer Science*, pages 145–162. Elsevier, 2004.

[137] Moshe Y. Vardi. Reasoning about the past with two-way automata. In *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings*, pages 628–641, 1998.

[138] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.

[139] Joachim von zur Gathen and Malte Sieveking. A bound on solutions of linear integer equalities and inequalities. *Proceedings of the American Mathematical Society*, 72(1):155–158, 1978.

[140] Zhe Xu, Calin Belta, and Agung Julius. Temporal logic inference with prior information: An application to robot arm movements. *IFAC-PapersOnLine*, 48(27):141–146, 2015. Analysis and Design of Hybrid Systems ADHS.

[141] Zhe Xu, Marc Birtwistle, Calin Belta, and Agung Julius. A temporal logic inference approach for model discrimination. *IEEE Life Sciences Letters*, 2(3):19–22, 2016.

[142] Zhe Xu and Ufuk Topcu. Transfer of temporal logic formulas in reinforcement learning. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 4010–4018. ijcai.org, 2019.

[143] Hengyi Yang, Bardh Hoxha, and Georgios E. Fainekos. Querying parametric temporal logic properties on embedded systems. In Brian Nielsen and Carsten Weise, editors, *Testing Software and Systems - 24th IFIP WG 6.1 International Conference, ICTSS 2012, Aalborg, Denmark, November 19-21, 2012. Proceedings*, volume 7641 of *Lecture Notes in Computer Science*, pages 136–151. Springer, 2012.