Probabilistic Verification Beyond Context-Freeness

Guanyan Li

St Cross

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2024

Acknowledgements

I would like to express my deepest gratitude to my supervisors, Prof. Andrzej Murawski and Prof. Luke Ong, for their invaluable guidance, support, and mentorship throughout my research journey. Their expertise and insights have been instrumental in shaping this thesis and my growth as a researcher.

I am particularly indebted to Prof. Murawski for his meticulous attention to detail, especially in the rigorous proofs that form the backbone of this thesis. His patience and expertise in navigating the complex landscape of theoretical computer science, with its abundance of symbols and technical intricacies, have been crucial to the development and refinement of my work. His commitment to precision has significantly elevated the quality of this research.

I am equally grateful to Prof. Luke Ong for his continued support and insightful contributions, even after his move from Oxford to Nanyang Technological University (NTU) in Singapore. His broad perspective and deep understanding of the field have been invaluable in shaping the direction of this research.

I extend my sincere thanks to Dr. Peixin Wang, a postdoctoral researcher and friend, for his collaboration on other projects and for his assistance in proofreading this thesis. His insights and feedback have been extremely valuable. I would also like to thank Zhe Zhou from Purdue University for his friendship and meticulous proofreading, which greatly improved the quality of this work.

My research has benefited greatly from the academic environments of several institutions. I am grateful to the University of Oxford for providing an excellent research atmosphere, to Tsinghua University and Prof. Fei He for hosting me during a portion of my research, and to NTU Singapore for accommodating my visits to work with Prof. Ong. These experiences have broadened my perspectives and enriched my research.

I owe a tremendous debt of gratitude to my family, especially my parents, whose unwavering support and encouragement have been the bedrock of my academic journey. Their belief in me, their patience during challenging times, and their constant words of motivation have been invaluable. This thesis would not have been possible without their sacrifices and the strength they've given me throughout this process. Their support has truly been the foundation upon which this work stands.

My heartfelt appreciation goes to my brother and my girlfriend for their constant support and understanding. Their presence in my life has been a source of strength and motivation throughout this challenging process.

I am deeply grateful to several individuals who contributed to the proofreading of this thesis. Dr Peixin Wang (NTU) provided valuable feedback on the introduction and overall structure. Zhe Zhou (Purdue University) offered insightful comments on the introduction. Yuanliang Chen (Tsinghua University) reviewed the experiments section, while Hongyu Fan (Tsinghua University) examined the preliminaries and interrelation chapters. I extend special thanks to Qiyuan Xu (NTU) for his extensive efforts, providing detailed comments on readability, suggesting sentence revisions, and identifying minor technical errors. His contributions significantly enhanced the quality of this thesis. I also thank Xingyu Xie (MPI-SP) for improving the readability of the termination analysis and for engaging in fruitful discussions on clarity.

Finally, I would like to acknowledge the friends I've made in Oxford. Their companionship, intellectual discussions, and moral support have made this journey not just academically rewarding but also personally fulfilling.

This thesis stands as a testament to the collective support of all these individuals and institutions. The complexity of theoretical computer science research, with its demanding proofs and abstract concepts, has been made navigable through their guidance and encouragement. Any errors or shortcomings in this work are entirely my own.

Abstract

Probability theory and probabilistic modelling have become indispensable across a wide range of scientific disciplines, with probabilistic programs emerging as a particularly flexible means of encoding sophisticated models in executable code. These programs, which merge traditional programming constructs with probabilistic reasoning, play a pivotal role in areas such as machine learning and systems biology. However, as these programs gain prominence, the demand for robust verification methods becomes increasingly urgent.

While Markov Chains (MCs) have long been foundational tools in this field, their limitations in modelling complex, recursive systems have prompted the development of more expressive models, such as Recursive Markov Chains (RMCs) and Probabilistic Pushdown Automata (pPDAs). Recent advancements, including extensions to higher-order grammars like Probabilistic Higher-Order Recursion Schemes (PHORS), offer enhanced expressiveness but introduce significant challenges, such as the undecidability of Almost Sure Termination (AST). This underscores the need for models that balance expressiveness with decidability.

This research presents and examines the Restricted Probabilistic Tree-Stack Automaton (rPTSA), a novel probabilistic model that extends recursive models while circumventing the undecidability issues associated with PHORS. Originating from the Restricted Tree-Stack Automaton (rTSA), the rPTSA model inherits broad connections to various fields, including linguistics and higher-order verification. Building on these connections, the research also introduces Probabilistic Additive HORS (PAHORS), which incorporates an affineness constraint and pair-product structures into PHORS.

We demonstrate that these two models, rPTSA and PAHORS, are equi-expressive, as they inherit characteristics from their non-probabilistic counterparts, rTSA and Multiplicative Additive HORS (MAHORS). Furthermore, rPTSA and PAHORS are shown to be strictly more expressive than order-1 PHORS and incomparable to order-2 PHORS, and they hence offer a balance between expressiveness and decidability.

Utilising their equi-expressiveness, this work primarily focuses on the rPTSA model. Firstly, it establishes the decidability of the Almost-Sure Termination (AST) and Positive Almost-Sure Termination (PAST) properties for rPTSA, and develops methods for quantitatively analysing termination probability and expected

termination time. The research also delves into model checking for both linear-time properties (ω-regular and LTL) and branching-time properties (PCTL*). Notably, the PCTL* model checking procedure relies heavily upon the algorithm established for LTL. Additionally, this research examines the restriction property of rPTSA, a crucial feature that prevents rPTSA from attaining Turing-completeness.

The practical contribution of this work is the development of the Probabilistic Tree-Stack Verifier (PTSV), an efficient tool for determining termination probability and expected termination time. Further attention is given to PAHORS and other key models, such as pPDA and MCFG. The study discusses the mutual conversions between rPTSA and PAHORS, as well as conversions from other models of interest, such as pPDA and Stochastic MCFG, to rPTSA.

Contents

The theory of probabilities is at bottom nothing but common sense reduced to calculus; it enables us to appreciate with exactness that which accurate minds feel with a sort of instinct for which times they are unable to account.

— Pierre-Simon Laplace

1 Introduction

Contents

1.1 Probabilistic Programs

Probability, a concept deeply embedded in the fabric of reality, permeates every aspect of our world. Its omnipresence, deeply intertwined with human reasoning, is poetically underscored by Blaise Pascal in his musings [1], as he observes:

> "Chance gives rise to thoughts, and chance removes them; no art can keep or acquire them"

Given its profound significance, probability is inevitably captured in the realm of science, where intuition and common sense are distilled into precise reasoning, giving rise to the theory of probability. It then serves as a bridge between the abstract notions of chance and the concrete frameworks of scientific inquiry, quantifying

uncertainty and allowing for systematic investigation of the stochastic nature of phenomena, as elucidated by Richard von Mises [2]:

> "Probability theory is the science of the laws of chance"

This conceptual foundation of probability theory finds its practical counterpart in the realm of computation through probabilistic programs. These programs equip usual computational constructs with probabilistic sampling statements. By integrating the power of probability with traditional programming paradigms, probabilistic programs enable us to model, simulate, and analyse complex systems where uncertainty and randomness are key factors [3, 4].

In fact, the interaction with probability theory happens at the early days of computer science [5] and has yeilded countless fruitful results. In the field of cryptography [6, 7], probability has long been thought of as essential for ensuring the security and integrity of digital communications, making them indispensable in the modern digital age. In the area of algorithmic, probabilistic programs are considered essential to the efficient solution of many algorithmic problems [8–10].

Wide Applications of Probabilistic Programs Over the past few decades, coinciding with the rapid advancement in machine learning, probabilistic programs have also found their place on a broader stage. For instance, in the realm of reinforcement learning (RL), probabilistic models are pivotal in navigating the uncertainties and complexities typical of diverse environments. These models often lead to greater efficiency [11–13] and enhanced robustness [14].

The evolution of this interplay between computer science and probability theory extends its benefits to fields beyond traditional computer science. For example, in biology, probabilistic models have become an essential tool in numerous analytical tasks. They provide a mathematical and a programmable framework to represent the inherent uncertainties and stochastic processes that are characteristic of many biological phenomena. In the study of signalling biological pathways [15–17], these models enable researchers to simulate and predict cellular responses to various stimuli or environmental changes. Moreover, they are applied in DNA analysis [18, 19] and pathological studies [20].

Similarly, in the field of power management, probabilistic models play a crucial role in tackling the complexities and uncertainties of energy systems. They are indispensable for simulating and predicting system behaviour under varied conditions, thereby facilitating efficient and reliable power management strategies [21–23].

Bayesian Statistical Probabilistic Programming Additionally, a noteworthy development in probabilistic programming is the emergence of Bayesian Statistical Probabilistic Programming (BSPP) [24–30]. This innovative approach is centred on formulating and analysing statistical models using probabilistic programming techniques. BSPP's primary utility lies in defining generative models, positioning it as a common methodology in generative machine learning.

More precisely, a BSPP program is distinguished from traditional probabilistic programs by its incorporation of the `observe` (or `score`) statements. In BSPP,

sampling statements articulate initial beliefs or existing knowledge about a phenomenon [25, 30], whereas the `observe` or `score` statements are used to integrate likelihoods based on actual observed data. A BSPP program hence represents a *posterior distribution* that refines the initial belief with actual data according to the Bayes' rule.

The versatility and expressiveness of BSPP programming enable the concise and elegant construction and examination of statistical models. As a result, the BSPP domain is witnessing rapid growth, with the development of several dedicated universal languages, including Church [31], Anglican [32], Gen [33], Pyro [34], Edward [35], and Turing [36], etc.

1.1.1 Verifying Probabilistic Programs

Probabilistic Verification Is Important

The preceding discussion, highlighting the critical role and widespread application of probabilistic programs, necessitates an exploration into the importance of probabilistic verification. This significance can be examined from the following three perspectives: as a means of ensuring safety, as a tool for system analysis, and as a basis for fundamental assumptions in probabilistic computing.

Probabilistic Verification as a Safety Guarantee Analogous to the deterministic scenario, in critical systems where failures can lead to catastrophic outcomes, the verification of program correctness is paramount. Historical incidents, such as the Therac-25 radiation therapy machine accident [37] and the Ariane 5 Flight 501 failure [38], underscore the disastrous consequences of system errors. In this context, program verification is not just a theoretical exercise but a practical necessity. Differing from traditional methods, probabilistic verification addresses the inherent uncertainty in complex systems, providing a more realistic appraisal by quantifying error probabilities. This approach is particularly crucial when obtaining precise results is impractical, such as in dealing with certain hardware issues [39] where absolute reliability is unattainable, and in managing medical treatments [40, 41], among other scenarios.

Probabilistic Verification as System Analysis In systems conceptualised through probabilistic models, verification sometimes transcends its conventional role to become an analytical tool for understanding system behaviour. For instance, in the biological analysis of signalling pathways [15–17], probabilistic model verification techniques such as probabilistic model checking can offer additional insights into system behaviour.

Probabilistic Verification for Fundamental Assumptions Furthermore, within the realm of certain probabilistic systems, such as BSPP, verification forms the cornerstone of all operations. Key assumptions for the correct functioning of these systems include *almost sure termination* (AST) and *positive almost sure termination* (PAST). AST refers to the property that a probabilistic program will terminate with probability one. PAST, a stronger requirement, not only mandates termination

to happen almost surely but also expects it to happen within a finite number of steps. In BSPP, where programs represent posterior distributions [42], effective sampling from these distributions is crucial. Programs that fail to demonstrate AST or PAST properties are considered malfunctioning, as this implies a potential failure in the sampling process due to program divergence. Another example happens in applications like QuickCheck [43, 44], a property-based testing framework in Haskell, the PAST assumption is essential for the efficacy of the testing process.

Challenges in Verifying Probabilistic Programs

The challenge of probabilistic verification is highlighted by the hardness of the AST analysis of universal probabilistic programs. This property is categorised as Π_2^0-complete [45]. Within the arithmetical hierarchy framework [46, 47], the Π_2^0 class encompasses all sets S defined as:

$$S := \{x \mid \forall y_1 \exists y_2 . (x, y_1, y_2) \in R\}$$

for a *decidable* relation R. This classification implies that the problem is not recursively enumerable, a complexity level even exceeding that of the halting problem for non-random universal programs.

Practical challenges are also prominent. A probabilistic program, unlike the deterministic case, defines a *distribution* of possible outcomes rather than a singular deterministic result. Reflecting in the running behavior, in discrete probabilistic programs, a run is not a straightforward deterministic sequence of statements. Instead, it resembles a Markov chain with transitions between states determined probabilistically [48, 49]. The situation is getting even more complicated when dealing with continuous distributions [26, 50].

These characteristics significantly alter the semantics of the programs [25, 30, 51] compared to their deterministic counterparts, often rendering conventional non-probabilistic techniques, such as predicate abstraction [48] and trace abstraction [52], inapplicable. This necessitates new formalisations of probabilistic semantics [49, 53] and novel techniques tailored to these semantics [48, 52, 54].

1.1.2 Approaches for Probabilistic Verification

In response to the formidable challenges previously discussed, researchers have devised various effective approaches to probabilistic verification. These approaches are broadly categorised into two types: abstraction-based methods and exact methods. Each type offers distinct advantages and is suitable for different verification scenarios.

Abstraction-Based Methods Abstraction-based methods in probabilistic verification strike a balance between computational feasibility and result precision. These techniques reduce the complexity of probabilistic systems by abstracting non-critical details while preserving those crucial for verification.

These methods are often used to analyse the *quantitative problem* in probabilistic programs, determining whether the probability of violating a property is below

a given threshold. An example is provided in [48], where the authors adapt the predicate-abstraction-based CEGAR procedure [55] for probabilistic contexts. Here, the *state space* of a probabilistic program is abstracted using predicates, with each assignment representing a distinct set of program states. These predicates are generated dynamically by identifying spurious counterexamples that do not lead to actual violations.

The *template-based exponential polynomial synthesis* method [54] is another approach, deriving bounds on the probability of property violation. This method constructs constraints at all program locations and represents program semantics as fixed points of these constraints, effectively synthesising bounds to approximate the violating probability. Recently, Wang et al. [42] also applies a similar approach to the analysis of Bayesian programs.

Trace abstraction [56, 57], originally developed for non-probabilistic verification, has been adapted for probabilistic contexts. Extensions by Smith et al. [52] use techniques like the union bound [58] to address probabilistic semantics within traces, turning verification into a program synthesis problem. Alternatively, Chen et al. [59] applied trace abstraction for program termination analysis.

Interval-based approaches [26, 50] have also been developed, proving AST of programs and analysing posterior distributions in BSPP. *Probabilistic abstract interpretation* [49, 53, 60] and *pre-expectation calculus* [51, 61] offer more precise analyses within the abstraction framework. However, similar to their non-probabilistic counterparts [62–64], they often require heuristic inputs, such as abstraction domains or probabilistic loop invariants, which complicates full automation.

Exact Methods On the other hand, exact methods aim for precise analysis of probabilistic programs without approximations. These are crucial in scenarios where high reliability is demanded and approximation is insufficient. While universal verification is undecidable, exact analysis can still be achieved by limiting the operations in probabilistic programs to allow for *closed-form* solutions. Techniques like generating functions, used by Zaiser et al. [27], or computer algebraic systems employed by PSI [65] and Hakaru [66, 67], exemplify this approach. Similar strategies are applicable in termination analysis, as demonstrated by Moosbrugger et al. [68].

Besides the approaches discussed, verification via probabilistic models is particularly prevalent and powerful. These models offer a formal framework for representing and analysing probabilistic systems, thereby facilitating the application of rigorous mathematical techniques. We shall now examine in detail the key probabilistic models that underpin many verification techniques, serving as a solid foundation for our research.

1.2 Probabilistic Models

Markov Chains (MCs), particularly in their finite-state, discrete-time form, have become fundamental tools in the realm of probabilistic models. Their underlying principle, which involves state transitions based on the current state, enables their widespread application across various fields including computer science, biology,

economics, and more. The extensive use of MCs over the past two decades has driven research aimed at enhancing their expressiveness.

1.2.1 Finite-State Markov Models

A discrete-time Markov chain is a stochastic model that describes a sequence of possible events, where the probability of each event solely depends on the state achieved in the preceding event. This defining feature, known as the *Markov property*, is key to the model's simplicity and effectiveness. In finite-state MCs, the number of possible states is limited, facilitating a thorough probabilistic analysis of the system's behavior over time.

Applications Across Disciplines (Finite-state) MCs' capability to model diverse stochastic processes renders them invaluable in many domains. In biology, they assist in deciphering genetic sequences and evolutionary patterns [15, 19]. Their application in economics and market analysis is prominent, particularly in modeling consumer behavior patterns [69]. In computer science, MCs are integral in almost every aspect of probabilistic program analysis [48, 52, 70–73].

Extensive Research Beyond their simplicity and real-world applicability, MCs' widespread use is also due to a well-developed toolkit. This toolkit facilitates the resolution of most analysis tasks, leveraging a wide range of established properties that have been thoroughly investigated, such as termination, reachability, and model checking problems [72], with various efficient algorithms developed for these purposes [74–76].

Leveraging these advancements, sophisticated tools like the probabilistic model checker PRISM [73] have been developed. PRISM is renowned for its wide-ranging model handling capabilities and diverse analytical applications [16, 19, 22, 74]. Additionally, the newer model checker Storm [77] offers broader input support and a Python API, enhancing integration with other programs and tasks.

Variants of MCs with additional characteristics have also been developed and studied. Markov Decision Processes (MDPs), which incorporate non-determinism into MCs, play a significant role in probabilistic analysis and verification [73] and have been extensively studied [72, 78]. Hidden Markov Chains [79, 80] and Partially Observed MDPs [81], which introduce state shadowing to MCs and MDPs, respectively, are also two of the intances of further extensions of these models.

1.2.2 Beyond Finite States: Recursive Models

While finite-state MCs stand as robust tools for modelling a wide range of probabilistic scenarios, their inherent limitation to a finite set of states constrains their applicability. Complex systems, especially those characterised by recursive procedures or dynamically evolving structures, demand an extended modelling framework that transcends these boundaries.

An archetypal example of such complexity is the (one-dimensional) unbounded random-walk model. This model is instrumental in understanding diverse phe-

nomena, ranging from diffusion processes in physics to stock price behaviors in finance and natural process progression in ecology [82, 83]. It depicts a scenario where an entity takes steps in random directions, with the potential for an infinite number of steps.

Consider, for instance, the following scenario involving a painter [84, 85]. The painter is tasked with creating a successful work of art, failing which (with probability p) he must produce two additional pieces, each subject to the same contractual terms. This situation raises questions about the probability of the painter fulfilling his contract and the expected number of artworks to be produced.

Evidently, modeling such a scenario (hence also the general random-walk problem) exceeds the capabilities of finite-state MCs due to the potential for an infinite number of steps or states. This characteristic challenges the limits of traditional probabilistic analysis, calling for more sophisticated approaches for accurate representation and study.

In response to these challenges, *Recursive Markov Chains* (RMCs) [86] and *Probabilistic Pushdown Automata* (pPDAs) [87] have emerged as significant advancements. These models allow for the handling of an infinite state space through recursive definitions, providing the necessary expressive power to represent various recursive models found in real-world applications, including the 1-D random-walk example. Further details on these models and their interrelation with our research will be discussed in subsequent chapters.

1.2.3 Probabilistic Higher-Order Verification

In the hierarchy of higher-order grammars [88, 89], the non-random counterpart of a Markov chain is the finite-state automaton, which corresponds to an order-0 grammar, while that of pPDA corresponds to an order-1 grammar.

Recent efforts have focused on extending the entire class of higher-order grammars to probabilistic processes [84, 85]. The intorduction of this extension surely benefits from a careful examination of the foundational principles of higher-order verification. Thus, before delving into probabilistic aspects, it is essential to establish the groundwork by revisiting the theory and practice of higher-order verification.

Higher-Order Model Checking

Functional programming is characterised by treating functions as first-class entities. This paradigm allows functions to be passed as arguments, returned from other functions, and assigned to variables. While this abstraction facilitates a more flexible coding style compared to the usual imperative programs, it also introduces significant challenges in control flow analysis when applying traditional methods from imperative programming.

Higher-Order Recursion Schemes (HORS) are employed [88, 90, 91] to tackle these verification challenges by effectively modelling the intricate control-flow inherent in functional programming. Formally, HORS is a class of higher-order grammars capable of expressing a broad spectrum of languages of strings and

trees. Its order establishes a strict hierarchy of expressiveness, encompassing, as mentioned, the regular languages expressed by finite-state automata at order-0 and the context-free languages by pushdown automata at order-1. Ascending further in the hierarchy, order-2 HORS match the expressiveness of indexed languages [92]. With each increase in order, the ability to model more complex computational processes expands significantly, which underscores the utility of HORS in the context of higher-order verification.

The exploration of model checking such a model initiated the field of higher-order model checking (HoMC), sparked by Ong's seminal work [88]. His demonstration of the decidability of HORS model checking against Monadic Second-Order (MSO) logic established the groundwork for this area. In subsequent years, HoMC has experienced significant advancements. Kobayashi [93] introduced a sound and complete algorithm for HORS model checking against trivial Buchi automata, representing a move towards more practical applications of HoMC with optimisable complexity. Furthermore, Kobayashi and Ong [89] devised an appropriate type system for HORS in relation to MSO logic, enhancing the theoretical framework of the field.

From a practical standpoint, the development of tools such as HorSat, HorSat2 [94], and Preface [95] signifies the evolution of HoMC from theoretical concept to practical software verification toolkit. These tools provide automated verification processes, rendering HoMC more accessible and relevant for real-world functional programming. Tools targeting actual languages, like MoCHI [96] for OCaml, have also emerged, facilitating the application of HoMC techniques in contemporary software development environments. Such tools not only validate functional programs but also help in optimising their performance, thereby enhancing both reliability and efficiency.

Undecidability of Probabilistic HORS

In light of the progress in HoMC and the inherently high expressiveness of HORS, the extension of HORS into the probabilistic realm seemed a promising avenue for largely enhancing the expressiveness of probabilistic models. This topic was recently investigated by Kobayashi et al. in [84, 85].

However, the findings are somewhat disheartening to those expecting a huge leap of expressiveness in probabilistic models – the research demonstrated that the almost sure termination (AST) problem for probabilistic HORS (PHORS) is undecidable. More precisely, they established that, despite PHORS not being (probabilistically) Turing-complete, determining whether a PHORS model terminates with a probability of 1 is not recursively enumerable. Moreover, this undecidability persists even at the level of order-2 PHORS.

The undecidability of the AST property poses a significant barrier to the advancement and application of PHORS as an effective probabilistic model. It is important to note that AST is crucial in the realm of probabilistic verification and analysis. AST, by itself, is a fundamental property among the mostly sought ones in probabilistic systems. For instance, in systems such as BSPP programs [97] and QuickCheck [43], the qualitative attribute of AST underpins their correctness. In

system design tasks [44, 98], AST insights also contribute to enhancing their efficiency and effectiveness. Furthermore, AST is the cornerstone for advanced probabilistic analyses. These analyses include expectation analysis, such as determining PAST, as well as reachability and model checking. All these render AST indispensable for comprehensive model evaluation. In light of this, the undecidability of AST for PHORS highlights a fundamental challenge that must be addressed to enable their broader adoption and practical use.

1.3 Research Overview

Motivated by the preceding results, this research introduces and investigates the *Restricted Probabilistic Tree-Stack Automaton* (rPTSA), a novel probabilistic model. Notably, this model constitutes a non-trivial extension of recursive models, while remaining a proper subclass of the PHORS hierarchy. Moreover, it is incomparable with order-2 PHORS, thereby avoiding the undecidability issues associated with the latter.

This model is a probabilistic extension of the *Restricted Tree-Stack Automaton* (rTSA) [99]. The rTSA model was originally conceived as an automaton representation for multiple context-free languages (MCFG) [99–101], a prominent linguistic formalism aimed at representing Mildly Context-Sensitive Languages (MCSL). As its name suggests, MCFG offers greater expressiveness compared to context-free grammar (CFG) [100]. This enhanced expressiveness is preserved in rPTSA within the probabilistic domain – rPTSA strictly surpasses first-order probabilistic models such as pPDA and RMC in terms of expressiveness.

This extra expressiveness can be illustrated by the following example:

Example 1.1. The language $\{A^n B^m C^n D^m\}$, can be generated by an MCFG [100], and is well-known to be strictly beyond the expressive power of CFG. Moreover, the probabilistic generation of this language cannot be captured by RMC or pPDA. However, rPTSA can model this generation process by simultaneously tracking the two independent counts, n and m, on separate branches of the tree-stack structure. Detailed analysis of this example is provided below in Example 3.7.

Furthermore, rTSA is recognised for its relationship with higher-order grammars, equating to Multiplicative Additive Higher-Order Recursion Schemes (MAHORS) [102]. MAHORS, an extension of HORS, incorporates the affineness constraint from linear logic [103] while enabling product-projection structures to mitigate the stringent constraint. The affineness constraint intuitively stipulates that each function variable is used at most once. The product-projection structures allow limited variable sharing between product components, which has been shown to significantly enhance expressiveness [102]. The MAHORS model has been demonstrated to be more expressive than order-1 HORS and *incomparable* to order-2 HORS.

In this context, we also introduce the *Probabilistic Additive HORS* (PAHORS) as the probabilistic counterpart of MAHORS. We will demonstrate that the

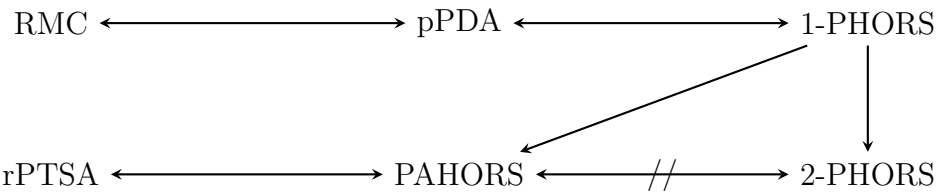

Figure 1.1: Expressiveness Overview

expressiveness of PAHORS as a probabilistic model mirrors its relationship with PHORS and rPTSA in the non-probabilistic context — that PAHORS is equi-expressive to rPTSA, more expressive than order-1 PHORS (corresponding to pPDA / RMC), and *incomparable* to order-2 PHORS. These conditions suggest that the rPTSA and PAHORS models not only extend the boundaries of expressiveness beyond recursive models, but also circumvent the established undecidability of PHORS, as identified by Kobayashi et al. in [85]. Figure 1.1 provides a summary of the expressiveness of our models alongside the introduced models. The arrows in the figure represent the relationships of "is-subsumed-by".

Given their expressive capabilities and connections to various models across distinct fields, rPTSA and PAHORS merit further exploration.

1.4 Contributions

This work primarily addresses the model of rPTSA, exploring potential solutions and discussing their applicability to Probabilistic Additive HORS (PAHORS). Specifically, we focus on and have developed solutions for the following properties of rPTSA:

Termination Analysis As a foundational property, we first investigate the termination analysis of rPTSA. Our research has established the decidability of both the AST (Almost-Sure Termination) and the PAST (Positive Almost-Sure Termination) properties. In particular, we have devised methods to resolve quantitative problems related to both termination probability and the (conditional) expected termination time. The quantitative problem here entails determining whether a given comparison relation holds between a target value and a specified value. These results and methodologies serve as the cornerstone for the subsequent analyses presented below.

Model Checking Our research in model checking spans two primary domains: For linear-time properties, we concentrate on ω-regular and Linear Temporal Logic (LTL) properties. This includes local quantitative model checking, focusing on the quantitative analysis of the probability that runs of a given rPTSA satisfy a specified ω-regular or LTL property. We also extend this to global qualitative model checking, where we aim to identify all configurations of rPTSA that almost surely satisfy a given property.

In the domain of branching-time properties, we acknowledge the undecidability of quantitative model checking, as demonstrated in probabilistic pushdown automata

(pPDA). Our investigation then centres on the qualitative model checking of rPTSA against Probabilistic Computation Tree Logic* (PCTL*).

Probabilistic Tree-Stack Verifier (PTSV) The development of the Probabilistic Tree-Stack Verifier (PTSV) constitutes the practical aspect of our research. PTSV is an efficient tool designed to determine the termination probability and (conditional) expected termination time. Despite the EXPSPACE complexity of these problems, we have designed and implemented a procedure that has demonstrated practical efficiency. The tool has undergone rigorous testing across a broad range of examples to confirm its effectiveness and reliability.

Other Considerations Our research also extends to other significant aspects of rPTSA. We will formally examine the interrelationships between rPTSA and other models such as PAHORS, pPDA, and Stochastic Multiple Context-Free Grammar (SMCFG). More specifically, we will present the mutual conversions between rPTSA and PAHORS, thus transplanting the established algorithms to PAHORS. Besides, the conversions from pPDA and SMCFG to rPTSA are also introduced. Notably, our findings indicate that SMCFG is not equivalent to rPTSA in the probabilistic setting, contrary to the non-probabilistic case.

Furthermore, we will explore the validation of the restriction within rPTSA, which is essential for the model's applicability and robustness.

1.5 Thesis Outline

This section presents an overview of the structure of this paper. In Chapter 2, we delve into the literature, providing a comprehensive introduction to related topics and their corresponding research. In Chapter 3, we define the rPTSA, outline the key properties under investigation, and specify the symbolic notations that will be used.

Chapter 4 focuses on a decomposition characterising rPTSA runs, which serves as the foundation for the algorithms developed in this study. Additionally, this chapter analyses the probability of termination and the (conditional) expected termination time, by instantiating the decomposition principles. In Chapter 5, we describe the linear-time model checking algorithm, while Chapter 6 explores the branching-time model checking algorithm.

Following this, Chapter 7 investigates the restriction property distinguishing rPTSA from being (probabilistically) Turing complete. Subsequently, Chapter 8 scrutinises the connections between rPTSA and other related models, including the mutual conversions with PAHORS. Chapter 9 discusses the tool PTSV, offering an overview of its usage and detailing the efficient algorithm it employs, followed by an empirical evaluation and case studies demonstrating the tool's application.

Finally, in Chapter 10, we provide a brief summary and discuss potential directions for future work.

All our knowledge begins with the senses, proceeds then to the understanding, and ends with reason. There is nothing higher than reason for processing the material of intuition and bringing it under the highest unity of thought. But although all our knowledge begins with experience, it does not follow that it all arises from experience. For it may well be that even our empirical knowledge is a compound of that which we receive through impressions and that which our own faculty of knowledge (sensible impressions merely giving occasion to it) supplies from itself.

— Immanuel Kant, "The Critique of Pure Reason"

2 Literature Review

Contents

In this chapter, we focus on key formalisms that are closely connected to the probabilistic models central to our research. These formalisms are essential as they provide the necessary background for better understanding the properties that will be introduced subsequently.

We begin by discussing recursive probabilistic models, offering a brief introduction to their formalisms, decision procedures for various properties, and related work. Following this, we explore Probabilistic Higher-Order Recursion Schemes (PHORS), an extension of recursive models incorporating higher-order functions, which also serve as a primary motivation for our work, as mentioned in the introduction.

Additionally, we introduce Multiple Context-Free Languages (MCFL) as an important background, particularly as the source of the development of Restricted Tree-Stack Automata (rTSA), the non-probabilistic counterpart of the rPTSA, which is the central formalism of our research. From this, the additional expressiveness of

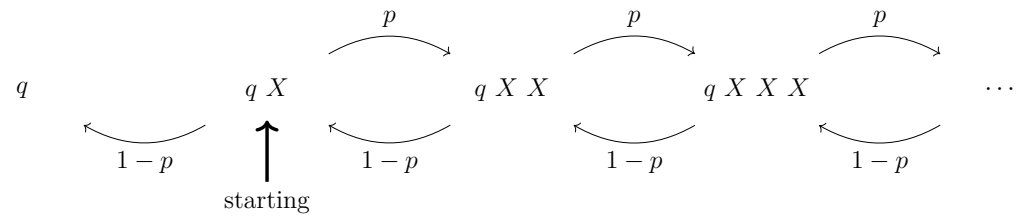

Figure 2.1: Run of Random Walk (Example 2.1)

rPTSA over recursive models can be understood in light of the extra-expressiveness of MCFL compared to pure Context-Free Languages (CFL).

Finally, we provide a brief overview of rTSA and MAHORS as non-probabilistic counterparts to the probabilistic models this research focuses on.

2.1 Recursive Probabilistic Models

In this section, we broadly categorise the recursive probabilistic models into two types: the complete recursive model, exemplified by pPDA and RMC, which is equivalent to order-1 PHORS, and the partial models, which impose certain limitations on the complete models. This section will provide a detailed overview of the complete models of pPDA and RMC, which have been the subject of extensive research over the past two decades. Following this, we will explore several common partial recursive models as well as extensions. Finally, more research on other related topics is discussed.

2.1.1 pPDA and RMC

pPDA The *Probabilistic Pushdown Automaton* (pPDA) emerged as a natural probabilistic extension of pushdown automata [104]. Its first comprehensive algorithmic analysis was presented in [87]. Formally, pPDA is typically represented through a set of rewriting rules, in the form of $q\ X \xrightarrow{p} q'\ X_1\ \ldots\ X_n$. Here, q and q' represent the control states, while X and each X_i (for $i \in \{1, \ldots, n\}$) denote the stack symbols.

Example 2.1 (Random Walk)**.** To illustrate the operation of a pPDA, consider the example of a 1-D random walk. This can be represented using the following rules:

$$q\ X \xrightarrow{p} q\ X\ X \qquad q\ X \xrightarrow{1-p} q$$

The runs of the pPDA are presented in Figure 2.1.

Example 2.2. A more complex example of pPDA, featuring additional control states and stack symbols is displayed in Figure 2.2. In the figure, we abbreviate "$\xrightarrow{1}$" as $\rightarrow$, and this convention will be used throughout without further elaboration.

$$
\begin{array}{lcl}
q_1\ X & \xrightarrow{1/2} & q_2\ X\ Y \\
q_2\ X & \xrightarrow{1/2} & q_2\ Y\ X \\
q_1\ Y & \xrightarrow{1/4} & q_1\ X\ Y \\
q_2\ Y & \rightarrow & q_2
\end{array}
\qquad\qquad
\begin{array}{lcl}
q_1\ X & \xrightarrow{1/2} & q_1 \\
q_2\ X & \xrightarrow{1/2} & q_1\ Y \\
q_1\ Y & \xrightarrow{3/4} & q_2
\end{array}
$$

Figure 2.2: Rules of Example 2.2

$$
\begin{array}{rcl}
q_{(and,init)}\ C & \xrightarrow{0.08} & q_{(or,1)} \\
q_{(and,init)}\ C & \xrightarrow{0.6} & q_{(or,init)}\ C\ C \\
q_{(and,1)}\ C & \xrightarrow{0.2} & q_{(or,1)} \\
q_{(or,init)}\ C & \xrightarrow{0.08} & q_{(and,1)} \\
q_{(or,init)}\ C & \xrightarrow{0.6} & q_{(and,init)}\ C\ C \\
q_{(or,0)}\ C & \xrightarrow{0.2} & q_{(and,0)}
\end{array}
\qquad
\begin{array}{rcl}
q_{(and,init)}\ C & \xrightarrow{0.32} & q_{(or,0)} \\
q_{(and,1)}\ C & \xrightarrow{0.8} & q_{(or,init)}\ C\ C \\
q_{(and,0)}\ C & \rightarrow & q_{(or,0)} \\
q_{(or,init)}\ C & \xrightarrow{0.32} & q_{(and,0)} \\
q_{(or,0)}\ C & \xrightarrow{0.8} & q_{(and,init)}\ C\ C \\
q_{(or,1)}\ C & \rightarrow & q_{(and,1)}
\end{array}
$$

Figure 2.3: And-Or Tree

Example 2.3 (And-Or Tree [105])**.** As a more complex example that closely resembles a real-world application, we consider the "and-or tree" model as introduced in [105]. This example addresses a sophisticated statistical problem that can be effectively represented using pPDA rules, as illustrated in Figure 2.3 [1].

RMC Independently, Etessami and Yannakakis introduced the *recursive Markov chain* (RMC) [86]. They also separately explored the ω-regular model checking problem for RMCs in [106]. The RMC model extends the concept of *recursive state machines* or *unrestricted hierarchical machines* [107, 108] to a probabilistic context. Informally, an RMC comprises one or more *components*, each consisting of entries, exits, internal nodes, and boxes. Each box refers to a component and has corresponding call and return ports mirroring the entries and exits of the referenced component. A typical run of an RMC begins from some node in a component and transits, like in a usual Markov chain, from node to node. Specially, the run may enter some boxes in the component, which then invoke the corresponding component that the box links to and continues within the box until exiting from one of the exits of the linked component, which marks the return from the corresponding box.

Example 2.4. Refer to Figure 2.4 for an RMC example. This RMC includes two components, **A** and **B**. Component **A** contains two entries, three exits, three internal nodes, and two boxes: one linking to **B** and the other to **A** itself. Component **B** has two internal nodes, one entry, one exit, and a box that links back to **B** itself.

In the graph, the Cs and Rs represent call and return ports, respectively, mapping to the specific entries and exits as indicated by their subscripts.

For this example, the behaviour of a typical run could be described as follows. Consider a run starting from the entry En^1_A, which then transits with probability 1

[1] This representation is a specific instantiation of the complex parameters in [105].

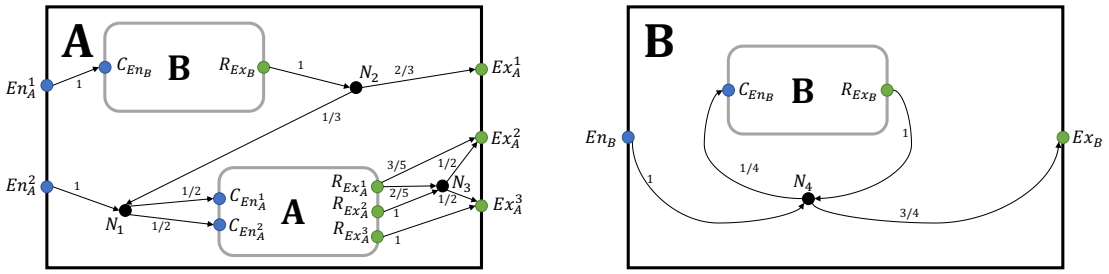

Figure 2.4: RMC of Example 2.4

to the call port C_{En_B} of a box linking to **B**. It then enters the linked component **B** from the corresponding entry En_B. Next, it transits to N_4, where there are two possibilities: a recursive transition to the box in **B** that links to itself with probability 1/4, or exiting to Ex_B with probability 3/4. For simplicity, we consider the latter case. Recall that the run entered component **B** from the linked box in **A**; hence, the run then transits to R_{Ex_B} in **A** and moves deterministically to N_2. At this point, it could either go to N_1 with probability 1/3 or the exit node Ex_A^1 with probability 2/3, where the latter concludes the run as there is no further box to return to.

Equivalence between pPDA and RMC Experts in the field [109, 110] have highlighted the equivalence between RMC and pPDA, for which we here present an overview.

Notably, both pPDA and RMC, along with other (discrete-time) probabilistic models, essentially define Markov chains which may have an infinite number of states. Thus, when discussing equivalence and comparing the expressiveness between these models, it refers to their capacity to express the same class of denumerable Markov chains [86, 109, 110] in the sense that is precisely defined in [109].

Conversion between RMC and pPDA (and vice versa) can be achieved in *linear time* [109], highlighting their practical equivalence. In converting from RMC to pPDA, entries, internal nodes, and boxes are conceptualised as stack symbols, with control states capturing the "return information" indicated by exits.

Example 2.5. An illustration of converting RMC rules to pPDA is shown in Figure 2.5. In this conversion, stack symbols encompass entries (En_A^1, En_A^2, En_B), internal nodes (N_1, N_2, N_3, N_4), and boxes (Box_A, $Box_B^{\mathbf{A}}$, $Box_B^{\mathbf{B}}$). Control states such as q_1, q_2, and q_3 represent the return information from component **A**, with an additional state q representing a general or "normal" scenario.

The conversion process from pPDA to RMC is more nuanced. Initially, it necessitates that the right-hand side (RHS) of pPDA rules contain no more than two stack symbols, a condition that does not limit expressiveness [109, 110]. Then, for a given pPDA, an RMC with a single component is constructed. The entries of this component match pairs of the form (q, X), where q ranges over control states and X spans stack symbols, with exits corresponding to the pPDA's control states. Each box within the component corresponds to a distinct pPDA stack symbol,

$$
\begin{array}{rcl}
q\ En_A^1 & \to & q\ En_B\ Box_B^{\mathbf{A}} \\
q\ Box_B^{\mathbf{A}} & \to & q\ N_2 \\
q\ En_A^2 & \to & q\ N_1 \\
q\ N_1 & \xrightarrow{1/2} & q\ En_A^1\ Box_A \\
q\ N_1 & \xrightarrow{1/2} & q\ En_A^2\ Box_A \\
q_1\ Box_A & \xrightarrow{2/5} & q\ N_3 \\
q_1\ Box_A & \xrightarrow{3/5} & q_2 \\
q_2\ Box_A & \to & q\ N_3 \\
q_3\ Box_A & \to & q_3
\end{array}
\qquad
\begin{array}{rcl}
q\ N_3 & \xrightarrow{1/2} & q_2 \\
q\ N_3 & \xrightarrow{1/2} & q_3 \\
q\ N_2 & \xrightarrow{1/3} & q\ N_1 \\
q\ N_2 & \xrightarrow{2/3} & q_1 \\
q\ En_B & \to & q\ N_4 \\
q\ N_4 & \xrightarrow{1/4} & q\ En_B\ Box_B^{\mathbf{B}} \\
q\ N_4 & \xrightarrow{3/4} & q \\
q\ Box_B^{\mathbf{B}} & \to & q\ N_4
\end{array}
$$

Figure 2.5: Equivalent pPDA Rules for Example 2.4

Equivalent RMC

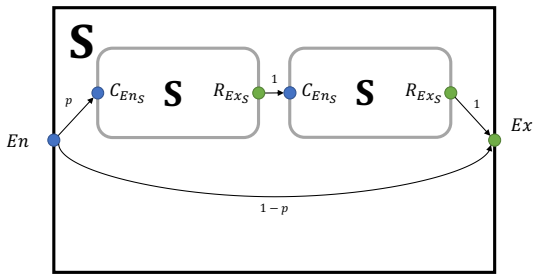

Figure 2.6: of Example 2.1

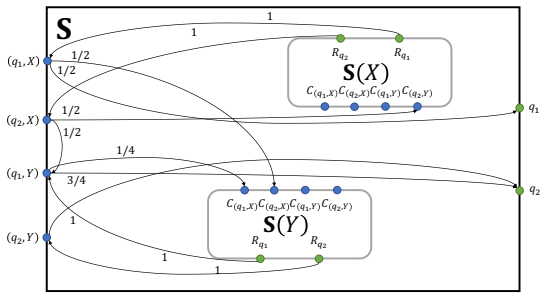

Figure 2.7: of Example 2.2

facilitating transitions between entries, call ports, and exits based on pPDA rules. This setup is demonstrated in the following examples.

Example 2.6 (Equivalent RMCs from pPDA)**.** A simpler illustration of the conversion process is provided by representing the random walk from Example 2.1 as an RMC, as depicted in Figure 2.6. This example showcases how a basic probabilistic model in pPDA format can be seamlessly translated into the RMC framework.

For a more intricate example, the conversion of Example 2.2 into RMC is visualized in Figure 2.7. Following the conversion methodology described previously, the boxes in the RMC are associated with the stack symbols X and Y. In the graph, these are designated by (X) and (Y), respectively, illustrating how complex pPDA rules are mapped into the RMC structure.

2.1.2 Termination Analysis

Termination analysis is pivotal to our project, and our methodology can be viewed as an extension of techniques applied to RMC and pPDA. Therefore, it should be beneficial to further investigate the nuances involved in performing such analyses on RMC and pPDA.

Termination Analysis for RMC The termination analysis for RMC, as outlined in the literature [86, 109], is essentially rooted in the techniques established for Markov chains. This process entails formulating a set of equations by breaking down the termination pathways of a node into those linked with its "geometrically" connected counterparts.

For instance, consider Example 2.4, whose graphical representation is displayed in Figure 2.4. To determine the termination probability from entry En_B of component **B**, with the aim of exiting from Ex_B, one could observe that the probability of transitioning from En_B to Ex_B, denoted as $x_{(En_B,Ex_B)}$, is equal to that from node N_4, represented as $x_{(N_4,Ex_B)}$. Thus, we have $x_{(En_B,Ex_B)} = x_{(N_4,Ex_B)}$. Furthermore, the probability of N_4 terminating in Ex_B is determined as follows:

- A transition to Ex_B with a probability of 3/4 directly results in reaching the target node.
- A transition to the call port C_{En_B} of the box in **B** occurs with a probability of 1/4. After this transition, in order to reach Ex_B, there must first be an exit from the return port R_{Ex_B}, which subsequently reaches the target exit Ex_B. In the procedure, the probability of exiting from R_{Ex_B} upon visiting C_{En_B} is precisely the same as the probability of transitioning from En_B to Ex_B – the quantity of interest at the beginning, denoted as $x_{(En_B,Ex_B)}$. While for the latter probability of reaching the exit from the return port, we denote it as $x_{(R_{Ex_B},Ex_B)}$ and relegate for further exploration.

Resulting from the analysis, there is one more variable remaining unexamined, namely, $x_{(R_{Ex_B},Ex_B)}$. One could observe from the figure easily that it is identical to $x_{(N_4,Ex_B)}$.

To summarise, the above discussion leads to the formation of the following system of equations:

$$\begin{aligned} x_{(En_B,Ex_B)} &= x_{(N_4,Ex_B)} \\ x_{(N_4,Ex_B)} &= 1/4 \cdot x_{(En_B,Ex_B)} x_{(R_{Ex_B},Ex_B)} + 3/4 \\ x_{(R_{Ex_B},Ex_B)} &= x_{(N_4,Ex_B)} \end{aligned} \tag{2.7}$$

The actual probability of termination is described as the *least fixed point* (LFP) of this equation system [86, 109, 110]. Typically, this value is irrational [86, 109, 111], making finite representation challenging. However, the quantitative termination probability p can be ascertained in relation to a given number c such that $p \sim c$, where $\sim$ represents comparison operators within $\{\leq, <, =, \neq, >, \geq\}$.

This analysis is facilitated by the *Existential fragment of the first-order Theory of the Reals* (EToR) [112], which provides a methodological approach to solving such problems.

Theorem 2.8 (EToR [112])**.** *Given a finite set of polynomial equations and inequalities with real coefficients, it is possible to determine whether a solution exists within the real numbers for this set of equations and inequalities. And the decision problem is in PSPACE.*

Termination Analysis for pPDA By converting from pPDA to RMC, the variable $x_{((q,X),q')}$, generated through the aforementioned procedure, then aligns exactly with the variable used in the equation system for pPDA – $\langle qXq' \rangle$. This equivalence highlights a method for *decomposing* the potentially infinite runs of the models into finite *patterns*. Essentially, these variables (either $x_{(N,Ex)}$ or $\langle qXq' \rangle$) encapsulate a set of runs commencing from a current state to a designated exit/popping point, thereby facilitating the analysis of infinite model behaviors through finitely many "patterns" of runs.

Specifically, the AST problem for both pPDA and RMC is in PSPACE [109, 110]. However, to the best of our knowledge, the lower bound of this problem remains unknown. As [109] pointed out, the problem is at least as hard as the square-root-sum problem, the exact complexity of which is a well-known open question.

Expectation Results Interest often extends beyond mere probability values to expectation-related outcomes, such as *expected total accumulated reward*, a concept generalised from *expected termination time* (ETT). A model is considered *positively almost sure terminating* (PAST) if it is AST and exhibits a finite ETT, denoting a high degree of termination certainty within the probabilistic framework that the model has finite expected termination time. Note that AST does not necessarily imply PAST. For instance, in the random walk example (i.e., Example 2.1), when the transition probability is set to $p = 1/2$, the model remains AST. However, the expected termination time is well-known to be infinite [84, 85, 105].

This analysis mirrors the methodology applied in termination probability analysis, with works such as [105, 113, 114] exploring these expectation outcomes. The approach essentially employs a similar decomposition pattern as seen in the termination probability analysis, aiming to dissect and understand complex probabilistic behaviors within these models.

2.1.3 Model Checking Recursive Models

Model Checking Problems The pioneering work of [87] introduced the first model checking algorithm for ω-regular properties and the qualitative fragment of probabilistic computation tree logic (qPCTL) properties within recursive models. The focus in probabilistic verification, both for model checking and termination analyses, frequently shifts between qualitative and quantitative issues. The qualitative aspect explores whether certain properties or termination criteria can be met with absolute certainty (probability 1) or impossibility (probability 0). Conversely, the quantitative perspective, similar to the analysis of termination probabilities, questions the comparative relationship between the probability p of satisfying properties or reaching termination and a specified constant c.

Efficient LTL Model Checking The quest for more efficient LTL model checking for Recursive Markov Chains (RMC) is addressed in [115]. While ω-regular model checking for RMC has clarified the decidability issue, the efficiency of the algorithm, particularly the conversion from LTL to Büchi automata or equivalent ω-regular models, remains a challenge [116]. It has been identified that solving for the

	One-Exit RMC / pBPA	RMC / pPDA
quantitative DRA	in PSPACE [110]	in PSPACE [110]
qualitative DRA	in P [110]	in PSPACE [110]
quantitative LTL	EXPTIME-Complete [110]	EXPTIME-Complete [110]
qualitative LTL	EXPTIME-Complete [110]	EXPTIME-Complete [110]
fixed formula qualitative LTL	in P [110, 115]	in PSPACE [115]
fixed formula quantitative LTL	in PSPACE [115]	in PSPACE [115]
PCTL	UNKNOWN [110, 120]	Undecidable [110]
PCTL*	Undecidable [110]	Undecidable [110]
qPCTL	EXPTIME-Complete [120]	EXPTIME-Complete
qPCTL*	2-EXPTIME-Complete [110]	2-EXPTIME-Complete [110]
fixed formula qPCTL	P-Complete [120]	EXPTIME-Complete [120]
fixed formula qPCTL*	P-Complete [120]	EXPTIME-Complete [120]

Table 2.1: Model Checking Results of RMC / pPDA

deterministic Rabin automaton (DRA) model checking presents a lower complexity than LTL. An EXPTIME-complete result for LTL model checking against RMC is documented in [115].

Branching-Time Model Checking The early investigations into the decidability of qualitative branching-time model checking of pPDA against PCTL* laid foundational knowledge [87]. Subsequent work broadened the scope to the pPDA model checking problem against the branching-time logic, PCTL(*), revealing undecidability for PCTL properties, while qualitative fragments of PCTL* was shown to be decidable with established bounds [117].

Further refinement and a lower complexity bound were achieved in the PhD thesis of Bradzil [118], inspired by the insights from [115]. Comprehensive complexity analyses of branching-time model checking pPDA were later consolidated [119].

Summaries Subsequent summaries for RMC [111] and pPDA [110] revisited proofs for branching-time logics [120], enriching the body of knowledge on model checking within these domains.

To sum up, the model checking results for RMC and pPDA are presented in Table 2.1. For the ω-regular properties, there are many presentation models [121, 122], such as *nondeterministic Büchi automaton* (NBA) [123], *Muller automaton* [124], etc. Here, we choose the *deterministic Rabin automaton* as the presentation model. Note that, as mentioned above, although in terms of expressiveness, LTL is subsumed by DRA, the conversion from LTL to DRA is itself challenging [116, 125, 126].

2.1.4 Partial Recursive Models and Extensions

This section further discusses advancements in partial recursive probabilistic models, alongside the extensions to the complete models previously described.

pOC and QBD The *Probabilistic One-Counter Automaton* (pOC) represents a specific subset within the recursive model spectrum [105, 110]. This model is characterised by its use of a single stack symbol, making it a simplified version of the pPDA. Both Example 2.3 and Example 2.1 fall under this model category. Efficient algorithms for analysing termination probabilities and expectations for pOC have been outlined [105].

In a related context, the *Quasi-Birth-Death process* (QBD) and its variant, the *Tree-Like* QBD, have been independently explored [127–131]. For these models, the model of QBD is equivalent to pOC, while the Tree-Like QBD has been demonstrated to hold equivalence to more comprehensive models such as RMC and pPDA [132].

pBPA and One-exit RMC The *Probabilistic Basic Process Algebra* (pBPA) has garnered attention for its applicability in various realistic scenarios [133–135]. pBPA is derived from pPDA by applying a single-control-state constraint, exemplified by the random walk in Example 2.1.

The corresponding sub-class within the RMC framework is the *one-exit RMC*, characterised by its limitation to a single exit point per component [109].

Formally, the *Stochastic Context-Free Grammar* (SCFG) aligns precisely with the structure of pBPA, where its rules mirror the single-control-state format of pBPA, which has the following form:

$$X \xrightarrow{p} X_1 \dots X_n$$

This alignment underscores the expressiveness subsumption of SCFG by pPDA, contrary to the non-probabilistic equivalence of CFG and PDA in terms of string languages.

Moreover, the resemblance between the equation systems of pPDA and pBPA suggests a close relationship in their runtime behaviours, allowing for the approximation of pPDA's behaviour through pBPA. This connection has been rigorously defined, facilitating the simplification of proofs for pPDA [110, 114].

Extensions to RMC Efforts to extend the foundational RMC model have introduced constructs such as control (non-determinism) and games, leading to the development of *Recursive Markov Decision Processes* and *Recursive Stochastic Games*. These enhanced models offer new dimensions to the study of recursive probabilistic models [136, 137].

2.1.5 More Related Research

Equation System Analysis The equation system, integral to the analysis of various properties of recursive models, as discussed in termination analysis for RMC and pPDA, plays a critical role in both the analytical frameworks of these recursive

models and our research. The examination of non-negative polynomial equation systems generated from these models is crucial.

Initial research into the generated equation systems was undertaken in [86], coinciding with the introduction of RMC. This work explored fundamental properties of the equation systems and proposed several straightforward yet effective methods for efficiently determining the least fixed point (LFP) of these systems.

Subsequently, in [138], the focus was extended towards efficient solutions for obtaining the LFP of a more general type of equation systems – namely, the *Monotone Systems of Polynomial Equations* (MSPEs). The study demonstrated that iteration via Newton's method can efficiently converge, especially when the MSPE is strongly connected, ensuring that the number of iterations needed to achieve a specified bit precision increases linearly. This finding provides a theoretical foundation for the precision of approximation results, offering a significant advancement over previous empirical-only outcomes.

While the aforementioned methods focus on the under-approximation of the LFP, research has also extended to over-approximation methods. In [139], a technique for efficiently obtaining *inductive upper bounds* for equation systems was introduced.

Implementations Several implementations have emerged, translating the theoretical frameworks into practical analysis tools.

PReMo [140] As of now, PReMo stands as a unique implementation focusing on the termination analysis of RMC, including its extensions with game and control mechanisms. This tool encapsulates algorithms for analysing both termination probabilities and expected termination times within the RMC framework, highlighting its comprehensive approach to model analysis.

pOC Analysis Implementation The algorithms proposed for efficient pOC analysis in [105] have been implemented, with tests conducted on various parameter settings within the and-or tree example (Figure 2.3).

[139] Implementation The methodology introduced in [139] has been realised through an implementation that has undergone testing across a broad spectrum of examples. This implementation validates the effectiveness of the proposed upper bound approximation method.

2.2 PHORS

The Higher-Order Recursion Schemes (HORS) family represents a highly expressive class of models, originally utilised for modelling higher-order control flows [89, 93, 141]. Essentially, the model is a call-by-name, simply-typed λ-calculus that incorporates recursion and uninterpreted symbols, referred to as terminals. Through iterative reduction and rewriting, this model functions as a generator of strings, trees, or as a pure process, which is not employed to produce structured formalisms.

Kobayashi et al. [84, 85] proposed an extension of the HORS model to include probability, resulting in the probabilistic HORS model (PHORS). In the PHORS

model, treated as a generator of pure stochastic processes, terminals are simplified to a single symbol representing termination, and each rule is assigned a transition probability. We here present a brief account for the PHORS model.

Type of PHORS The concept of types θ is foundational to defining the models, expressed as:

$$\theta := o \mid \theta_1 \to \theta_2$$

Here, o denotes the base type. The function types constructed by "$\to$" are usually written in a right-associative manner. The order of a type θ is determined by:

$$\begin{aligned} \texttt{ord}(o) &:= 0 \\ \texttt{ord}(\theta_1 \to \theta_2) &:= \max\{\texttt{ord}(\theta_1) + 1, \texttt{ord}(\theta_2)\} \end{aligned}$$

PHORS A probabilistic higher-order recursion scheme (PHORS) is defined as a tuple $(\mathcal{N}, \mathcal{R}, S)$, where: $\mathcal{N}$ maps a finite set of non-terminals to their respective types; $\mathcal{R}$ represents the rewriting relation, comprising elements of the form:

$$F\ x_1\ \ldots\ x_n \xrightarrow{p} t$$

Here, $F : \theta_1 \to \cdots \to \theta_n \to o \in \mathcal{N}$, $p \in (0, 1)$ denotes the probability, and t is a λ-term devoid of λ-abstraction. The term's symbols are drawn from $\mathrm{dom}(\mathcal{N}) \cup \{\top\} \cup \{x_1, \ldots, x_n\}$ and $\mathcal{E}' \vdash t : o$, where the typing context $\mathcal{E}'$ is defined as:

$$\mathcal{N} \cup \{x_1 \mapsto \theta_1, \ldots, x_n \mapsto \theta_n\} \cup \{\top \mapsto o\}$$

The standard typing context $\mathcal{E} \vdash t : \theta$ is as follows:

$$\frac{\mathcal{E}(x) = \theta}{\mathcal{E} \vdash x : \theta} \qquad \frac{\mathcal{E} \vdash t_1 : \theta' \to \theta \quad \mathcal{E} \vdash t_2 : \theta'}{\mathcal{E} \vdash t_1\ t_2 : \theta}$$

Lastly, $S : o \in \mathcal{N}$ serves as the starting non-terminal.

Termination The rewriting relation $\xrightarrow[r]{p}$, defined for a rule $r \in \mathcal{R}$ in PHORS, follows the rule:

$$\frac{r = (F\ x_1\ \ldots\ x_n \xrightarrow{p} t) \in \mathcal{R}}{F\ t_1\ \ldots\ t_n \xrightarrow[r]{p} t[t_1/x_1, \ldots, t_n/x_n]}$$

The transitive and reflexive closure of $\xrightarrow[r]{p}$ is denoted by $\overset{p}{\underset{r}{\Rightarrow}}$.

The termination probability of a PHORS is calculated as:

$$\sum_{\{(s,p)\ \mid\ s \in \mathcal{R}^*, S \overset{p}{\underset{s}{\Rightarrow}} \top\}} p$$

$$
\begin{array}{rcl}
S & \to & F_{(q_1,X)}\ \top\ \top \\
F_{(q_1,X)}\ x_{q_1}\ x_{q_2} & \xrightarrow{1/2} & F_{(q_2,X)}\ \left(F_{(q_1,Y)}\ x_{q_1}\ x_{q_2}\right)\ \left(F_{(q_2,Y)}\ x_{q_1}\ x_{q_2}\right) \\
F_{(q_1,X)}\ x_{q_1}\ x_{q_2} & \xrightarrow{1/2} & x_{q_1} \\
F_{(q_2,X)}\ x_{q_1}\ x_{q_2} & \xrightarrow{1/2} & F_{(q_2,Y)}\ \left(F_{(q_1,X)}\ x_{q_1}\ x_{q_2}\right)\ \left(F_{(q_2,X)}\ x_{q_1}\ x_{q_2}\right) \\
F_{(q_2,X)}\ x_{q_1}\ x_{q_2} & \xrightarrow{1/2} & F_{(q_1,Y)}\ x_{q_1}\ x_{q_2} \\
F_{(q_1,Y)}\ x_{q_1}\ x_{q_2} & \xrightarrow{1/4} & F_{(q_1,X)}\ \left(F_{(q_1,Y)}\ x_{q_1}\ x_{q_2}\right)\ \left(F_{(q_2,Y)}\ x_{q_1}\ x_{q_2}\right) \\
F_{(q_1,Y)}\ x_{q_1}\ x_{q_2} & \xrightarrow{3/4} & x_{q_2} \\
F_{(q_2,Y)}\ x_{q_1}\ x_{q_2} & \to & x_{q_2}
\end{array}
$$

Figure 2.8: 1-PHORS Equivalence to Example 2.2

Order-1 PHORS and pPDA An order-n PHORS is defined such that the maximum order of the non-terminal types in the PHORS does not exceed n. Specifically, the family of order-1 PHORS is as expressive as RMC and pPDA.

Consider the following example for illustration:

Example 2.9. The equivalence of a 1-PHORS to the pPDA example provided in Example 2.2 is depicted in Figure 2.8. Here, the type of the starting non-terminal S is o, and the types of $F_{(q_1,X)}$, $F_{(q_1,Y)}$, $F_{(q_2,X)}$, and $F_{(q_2,Y)}$ are all $o \to o \to o$ [2].

This simulation from pPDA to 1-PHORS employs terms to precisely represent the corresponding control state and stack symbol. Each non-terminal $F_{(q,X)}$ in PHORS denotes the control state and the top stack symbol, with a type of $\underbrace{o \to \cdots \to o}_{n} \to o$, where n is the number of control states. Assuming the control states of the pPDA are $\{q_1, \ldots, q_n\}$, the argument at position i signifies the continuation after popping the current stack symbol, transitioning to control state q_i.

This method draws from the conversion of 1-Tree-Stack Automata (1-TSA) to order-1 Multiplicative Additive HORS (MAHORS) [142]. The conversion from 1-PHORS to pPDA is more complex and involves *game semantics*, which will be briefly discussed below in Chapter 8, when the relationship between Restricted Probabilistic Tree Stack Automata (rPTSA), the main object of this work, and Probabilistic Additive HORS (PAHORS), a limited version of PHORS, is addressed.

The PHORS Hierarchy As established, order-1 PHORS corresponds exactly with probabilistic recursive models such as pPDA and RMC. Similar to non-probabilistic systems [91, 143], PHORS exhibits a well-defined hierarchy based on model order [84, 85]. In this hierarchy, an order-$(n+1)$ PHORS model possesses strictly greater expressive power than an order-n model.

Remark 2.10 (Expressiveness Inheritance)**.** The strictness of the PHORS hierarchy can be directly traced to the properties of non-probabilistic models.

[2] Following the basic functional convention, the operator $\to$ is right-associative, meaning it is equivalent to $o \to (o \to o)$.

To illustrate, if two classes of model are capable of representing the same set of probabilistic models, then one class should be able to reproduce the probabilistic string distributions generated by the other. Consider a language L representable by an order-$(n+1)$ HORS but not by an order-n HORS. The existence of such a language is supported by the established hierarchy in string language generation capabilities [88, 89]. By assigning non-zero probabilities to the transitions within the HORS generating L, an order-$(n+1)$ PHORS can be constructed as a probabilistic string generator. Since L cannot be represented by an order-n HORS, no order-n PHORS can match the distribution's support set for L, demonstrating that the distribution exceeds the expressive capacity of order-n models. This argument solidifies the notion that the strictness observed in the non-probabilistic hierarchy is inherited by the probabilistic case.

In essence, if two models differ in their ability to generate string languages, this difference will be reflected in their probabilistic extensions. Nonetheless, generating the same class of string languages does not imply equivalence in the probabilistic realm – as seen in the case of SCFG and pPDA [86, 109, 110], where SCFG corresponds only to pBPA, a subset of pPDA.

Termination Analysis The work by Kobayashi et al. [84, 85] addresses termination issues in PHORS models, especially the Almost Sure Termination (AST) problem. Their findings reveal that undecidability in termination emerges starting from order-2. Termination analysis is a critical component in the study of probabilistic models [50, 109, 110], impacting model checking and reachability analyses that depend on decidable termination properties.

The undecidability proof for termination in PHORS, as detailed in [84, 85], uses a reduction from Hilbert's 10th problem [144]. This method marks a departure from traditional techniques employed in undecidability proofs for quantitative model checking of branching-time logics in pPDA [110, 120].

2.3 Multiple Context-Free Languages

To introduce the background of the Restricted Tree-Stack Automaton (rTSA) more effectively, it should be helpful first to understand its linguistic origins – specifically, the *mildly context-sensitive languages* (MCSL) and the *multiple context-free grammars* (MCFGs), which was intended to be proposed as an appropriate representation for this class of language.

2.3.1 Mildly Context-Sensitive Languages

The quest to formalise human languages began with the seminal works of Chomsky [145, 146]. However, the hierarchy Chomsky proposed [147] was later deemed insufficiently nuanced to furnish the mathematical tools needed for language analysis. Researchers found the context-free (type-2) languages in this hierarchy to be too limited [148], while context-sensitive (type-1) languages were considered too

permissive, potentially allowing the generation of pathological constructs and offering scant analytical guidance [149, 150].

This discrepancy spurred efforts to identify grammars that are adequate for accurately capturing human languages. Among these, the concept of *mildly context-sensitive languages* (MCSL) has been particularly influential [149, 151, 152].

Weir [151] initially proposed the MCSL concept and its objectives. Although the term "adequate" remains somewhat ambiguous, leading to ongoing debates, certain characteristics are widely accepted as fundamental to MCSL [152, 153]: *cross-serial dependencies*, *constant growth*, and *polynomial parsing*. The first two criteria are theoretical, while the third has practical implications.

Cross-serial dependencies lack a universally agreed-upon definition but generally refer to the ability to handle strings where dependencies between words intersect. The COPY language, illustrated in Example 2.14, is a well-known example.

Constant growth is defined as the property wherein the length difference between adjacent words is bounded by a constant. The exponential language, such as $\{a^{2^n}\}$, serves as a notable counterexample. The requirement for "polynomial parsing" is self-explanatory.

To identify the most fitting generative grammar, numerous refined models have been proposed from various perspectives, including (MC)TAG [152, 154], HPSG [155], MG [156], and LIG [157], etc.

2.3.2 Multiple Context-Free Grammar

In the exploration of grammars capable of representing mildly context-sensitive languages (MCSL), *Multiple Context-Free Grammar* (MCFG) stands out as a particularly significant development. The languages expressible through MCFG are hence known as *Multiple Context-Free Languages* (MCFLs).

In simple terms, MCFG rewriting rules take the form:

$$F(\alpha_1, \ldots, \alpha_n) \leftarrow F_1(x_1^{(1)}, \ldots, x_{n_1}^{(1)}), \ldots, F_m(x_1^{(m)}, \ldots, x_{n_m}^{(m)})$$

Here, F and F_i (for $i \in \{1, \ldots, m\}$) function as non-terminals, akin to those in context-free grammars. The *rank* of a non-terminal, denoted by n for F and n_i for each F_i, signifies the number of arguments it takes. Each variable, $x_i^{(j)}$, serves as a placeholder for a string segment corresponding to the non-terminal. The terms α_i (for $i \in \{1, \ldots, n\}$) consist of sequences of either terminals or variables from the right-hand side (RHS). The maximum rank of non-terminals is also referred to as the rank of the overall MCFG. Let it be k, we usually write k-MCFG to make the rank explicit.

Crucially, MCFG mandates that each variable appear on the left-hand side (LHS) no more than once, a condition essential for distinguishing MCFG from the more expressive *Parallel MCFG* (PMCFG).

This framework essentially inverts traditional context-free grammar (CFG) rules, extending them to support multiple ranks. For instance, a standard CFG rule with terminals `a` and `b`, and non-terminals F and G:

$$F \rightarrow \mathtt{a}G\mathtt{b}$$

is expressed in MCFG-style rule as:

$$F(\mathtt{a}x\mathtt{b}) \leftarrow G(x)$$

This way of presenting the grammar is also called a *bottom-up view*, which was first presented by Kanazawa [158].

Accordingly, all CFG rules can be reformulated into MCFG rules with a rank of 1. Hence, let the languages expressible by n-MCFG be called the n-MCFL, standing for Rank-n Multiple Context-Free Language, and the languages expressible in CFG be CFL, we have:

Theorem 2.11 ([100])**.** *CFG and* 1*-MCFG are equi-expressive.*

Moreover, string derivation in MCFG is visualised through proof trees. The relationship between such trees and the reduction sequences in CFG could be seen in the below example. Note that at each derivation step, the RHS is fully resolved into strings before substituting these strings for their corresponding variables on the LHS.

Example 2.12. The following CFG generating the Dyck language:

$$S \to [S]S \mid \varepsilon$$

can be written in a bottom-up style MCFG rules as:

$$S([x]y) \leftarrow S(x), S(y) \qquad\qquad S(\varepsilon) \leftarrow \varepsilon$$

The rewriting sequence with CFG rules:

$$S \to [S]S \to [[S]S]S \to [[]S]S \to [[]]S \to [[]][S]S \to [[]][]S \to [[]][]$$

can be written as this derivation tree:

$$\dfrac{S([x]y) \leftarrow \varepsilon \qquad \dfrac{\dfrac{S(\varepsilon) \leftarrow \varepsilon}{S(x \leftarrow \varepsilon)} \quad \dfrac{S(\varepsilon) \leftarrow \varepsilon}{S(y \leftarrow \varepsilon)}}{S(x \leftarrow [])} \qquad \dfrac{S([x]y) \leftarrow \varepsilon \qquad \dfrac{\dfrac{S(\varepsilon) \leftarrow \varepsilon}{S(x \leftarrow \varepsilon)} \quad \dfrac{S(\varepsilon) \leftarrow \varepsilon}{S(y \leftarrow \varepsilon)}}{S(y \leftarrow [])}}{S([[]][])}$$

Furthermore, the rank in MCFG forms a strict hierarchy of expressiveness. That is:

Theorem 2.13 ([100])**.** *For every* $n \geq 1$*,* n*-MCFL is a strict subset of* $(n+1)$*-MCFL.*

In particular, this theorem then shows the ability of MCFG to capture languages beyond the scope of CFG through the utilisation of multiple ranks.

Example 2.14. The language COPY is often employed to demonstrate the superior expressive power of MCFG over CFG. This language consists of strings of the form

$$\dfrac{\dfrac{}{F(\varepsilon,\varepsilon)\leftarrow\varepsilon}}{\dfrac{F(\mathtt{a}x\mathtt{b},\mathtt{c}y)\leftarrow \quad F(x\leftarrow\varepsilon,y\leftarrow\varepsilon)}{F(x\leftarrow\mathtt{ab},y\leftarrow\mathtt{c})}}$$
$$\dfrac{S(xy)\leftarrow \qquad F(x\leftarrow\mathtt{aabb},y\leftarrow\mathtt{cc})}{S(\mathtt{aabbcc})}$$

Figure 2.9: Deriving Tree of `aabbcc` of Example 2.15

"ww", where w is a word in the regular language $(\mathtt{a}|\mathtt{b})^*$. The following is a 2-MCFG representing this language:

$$\begin{aligned} S(xy) &\leftarrow F(x,y) \\ F(\mathtt{a}x,\mathtt{a}y) &\leftarrow F(x,y) \\ F(\mathtt{b}x,\mathtt{b}y) &\leftarrow F(x,y) \\ F(\varepsilon,\varepsilon) &\leftarrow \varepsilon \end{aligned}$$

Besides, common examples showcasing the extra-expressiveness of MCFG over CFG also include the following.

Example 2.15. Consider the language $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n\}$, which can be described by a 2-MCFG but not by a CFG [100]:

$$\begin{aligned} S(xy) &\leftarrow F(x,y) \\ F(\mathtt{a}x\mathtt{b},\mathtt{c}y) &\leftarrow F(x,y) \\ F(\varepsilon,\varepsilon) &\leftarrow \varepsilon \end{aligned}$$

Here, S is the *starting non-terminal* with a rank of 1, reflecting the requirement for a singular resultant string.

As an example of derivation, the string `aabbcc` can be derived by the proof tree shown in Figure 2.9.

Example 2.16. The language $\left\{\mathtt{a}_1^n\dots\mathtt{a}_{2m+1}^n\right\}$ can be generated by $(m+1)$-MCFG as the following:

$$\begin{aligned} S(x_1\dots x_{m+1}) &\leftarrow F(x_1,\dots,x_{m+1}) \\ F(\mathtt{a}_1x_1\mathtt{a}_2,\dots,\mathtt{a}_{2i-1}x_i\mathtt{a}_{2i},\dots,\mathtt{a}_{2m+1}x_{m+1}) &\leftarrow F(x_1,\dots,x_{m+1}) \\ F(\underbrace{\varepsilon,\dots,\varepsilon}_{m+1}) &\leftarrow \varepsilon \end{aligned}$$

Free Word Order

A notable discovery regarding the class of MCFG is its inclusion of the class of languages with free word order [159–163].

Free word order, as suggested by its name, refers to the freedom in the order of words forming a sentence. In natural language, this phenomenon is well-known in Polish [164], Hungarian [165], Latin [165, 166], and others.

In linguistics, the language of $\mathbf{MIX}_3$ is a well-known extreme example of absolute free word order. This language is formed by an alphabet with three elements, namely, a, b, and c. A valid word in the language is any string that contains the same number of a, b, and c.

The language of $\mathbf{O}_2$ has also attracted much attention as a (rationally) equivalent language to $\mathbf{MIX}_3$ [160]. In this language, the alphabet consists of four elements, $\{a_1, a_2, \overline{a_1}, \overline{a_2}\}$. To be a valid word in the language, the number of a_i and $\overline{a_i}$ for $i \in \{1, 2\}$ must be the same. Moreover, it has been shown that these two languages can be represented by the class of 2-MCFG [160, 161].

Example 2.17 ($\mathbf{O}_2$[167])**.** The language of $\mathbf{O}_2$ is represented in the following 2-MCFG.

$$
\begin{aligned}
\mathsf{S}(xy) &\leftarrow \mathsf{Inv}(x, y) \\
\mathsf{Inv}(x_1y_1x_2y_2, \varepsilon) &\leftarrow \mathsf{Inv}(x_1, x_2), \mathsf{Inv}(y_1, y_2) \\
\mathsf{Inv}(x_1y_1x_2, y_2) &\leftarrow \mathsf{Inv}(x_1, x_2), \mathsf{Inv}(y_1, y_2) \\
\mathsf{Inv}(x_1y_1, x_2y_2) &\leftarrow \mathsf{Inv}(x_1, x_2), \mathsf{Inv}(y_1, y_2) \\
\mathsf{Inv}(x_1, y_1x_2y_2) &\leftarrow \mathsf{Inv}(x_1, x_2), \mathsf{Inv}(y_1, y_2) \\
\mathsf{Inv}(\varepsilon, x_1y_1x_2y_2) &\leftarrow \mathsf{Inv}(x_1, x_2), \mathsf{Inv}(y_1, y_2) \\
\mathsf{Inv}(\theta x_1\overline{\theta}, x_2) &\leftarrow \mathsf{Inv}(x_1, x_2) \\
\mathsf{Inv}(\theta x_1, \overline{\theta}x_2) &\leftarrow \mathsf{Inv}(x_1, x_2) \\
\mathsf{Inv}(\theta x_1, x_2\overline{\theta}) &\leftarrow \mathsf{Inv}(x_1, x_2) \\
\mathsf{Inv}(x_1\theta, \overline{\theta}x_2) &\leftarrow \mathsf{Inv}(x_1, x_2) \\
\mathsf{Inv}(x_1\theta, x_2\overline{\theta}) &\leftarrow \mathsf{Inv}(x_1, x_2) \\
\mathsf{Inv}(x_1, \theta x_2\overline{\theta}) &\leftarrow \mathsf{Inv}(x_1, x_2) \\
\mathsf{Inv}(\varepsilon, \varepsilon) &\leftarrow \varepsilon
\end{aligned}
$$

where θ ranges over $\{a_1, a_2, \overline{a_1}, \overline{a_2}\}$ and $\overline{\overline{a_i}} = a_i$ with $i \in \{1, 2\}$.

Furthermore, the more general forms of $\mathbf{MIX}_3$ and $\mathbf{O}_2$ are also shown to be in MCFG [159, 167]—the $\mathbf{MIX}_n$ and $\mathbf{O}_n$ languages, respectively. In $\mathbf{MIX}_n$, the size of the alphabet is n rather than just 3, and in $\mathbf{O}_n$, the alphabet becomes $\{a_1, \ldots, a_n, \overline{a_1}, \ldots, \overline{a_n}\}$. The requirements are direct extensions of the original ones.

This fact is significant as it implies that the class of MCFL is closed under the operation of permutation closure. This follows as a corollary of an old prophecy [168], stating that if $\mathbf{O}_n$ is MCFL for any n, then MCFL is closed under permutation.

2.3.3 MCFL Hierarchy

The study of MCFG and its variants has revealed a fascinating hierarchy of expressiveness, depicted in Figure 2.10. This hierarchy demonstrates that n-MCFLs

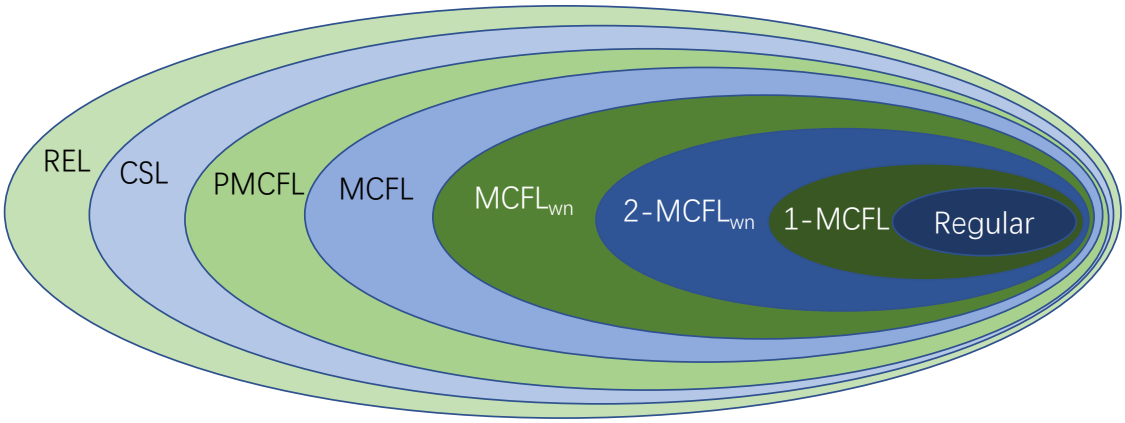

Figure 2.10: The MCFL Hierarchy

create a strict sequence in terms of expressiveness, where n denotes the *highest* rank of non-terminals used within a grammar.

Furthermore, the class MCFL$_{wn}$ represents a subset of MCFL characterized by the *well-nested* constraint [169, 170]. This subset has been the focus of significant research [169, 171], intersecting with several notable models.

As a particular subset, 2-MCFL$_{wn}$ emerges as a natural subclass within the languages generated by MCFG. The former itself is extensively analysed and paralleled by established frameworks such as TAG [154], Head Grammar (HG) [172], Linear Indexed Grammar (LIG), and Coupled Context-Free Grammar (CCG) [173].

Moreover, MCFG itself aligns with a variety of extensively studied models, including Multiple Context-Free Tree Adjoining Grammar (MCTAG), Hyperedge Replacement (HR) grammars for string generation [174], the languages generated by Deterministic Tree Walking Transducers (DTWT) [175–177], the yield language of finite copying tree transducers over regular trees (yDT$_{fc}$(REGT)) [178], Linear Unification-Based Syntactic Categorial Grammar (LUSCG) [179], and Second-order Abstract Categorial Grammar (ACG) with arity up to 4.

Going further towards expressiveness, MCFL is encompassed by *Parallel* MCFL (PMCFL), which is further subsumed by the broader and standard hierarchy – starting with *Context-Sensitive Languages* (CSL) and, more generally, *Recursively Enumerable Languages* (REL).

From the discussion, one may notice that, although initially conceptualised to capture facets of human languages [100, 101, 180], the MCFL hierarchy, which was not directly aimed at verification, intrinsically suggests a rich vein of models that could inspire further developments in the field. This hierarchy underscores the well-defined nature and linguistic expressiveness of MCFL, alongside a breadth of models predominantly encompassed by or related to MCFL/MCFG, potentially offering new insights into verification challenges.

Relation with HORS Hierarchy The MCFL and HORS hierarchies exhibit a close relationship. Specifically, MCFG is encompassed by third-order HORS when viewed as string generators and stands incomparable with second-order HORS.

When considering pure processes, MCFG also falls within the HORS hierarchy. Nonetheless, these hierarchies offer distinct perspectives on recursive (context-free) models, with the MCFL hierarchy providing a more detailed landscape of sub-models and equivalents.

2.4 rTSA and MAHORS

Finally, we present a brief introduction to Restricted Tree-Stack Automata (rTSA) and Multiplicative Additive HORS (MAHORS), which are the non-probabilistic counterparts of rPTSA and PAHORS, the probabilistic models that form the core of our research, respectively.

rTSA

As MCFG continues to garner attention in the field, where *Finite-State Automata* represent regular languages and PDAs handle CFGs, there has been a proposal to characterise MCFG through a specific type of automaton. This proposal is known as the *Restricted Tree-Stack Automaton* (rTSA) [99]. The TSA, which extends the concept of the PDA (pushdown automaton), generalises the idea of a linear stacks into trees, thus earning its name.

This formalism intuitively views each stack symbol as a proper node within a tree, storing information at each node. A full illustration of this concept will be provided below in Section 3.2, and its complete formalism will be introduced in Definition 8.24.

Importantly, the full TSA model is Turing-complete – it is capable of simulating a Turing machine using just a single stack symbol and branch. However, to align TSA with MCFG, a restriction is necessary: the number of times a node can be revisited from below must be limited by a given finite number. These details will be discussed further in the following sections, where we will focus on the *Restricted Probabilistic Tree-Stack Automaton* (rPTSA), the central topic of our research.

MAHORS Conversely, the emergence of *Multiplicative HORS* (MHORS) and *Multiplicative Additive HORS* (MAHORS), as HORS sub-classes, responds to the trend towards resource-aware computation and linear logic [181–184]. The MHORS model introduces an *affine* constraint [182] to HORS, mandating a one-time use for each parameter in the rewriting process, a constraint deemed too restrictive for broad applicability.

MAHORS, seeking to mitigate this, incorporates pairing and projection constructs [181, 185], enriching expressiveness. The types in MAHORS are defined as:

$$\theta := o \mid \theta \multimap \theta \mid \theta \& \theta$$

Here, $\multimap$ signifies linearity in contrast to the traditional $\rightarrow$, and the product type constructor & facilitates parameter sharing across terms.

Accommodating the product type &, MAHORS extends the λ-calculus with projection $\pi_i t$ (for $i \in \{1, 2\}$) and pairing $\langle t_1, t_2 \rangle$. In $\langle t_1, t_2 \rangle$, the types $\theta_1 \& \theta_2$ enable shared parameter usage between t_1 and t_2, with $\pi_i t$ selecting the term for execution.

This section briefly contrasts the type and term frameworks of MAHORS with HORS. These distinctions will be further explored in the context of its probabilistic counterpart, Probabilistic Additive HORS (PAHORS), in forthcoming discussions in Chapter 8.

Expressiveness of MAHORS and rTSA

Recent investigations [142] into MHORS and MAHORS expressiveness reveal that the strict affineness significantly limits expressiveness, equating MHORS's tree-generating capacity to that of merely *regular* trees. Conversely, MAHORS's expressiveness is notably enhanced by pairing and projection, aligning it with rTSA as tree generators. Adapting to them as string generators, MAHORS and rTSA generate exactly the multiple context-free languages as introduced. The study also details a mutual conversion methodology between rTSA and MAHORS, which serves as the cornerstone for the rPTSA and PAHORS conversion to be introduced below.

Remarkably, order-1 MAHORS and HORS are equivalent [142], a conclusion easily drawn by noting the inherent affine nature of first-order HORS – once control is transferred to a parameter, it does not revert, ensuring parameter exclusivity [84, 85, 142]. This conclusion easily extends to the probabilistic context, establishing the equivalence between the order-1 PAHORS and PHORS.

To realise the potential for excellence in any endeavor, one must first ensure the thorough preparation and refinement of his tools.

— Confucius, The Analects

3 Preliminaries

Contents

We now turn to the chapter introducing the basic concepts underlying this work. At the outset, we present some notational conventions and basic concepts to be used throughout the work.

Subsequently, we introduce the main object of study for which the properties and decision procedures are developed: the *Restricted Probabilistic Tree-Stack Automaton* (rPTSA). Following this, we explicitly define the properties that will be the focus of this work, including termination and model checking.

3.1 Notational Conventions and Basic Concepts

Symbolic Convention We use symbols such as X, Y, Z, x, y, z to denote local variables, whose meanings may change according to the context. Conversely, f, g, h are used to denote local functions, which may also vary in their meanings depending on the context. To avoid introducing redundant symbols, we may use the wildcard

symbol "$-$" as a placeholder.

We denote the set of natural numbers by $\mathbb{N}$; the set of real numbers by $\mathbb{R}$, and the set of non-negative real numbers by $\mathbb{R}_{\geq 0}$. For a set X, we denote its power set by 2^X.

List (Sequence) In accordance with functional programming conventions, we assume the data structure of a list (sequence) as a *linked list*, either finite or infinite, using the standard constructor :: to append a new element to an existing list. To concatenate two lists, we use the symbol $+\!\!+$. Formally, when expressing a list as $[x_1, x_2, \dots]$, the following equivalences hold:

$$x_1 :: [x_2, x_3, \dots] \equiv [x_1, x_2, x_3, \dots],$$
$$[x_1, \dots, x_i] +\!\!+ [x_{i+1}, \dots] \equiv [x_1, \dots, x_i, x_{i+1}, \dots].$$

An empty list is denoted by [] or ε. The length of a list s is denoted by $|s|$. Occasionally, we may omit the square brackets and commas and express a list simply as a consecutive sequence of elements, such as $x_1 x_2 \dots$ or $x_1 \dots x_n$.

We also use standard functions to extract information from lists. The function $\mathsf{hd}(-)$ extracts the head element of the list, while $\mathsf{last}(-)$ retrieves the last element for finite lists. The function $\mathsf{concat}(-)$ concatenates a sequence of lists into a single list, where, naturally, except for the last list in the sequence, the preceding lists must be of finite length. The ith element (counting from 1) of a list s is accessed by writing $s[i]$, following programming conventions. Furthermore, the sub-sequence from the ith element to the jth element, inclusive, is denoted by $s[i..j]$. The notation $s[i..]$ represents the sub-sequence starting from the ith element, while $s[..j]$ denotes the prefix up to the jth element, both inclusive. We write $s_1 \sqsubseteq s_2$ to denote that s_1 is a prefix of s_2.

Given a set X, the set of finite length sequences of elements from X is denoted by X^*; the set of finite sequences excluding the empty sequence is denoted by X^+. The set of infinite sequences of elements from X is denoted by X^ω. We may also apply set operations directly to sequences, such as the $\in$ operator and the comprehension notation (e.g., $[x \in X \mid x > 10]$).

Function For a function f from set X to set Y, denoted by $f : X \to Y$, we denote its domain by the operator $\mathrm{dom}(-)$, such that $\mathrm{dom}(f) = X$. A function may be written as a set of mappings in the form $\{x_1 \mapsto y_1, \dots, x_n \mapsto y_n\}$, and updating the mapping from an element $x \in X$ to a (potentially) new value $y \in Y$ is denoted by $f[x \mapsto y]$. Additionally, we denote a partial function f from set X to set Y by $f : X \rightharpoonup Y$, where $\mathrm{dom}(f) \subseteq X$. An update to a partial function f may also expand its domain, thereby adding new elements to $\mathrm{dom}(f)$.

For two functions $f : X \to Y$ and $g : Y \to Z$, the composition of functions, denoted $g \circ f$, is also standard; it is a function of type $X \to Z$ such that for all $x \in X$, $(g \circ f)(x) := g(f(x))$.

We may use the standard lambda notation to write anonymous functions, i.e., $\lambda x_1\ x_2\ \dots\ x_n.t$, where t could be any expression. For example, we may express a constant empty function to be something like "$\lambda - .\emptyset$".

The *inverse function* of f, denoted by f^{-1}, is a function $Y \to 2^X$ that

intuitively captures all elements mapping to a specific element in Y via f. Formally, $f^{-1}(y) := \left\{x \in X \,\middle|\, f(x) = y\right\}$.

Alphabet An alphabet is simply defined as a finite set. More generally, an *($\mathcal{I}$-)indexed* alphabet to the set of indices $\mathcal{I}$ is a function that maps each element of the alphabet to a *subset* of $\mathcal{I}$, where we may refer to the set of the underlying indices $\mathcal{I}$ of the indexed alphabet Σ by $\mathsf{Idx}(\Sigma)$. We may use the symbol Σ to denote a usual or an indexed alphabet without explicit distinction, if it is clear from the context. For an indexed alphabet Σ, we also use the notation $x \in \Sigma$ to signify $x \in \mathrm{dom}(\Sigma)$.

Indexed Vector Given a *finite* set $\mathcal{I}$, an $\mathcal{I}$-indexed vector v of values in V is a function $\mathcal{I} \to V$ with the following special notations. First, we write $v[\imath]$ for $\imath \in \mathcal{I}$ to obtain $v(\imath)$, and each $\imath$ is called a *dimension* of v. Equality of two $\mathcal{I}$-indexed vectors x and y is defined pointwise – we write $x = y$ iff $\forall \imath \in \mathcal{I}.x[\imath] = y[\imath]$.

Extending this, for an $\mathcal{I}$-indexed vector v and a function f applicable to elements of V, we denote pointwise application of f to v by $f(v)$. Similarly, for two vectors x and y of V, indexed by the same finite set, with a binary infix predicate $\sim$ applicable to elements of V, we write $x \sim y$ to denote a pointwise application of $\sim$. For instance, we may simply write $x > y$ for a pointwise comparison. We may omit the indexing set $\mathcal{I}$ if it is clear from the context.

As for symbolic representation, we may write $\vec{x}$ to emphasise the sequential nature of the symbol for both vectors and lists (sequences), but we do not insist on this form.

Trees Throughout this work, when we refer to a tree, we mean a partial function that maps the path of a node to the label of this node. Formally, given an index set $\mathcal{I}$ and an alphabet Σ, a $(\mathcal{I}, \Sigma)$-tree t is a partial function $\mathcal{I}^* \rightharpoonup \Sigma$ with a prefix-closed domain. The property of being "prefix-closed" implies that if a sequence $[x_1, \ldots, x_n] \in \mathrm{dom}(t)$, then any prefix of this sequence, $[x_1, \ldots, x_i]$ where $i \leq n$, must also be in $\mathrm{dom}(t)$. This definition aligns with the intuitive structure of trees, where branches are indexed by $\mathcal{I}$ and nodes are labelled with elements from Σ.

Under this setting, a sequence within the domain of a tree then intuitively uniquely determines a node. Thus we may sometimes refer to it as a *path* or *route* to avoid confusion with the running paths below.

Intuitively, the set $\mathcal{I}$ denotes the possible indices of the branches and Σ is the labels. The tree function stores the information of each path to a node and the corresponding label at that node. For example, the binary trees labelled by Boolean values could be expressed as with indices set $\mathcal{I} = \{\ell, r\}$, and $\Sigma = \{\mathsf{True}, \mathsf{False}\}$. Then, a binary tree t_b whose the root is labelled by True with its two branches each labelling with a single node labelled by True and False respectively could is the partial function:

$$\{\varepsilon \mapsto \mathsf{True}, [\ell] \mapsto \mathsf{True}, [r] \mapsto \mathsf{False}\}$$

Further, given a path $s = [x_1, \ldots, x_n] \in \mathrm{dom}(t)$, the *sub-tree* of t from s is a new $(\mathcal{I}, \Sigma)$-tree t' defined such that: (1) For all sequences $s' \in \mathcal{I}^*$, $s' \in \mathrm{dom}(t')$ iff $s +\!\!+ s' \in \mathrm{dom}(t)$; and (2) If $s' \in \mathrm{dom}(t')$, then $t'(s') = t(s +\!\!+ s')$. In simpler terms, we define $t'(x) := t(s +\!\!+ x)$ for all possible x, with t' implicitly sharing the domain of t shifted by s. For example, the sub-tree of t_b at route $[\ell]$ is simply $\{\varepsilon \mapsto \mathsf{True}\}$.

By default, we consider only *finite* trees, meaning that the domains of the trees are finite.

σ-Algebra A σ-algebra is a collection of sets that is closed under countable unions, countable intersections, and complementation. Formally, given a set X, a σ-algebra $\Theta_X \subseteq 2^X$ on this set is a set of subsets of X satisfying:

- The empty set $\emptyset \in \Theta_X$;
- For any set $Y \in \Theta_X$, its complement $X \setminus Y \in \Theta_X$;
- For any countable set of subsets $\{Y_n\}_{n=1}^{\infty} \subseteq \Theta_X$, it holds that $\bigcup_{n=1}^{\infty} Y_n \in \Theta_X$ and $\bigcap_{n=1}^{\infty} Y_n \in \Theta_X$.

Given a set X, and a set S containing subsets of X, the σ-algebra *generated* by S is defined as the smallest σ-algebra containing S, i.e., the least superset of S that satisfies the conditions required for a σ-algebra.

Probability Measure A *probability measure* is a function that assigns a real number to each event (a subset of the sample space) in a σ-algebra, satisfying specific properties. Given a set X, called the sample space, and a σ-algebra Θ_X, a probability measure P is a function $P : \Theta_X \to [0, 1]$ that satisfies:

- $P(\emptyset) = 0$ and $P(X) = 1$;
- For any countable collection of disjoint sets $\{A_n\}_{n=1}^{\infty} \subseteq \Theta_X$, $P\left(\bigcup_{n=1}^{\infty} A_n\right) = \sum_{n=1}^{\infty} P(A_n)$.

Probability Space A probability space is a triple (X, Θ_X, P) where Θ_X is a σ-algebra on X, and P is a probability measure defined on Θ_X.

We may use the symbol $\mathbb{P}[-]$ to denote $P(-)$ when the underlying probability space is implicitly determined, while the symbol P is reserved for specific references to the probability measure, especially in definitions.

Distribution Given a countable set X, a probability distribution over X is a function $f : X \to \mathbb{R}_{[0,1]}$ such that:

$$\sum_{x \in X} f(x) = 1.$$

A sub-distribution is a distribution where the sum of values over all elements of its domain can be ≤ 1 rather than strictly 1, while remaining ≥ 0.

Given a (sub-)distribution f over set X, the set of elements with non-zero values under the distribution is called its *support*, denoted by $\mathrm{Supp}(f)$, formally defined as:

$$\mathrm{Supp}(f) := \left\{ x \in X \,\middle|\, f(x) > 0 \right\}.$$

The set of all distributions over X is denoted as $\mathsf{Dist}(X)$.

Markov Chain A *Markov chain* (MC) $\mathscr{M}$ is a pair $(\mathbf{St}, \mathcal{P})$ where $\mathbf{St}$ is a (possibly infinite) countable set of states, and $\mathcal{P} : \mathbf{St} \to \mathsf{Dist}(\mathbf{St})$ is the transition function that assigns to each state a distribution over the next states.

Trajectory A *trajectory* $\mathscr{u}$ of a Markov chain $\mathscr{M} = (\mathbf{St}, \mathcal{P})$ is a (possibly infinite) non-empty sequence of states in $\mathbf{St}$ such that for every contiguous states $\mathscr{s}, \mathscr{s}'$ inside, $\mathcal{P}(\mathscr{s})(\mathscr{s}') > 0$. The weight of a finite-length trajectory $wt(\mathscr{u})$ is the product of transitions:

$$wt(\mathscr{s}_1 \dots \mathscr{s}_n) := \prod_{i=1}^{n-1} \mathcal{P}(\mathscr{s}_i)(\mathscr{s}_{i+1}).$$

Naturally, we let the trajectory with a single element to have weight 1.

The set of finite-length trajectories starting from a state $\mathscr{s} \in \mathbf{St}$ is denoted $\mathsf{FinTraj}(\mathscr{s})$. The set of infinite-length trajectories is denoted $\mathsf{InfTraj}(\mathscr{s})$, which extends to finite trajectories as follows: for a finite trajectory $\mathscr{u}$, we use $\mathsf{InfTraj}(\mathscr{u})$ to denote all infinite trajectories with prefix $\mathscr{u}$.

For each state $\mathscr{s} \in \mathbf{St}$, there exists a standard probability space associated with it, denoted by $(X^{(\mathscr{s})}, \Theta_{X^{(\mathscr{s})}}, P^{(\mathscr{s})})$, where:

- $X^{(\mathscr{s})}$ is exactly the set $\mathsf{InfTraj}(\mathscr{s})$;
- $\Theta_{X^{(\mathscr{s})}}$ denotes the σ-algebra generated by $\left\{\mathsf{InfTraj}(\mathscr{u}) \,\middle|\, \mathscr{u} \in \mathsf{FinTraj}(\mathscr{s})\right\}$, where each set $\mathsf{InfTraj}(\mathscr{u})$ corresponding to a finite trajectory $\mathscr{u}$ is referred to as a *cylinder set*.
- The probability measure $P^{(\mathscr{s})}$ must satisfy that $P^{(\mathscr{s})}(\mathsf{InfTraj}(\mathscr{u})) = wt(\mathscr{u})$ for any finite trajectory $\mathscr{u} \in \mathsf{FinTraj}(\mathscr{s})$.

Notably, the preceding definition ensures the existence of a valid probability space, where, especially, $P^{(\mathscr{s})}$ is unique and well-defined, as guaranteed by the *Carathéodory's Extension Theorem* [186]. For completeness, we present an adapted version of the theorem:

Theorem 3.1 (Carathéodory's Extension Theorem [186])**.** *Let X be a set, and let Ξ be an* algebra *of subsets of X; that is, Ξ contains X, is closed under complementation, and under* finite *unions and intersections.*

Suppose $f : \Xi \to [0, 1]$ is a function with $f(X) = 1$, such that f is σ-additive *on Ξ in the following sense: For any countable collection of disjoint sets $\{X_i\}_{i=1}^{\infty}$ in Ξ such that $\bigcup_{i=1}^{\infty} X_i \in \Xi$, we have: $f\left(\bigcup_{i=1}^{\infty} X_i\right) = \sum_{i=1}^{\infty} f(X_i)$.*

Then there exists a unique probability measure g defined on the σ-algebra that is generated by Ξ such that $g(\xi) = f(\xi)$ for all $\xi \in \Xi$.

In our setting, the set $X^{(\mathscr{s})} = \mathsf{InfTraj}(\mathscr{s})$ naturally corresponds to X. We define Ξ as the collection of *all finite unions of the cylinder sets*, including the empty set:

$$\Xi = \left\{ \bigcup_{i=1}^{n} \mathsf{InfTraj}(\mathscr{u}_i) \,\middle|\, n \geq 0,\ \mathscr{u}_i \in \mathsf{FinTraj}(\mathscr{s}) \right\}.$$

This Ξ is an algebra because it contains $X^{(\mathscr{s})}$ and is closed under complementation and finite unions and intersections. The set $X^{(\mathscr{s})}$ itself is in Ξ, since it can be represented by the cylinder set $\mathsf{InfTraj}(\mathscr{s})$ corresponding to the trivial trajectory

$[\mathscr{st}]$. The empty set is also included when $n = 0$. Note especially that the σ-algebra generated by the cylinder sets, $\Theta_{X^{(\mathscr{st})}}$, is the same as that generated by Ξ, as the finite unions are natural members of $\Theta_{X^{(\mathscr{st})}}$.

We further define the function $f : \Xi \to [0, 1]$ by setting $f(\emptyset) = 0$ and, for any non-empty $A \in \Xi$ represented as a finite union of disjoint cylinder sets $A = \bigcup_{i=1}^{n} \mathsf{InfTraj}(\mathscr{u}_i)$, we define $f(A) = \sum_{i=1}^{n} wt(\mathscr{u}_i)$, which can be straightforwardly verified to be σ-additive.

By the theorem, f extends uniquely to a probability measure g on the σ-algebra generated by Ξ, which is precisely $\Theta_{X^{(\mathscr{st})}}$, as discussed. The extended measure g coincides with $P^{(\mathscr{st})}$ on the domain of f, since any measure satisfying the conditions of $P^{(\mathscr{st})}$ must, by definition of probability measures, agree with g on this domain. Therefore, $P^{(\mathscr{st})}$ is uniquely defined on $\Theta_{X^{(\mathscr{st})}}$, and the probability space $(X^{(\mathscr{st})}, \Theta_{X^{(\mathscr{st})}}, P^{(\mathscr{st})})$ is well-defined.

3.2 Restricted Probabilistic Tree-Stack Automaton

We hereby introduce the main object of this work, the *Restricted Probabilistic Tree-Stack Automaton* (rPTSA). Initially, we present the concept of PTSA, along with its associated notations, and subsequently, we introduce the notion of *restriction* based on these concepts.

3.2.1 PTSA

Intuitively, the PTSA extends the concept of *Probabilistic Pushdown Automata* (pPDA) by transforming the linear stack structure of pPDA into a tree-shaped one. This automaton can thus be envisioned as a finite-state machine walking along trees composed of stack symbols, rather than operating on a linear stack. Consequently, the automaton's operations shift from *push* and *pop* to *up* and *down*, enabling the traversal of the tree and the revisitation of locally stored information at each node.

Definition 3.2. A ***Probabilistic Tree-Stack Automaton*** (PTSA) $\mathcal{A}$ is a tuple $(\mathcal{Q}, \mathcal{M}, \Gamma, \Delta, q_{init}, m_{init})$, where:

- $\mathcal{Q}$ is a *finite* set whose elements are called ***control states***.
- $\mathcal{M}$ is a *finite* set whose elements are called ***local memories***.
- Γ is a *finite* set whose elements are called ***stack symbols***.
- $\Delta : \mathcal{Q} \times \mathcal{M} \times (\Gamma \uplus \{\bot\}) \to \mathsf{Dist}(\mathbf{Op})$ is the transitions function, where $\bot \notin \Gamma$ is a distinct symbol representing the *bottom* (or, sometimes referred to as *root*) of the tree-stack, and the set of operations $\mathbf{Op}$ is given by:

$$\mathbf{Op} := \mathcal{Q} \times \mathcal{M} \times (\{up_\gamma \mid \gamma \in \Gamma\} \uplus \{down\})$$

- $q_{init} \in \mathcal{Q}$ is called the ***initial control state***.

- $m_{init} \in \mathcal{M}$ is called the ***initial (/default) local memory***.

The definition may appear somewhat dense, yet the underlying intuition is straightforward: the automaton traverses a tree, known as the *tree-stack*, where the indices are labelled by Γ and the local information at each node can be any element of $\mathcal{M}$. The transitions occur by accessing the locally stored information and determining the subsequent traversal direction either up_γ, which involves moving to the child node at the branch labelled by γ to read its local information, or *down*, which involves returning to the parent node. Let us first introduce the necessary formalisms to fully formalise this intuition, followed by Examples 3.7 and 3.8, which will help to convey the concept more intuitively.

Symbolic Convention In the following, to facilitate exposition, we fix a specific PTSA $\mathcal{A} = (\mathcal{Q}, \mathcal{M}, \Gamma, \Delta, q_{init}, m_{init})$.

We typically use the symbol ρ to denote sequences of stack symbols, where $\rho \in \Gamma^*$. For a non-empty sequence ρ, we often make its last element explicit by writing $\rho = \rho'\gamma$, denoting $\rho = \rho' +\!\!+ [\gamma]$. We range over symbols in $\Gamma \uplus \{\bot\}$ with g. When $\Delta(q, m, g)(op) = p$ with $p > 0$, we say there is a rule $\delta = (q, m, g) \xrightarrow{p} op$ in Δ. In particular, if $p = 1$, we may omit it and write $(q, m, g) \to op$.

We denote $p^{(\delta)}$ as the probability (or weight) of the rule δ and $op^{(\delta)}$ as the operation of the rule δ. Similarly, we define $q^{(\delta)}$, $m^{(\delta)}$, and $g^{(\delta)}$ to extract information from the rule δ, and call them the *source state*, *source memory* and *source stack status*, respectively. Additionally, we define the set $\mathcal{K} := \Gamma \uplus \{\Downarrow\}$ as the *direction*s, ranged over by κ. We use $\kappa(\delta)$ to extract the operation direction of the rule δ: if $op = (-, -, down)$, then $\kappa(\delta) = down$; if $op = (-, -, up_\gamma)$, then $\kappa(\delta) = \gamma$. We intuitively call an operation of the form $(-, -, down)$ a *down* operation, while one of the form $(-, -, up_\gamma)$ an *up* operation (to the direction γ).

Configuration To define the running behaviour of the PTSA, we first introduce the concept of *configuration*s. A configuration c is either the termination symbol $\top$, called the *terminating configuration*, or a triple (q, t, ρ), called the *non-terminating configuration*s, where:

- $q \in \mathcal{Q}$ is a control state of $\mathcal{A}$, referred to as the current control state;
- t is a $(\Gamma, \mathcal{M})$-tree, known as the *tree-stack*;
- $\rho \in \Gamma^*$ is a sequence of stack symbols with the constraint that $\rho \in \mathrm{dom}(t)$, termed the *stack pointer*.

Below, unless explicitly stated otherwise, the term "configuration" refers to a "non-terminating configuration".

Example 3.3. A typical configuration is depicted in Figure 3.1. Particularly, the figure provides an intuitive graphical representation of the configuration $(q_2, t, [\gamma_1])$,

where the tree-stack t is defined as follows:

$$t := \left\{ \begin{array}{lcl} \varepsilon & \mapsto & m_2, \\ [\gamma_1] & \mapsto & m_1, \\ [\gamma_1, \gamma_3] & \mapsto & m_2, \\ [\gamma_1, \gamma_2] & \mapsto & m_3, \\ [\gamma_1, \gamma_2, \gamma_1] & \mapsto & m_1 \end{array} \right\}$$

As illustrated, a configuration can be understood as a tree, with branches labelled by Γ^*, nodes labelled by local memories from $\mathcal{M}$, and an arrow indicating the position of the stack pointer (e.g., $[\gamma_1]$), alongside the current control state (e.g., q_2).

The *initial configuration* of $\mathcal{A}$, denoted by $c^{(\mathcal{A})}_{init}$, is $(q_{init}, \{\varepsilon \mapsto m_{init}\}, \varepsilon)$. The set of *non-terminating* configurations of $\mathcal{A}$ is denoted $\mathcal{C}^{(\mathcal{A})}$. The set of all configurations, including the special terminating configuration $\top$, is denoted $\mathcal{C}^{(\mathcal{A})^\top}$. We may omit the superscript denoting the *host* PTSA $\mathcal{A}$ upon which this configuration is based, if it is clear from the context.

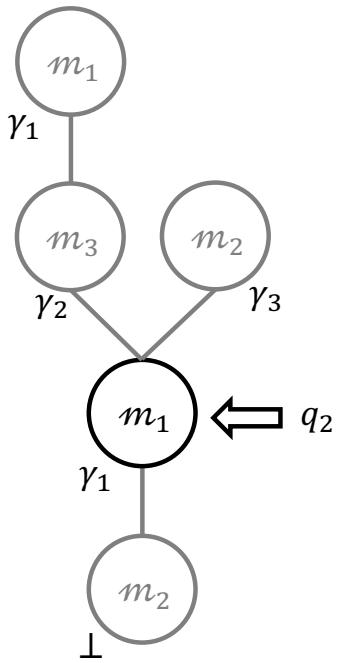

Figure 3.1: Illustrative Configuration of Example 3.3

The concept of configuration should now make it easier to anticipate the running behaviour of an rPTSA intuitively. A typical run of such automata begins with the initial configuration, where the stack pointer positions at the stack bottom. To proceed with a step in the run, one must first match the "local information" to find a rule in Δ to execute, referred to as *status* below, which includes the current control state q, the local memory m at the node pointed to by the stack pointer, and the specific "branch type" g of the current node. This branch type can either be a stack symbol $\gamma \in \Gamma$, indicating which branch the current node belongs to relative to its parent, or simply the stack bottom, denoted by $\perp$. This "branch type" information is termed *stack status* below. For instance, the status of the configuration shown in Figure 3.1 is (q_2, m_1, γ_1). With this triple information, one can determine the distribution of the operations to perform using $\Delta(q, m, g)$. An operation then, moves the stack pointer either upwards or downwards, while also changing the current control state and the local memory at the node originally pointed to by the stack pointer.

Next, let us delve into fully formalising the above sense.

Stack Status and Status Given a sequence of stack symbols $\rho \in \Gamma^*$, we denote $g(\rho) \in \Gamma \uplus \{\perp\}$ as the *stack status* of ρ. This is defined such that if $\rho = \varepsilon$, then it is $\perp$; otherwise, if $\rho = \rho'\gamma$, then it is γ. For convenience, we also refer to the elements within $\Gamma \uplus \{\perp\}$, ranged over by g, as stack status.

Given a (non-terminating) configuration $c = (q_c, t_c, \rho_c)$, the *current status* (or simply *status*) of c, denoted $\mathsf{status}(c)$, is a triple (q, m, g), where:

- $q = q_c$, representing the *current control state* in c;
- $m = t_c(\rho_c)$, referred to as the *current local memory*; and,
- $g = g(\rho_c)$, denoting the *current stack status*.

We may write $q^{(c)}$, $t^{(c)}$, and $\rho^{(c)}$ to extract the respective information from a given configuration c.

Transition A status determines the subsequent transition of a given configuration. Given a non-terminating configuration c, let its status $\mathsf{status}(c)$ be (q, m, g), then any rule with its LHS being exactly the triple (q, m, g) is said to be *enabled* by the configuration. For a source non-terminating configuration $c = (q, t, \rho)$ and an enabled rule $(q, m, g) \xrightarrow{p} op$, the *next configuration* c is uniquely determined by the operation op as described below:

Up If the operation describes an *up* transition $op = (q', m', up_{\gamma'})$ for some q', m', and γ', then the next configuration is $c' = (q', t', \rho\gamma')$, where t' depends on whether the new stack pointer $\rho\gamma'$ specifies an existing node in the domain of the tree-stack t:

- If $\rho\gamma' \in \mathrm{dom}(t)$, then $t' = t[\rho \mapsto m']$, indicating that it reads information from an existing node.
- If $\rho\gamma' \notin \mathrm{dom}(t)$, then $t' = t[\rho \mapsto m', \rho\gamma' \mapsto m_{init}]$, signifying that it creates a new node with the default local memory.

Down If the operation describes a *down* transition $op = (q', m', down)$ for some q', m', the next configuration depends on the structure of the stack pointer and may be either terminating or non-terminating:

- If $\rho = \varepsilon$, the next configuration is $c' = \top$.
- If $\rho = \rho'\gamma$, the next configuration is $c' = (q', t[\rho \mapsto m'], \rho')$.

Given a source configuration c, an enabled transition rule δ, and the next configuration c', a transition between c and c' via the rule δ is denoted $c \xrightarrow{\delta} c'$. The weight of such a transition is $p^{(\delta)}$.

The relation $c \xrightarrow{\delta} c'$ is deterministic, meaning that the next configuration and the rules can be uniquely determined if one of them is fixed. Formally, we have the following:

Lemma 3.4. *Given two configurations c and c', if there exists a transition $c \xrightarrow{\delta} c'$, then the rule δ is uniquely determined.*

Proof. The operation op can be uniquely determined from the form of c and c'. Let the transition function of $\mathcal{A}$ be Δ, then the probability $p = \Delta(\mathsf{status}(c))(op)$ is also unique. □

Remark 3.5 (On Horizontal Operation)**.** In the literature concerning r(P)TSA [99, 102, 187], an additional operation, referred to as the *horizontal* operation, is often discussed. This operation is formally denoted by a state $q' \in \mathcal{Q}$. The transition rule involving this operation is expressed as $(q, t, \rho) \xrightarrow{(q, t(\rho), q(\rho)) \xrightarrow{p} q'} (q', t, \rho)$, which merely changes the current control state to a newly designated one.

This operation is both simple and intuitive, and will not affect the model's expressiveness [99, 187]. In fact, we can equivalently represent such an operation by a sequence of *up* and *down* operations, thereby producing an automaton with effectively the same execution behaviour, interspersed with some auxiliary deterministic transitions, as will be elaborated upon in Section 9.3.2.

Nonetheless, the inclusion of such operations adds extra analytical complexity and can occasionally obscure the exposition. Hence, we opt to exclude this type of operation to improve overall clarity.

Denumerable Markov Chain A PTSA $\mathcal{A}$ describes a denumerable Markov chain, denoted $\mathscr{M}(\mathcal{A})$, which is a pair $(\mathbf{St}^{(\mathcal{A})}, \mathcal{P}^{(\mathcal{A})})$, where $\mathbf{St}^{(\mathcal{A})}$ is exactly the set of *all* configurations ${\mathcal{C}^{(\mathcal{A})}}^{\top}$ and the transition is given by $\mathcal{P}^{(\mathcal{A})}(c)(c') := p^{(\delta)}$, whenever there is a transition $c \xrightarrow{\delta} c'$, otherwise $\mathcal{P}^{(\mathcal{A})}(c)(c') = 0$. Additionally, let the termination configuration $\top$ be a self-absorbing state, i.e., $\mathcal{P}^{(\mathcal{A})}(\top)(\top) = 1$. The superscript may also be ignored when it is clear from the context.

Run A *run* μ of the PTSA $\mathcal{A}$ is a *trajectory* of the denumerable Markov chain associated with $\mathcal{A}$, with the specific constraint that the terminating configuration may appear at most once, at the end. This ensures that for every two contiguous configurations in any run, there is a transition between them. A run is termed an *initial run* if it *begins* with the initial configuration $c_{init}^{(\mathcal{A})}$. Note that, inheriting from the definition of trajectories, a run should contain at least one configuration. A *terminating run* of $\mathcal{A}$ is a *finite* run that ends with the (sole) occurrence of the terminating configuration $\top$; otherwise, it is termed a *non-terminating run*.

The set of runs of $\mathcal{A}$ is denoted by $\mathbf{Run}^{(\mathcal{A})}$, with the set of *initial and terminating runs* being $\mathbf{Run}_{ter}^{(\mathcal{A})}$. Additionally, the set of *infinite runs* starting from a given configuration c is denoted as $\mathbf{Run}_{inf}^{(\mathcal{A})}(c)$, with the set of all infinite runs being $\mathbf{Run}_{inf}^{(\mathcal{A})}$. Superscripts are omitted when the context is clear.

The *weight* of a *finite* run μ, denoted $wt(\mu)$, is also inherited from that for a trajectory. Additionally, the *transition length* of a *finite* run μ, denoted $\ell(\mu)$, is defined as $|\mu| - 1$. Extending this concept, the *total accumulated reward* (TAR) of a *finite* run with respect to a *reward function* $f : \mathcal{C} \times \mathcal{C}^{\top} \to \mathbb{R}_{\geq 0}$, denoted $\mathsf{rwd}_f(\mu)$, is defined as:

$$\mathsf{rwd}_f(c_1 \dots c_n) := \sum_{i=0}^{n-1} f(c_i, c_{i+1}) \tag{3.6}$$

The transition length is a specific case of TAR with the reward function f defined as $f(-,-) := 1$. Thus, $\ell(\mu) = \mathsf{rwd}_{\lambda(-,-).1}(\mu)$. The reward function f is termed *simple* if it depends only on the rule involved in the transition, that is,

$$
\begin{array}{rrcll}
\delta_0 = & (q_{init}, m_{init}, \bot) & \to & (q_A, m_A, up_{\gamma_{AC}}) & \text{Start generating } A \\
\delta_1 = & (q_A, m_{init}, \gamma_{AC}) & \xrightarrow{1/2} & (q_A, m_c, up_{\gamma_{AC}}) & \text{Produce an } A \\
\delta_2 = & (q_A, m_{init}, \gamma_{AC}) & \xrightarrow{1/2} & (q_{back}, m_{top}, down) & \text{End generating } A \\
\delta_3 = & (q_{back}, m_c, \gamma_{AC}) & \to & (q_{back}, m_c, down) & \text{Back to root} \\
 & & & & \\
\delta_4 = & (q_{back}, m_A, \bot) & \to & (q_B, m_B, up_{\gamma_{BD}}) & \text{Start generating } B \\
\delta_5 = & (q_B, m_{init}, \gamma_{BD}) & \xrightarrow{1/2} & (q_B, m_c, up_{\gamma_{BD}}) & \text{Produce a } B \\
\delta_6 = & (q_B, m_{init}, \gamma_{BD}) & \xrightarrow{1/2} & (q_{back}, m_{top}, down) & \text{End generating } B \\
\delta_7 = & (q_{back}, m_c, \gamma_{BD}) & \to & (q_{back}, m_c, down) & \text{Back to root} \\
 & & & & \\
\delta_8 = & (q_{back}, m_B, \bot) & \to & (q_C, m_C, up_{\gamma_{AC}}) & \text{Start generating } C \\
\delta_9 = & (q_C, m_c, \gamma_{AC}) & \to & (q_C, m_c, up_{\gamma_{AC}}) & \text{Generate a } C \text{ to match } A \\
\delta_{10} = & (q_C, m_{top}, \gamma_{AC}) & \to & (q_{back}, m_{top}, down) & \text{End generating } C \\
 & & & & \\
\delta_{11} = & (q_{back}, m_C, \bot) & \to & (q_D, m_D, up_{\gamma_{BD}}) & \text{Start generating } D \\
\delta_{12} = & (q_D, m_c, \gamma_{BD}) & \to & (q_D, m_c, up_{\gamma_{BD}}) & \text{Generate a } D \text{ to match } B \\
\delta_{13} = & (q_D, m_{top}, \gamma_{BD}) & \to & (q_{back}, m_{top}, down) & \text{End generating } D \\
 & & & & \\
\delta_{14} = & (q_{back}, m_D, \bot) & \to & (q_{back}, m_D, down) & \text{Termination}
\end{array}
$$

Figure 3.2: Rules of Example 3.7

$f(c_1, c_1')$ and $f(c_2, c_2')$ will have the same value, as long as $c_1 \xrightarrow{\delta} c_1'$ and $c_2 \xrightarrow{\delta} c_2'$ for the same δ. In such cases, we may express f as a function assigning values to rules, i.e., we may simply write $f(\delta)$ to denote the value of $f(c, c')$ for some c and c' such that $c \xrightarrow{\delta} c'$. Notably, the value of f for two configurations without a transition between them can be arbitrary, as they will not be used.

Example 3.7. Consider a 2-PTSA depicting the process mimicking the (probabilistic) generation of the language $\{A^n B^m C^n D^m\}$, which demonstrates the concept of *cross-serial dependency* in mildly context-sensitive languages. The rules with corresponding explanations are shown in Figure 3.2.

Note that, for simplicity, we hereafter present only the *essential* rules that are the focus of our discussion. Technically, the transitions of rPTSA are *total* such that every state should have a full distribution with the probability summing up to 1. Yet, mentioning all of them requires additional efforts and might cause distraction. Technically, we assume that the missing transitions all lead to a dummy divergence direction γ_Ω, which simply goes infinitely *up* to itself.

The intuitive explanations of the rules corresponding to the generation of the language are attached in the figure. To view more visually the execution of the defined rules, we represent the (initial) runs of this rPTSA in Figure 3.3. A configuration is intuitively drawn as a tree along with an arrow representing

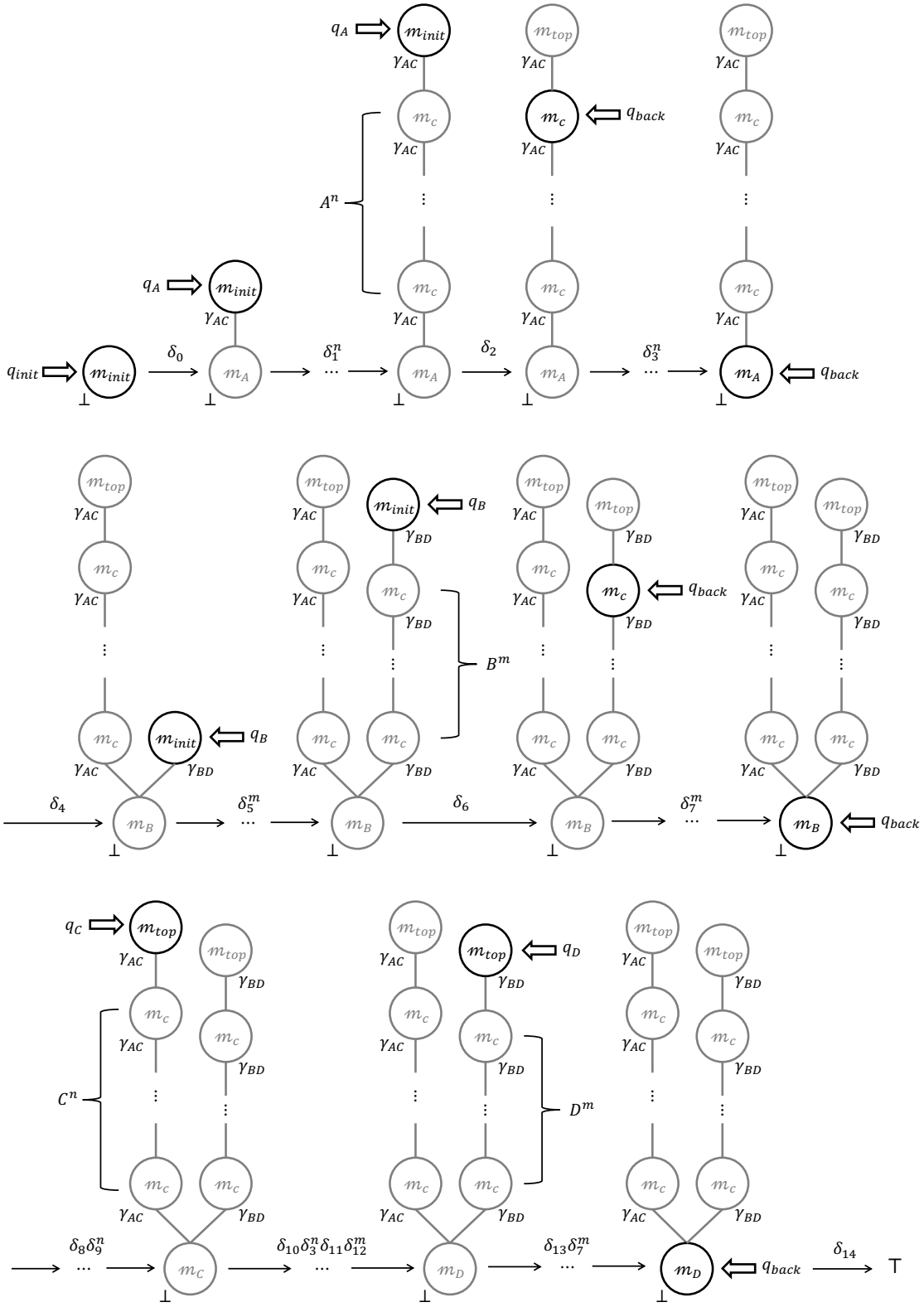

Figure 3.3: Runs of Example 3.7

the position of the stack pointer. Then, the runs, as defined, are sequences of configurations.

More specifically for this example, the (initial) runs begin with the generation of a sequence of A with length n. During this process, the value of n is implicitly stored in the height of the tree-stack on the γ_{AC} branch – more specifically, the number of nodes with $\mathscr{m}_c$ (standing for "count") as local memory on this branch. The sequence then transitions to generating B with length m, where the value of m is similarly stored in the tree-stack on the γ_{BD} branch as the number of nodes with local memory $\mathscr{m}_c$. This is indicated in the runs where configurations transition through states producing A and B elements.

Once the whole B sequence is produced, indicated by reaching the state with "m_{top}", the sequence transitions to generating C, matching the length of the A sequence previously generated. This matching is ensured because the process of generating C is controlled by reading the local memories stored on the tree-stack for each node with $\mathscr{m}_c$. Once the end of A is matched, indicated by $\mathscr{m}_{top}$, the generation of C will also end. Similarly, the generation of the sequence of D matches the length of the B sequence by matching the number of $\mathscr{m}_c$ on the γ_{BD} branch. This demonstrates the capability of rPTSA in expressing the cross-serial dependency, ensuring that the number of A elements matches the number of C elements, and the number of B elements matches the number of D elements.

Example 3.8. As another case, the 2-PTSA depicting the process mimicking the generation of the COPY language $\{ww \mid w \in (A|B)^*\}$ is presented in Figure 3.4. The general idea of the generation is similar to that of the language $\{A^nB^mC^nD^m\}$.

This time, we use a single-branch tree-stack where the local memories, instead of solely storing the information of the length as pure counters, now also keep the information of the entire first w from bottom to top in the local memory of each node. After the first generation of w, the automaton gets all the way *down* back to the root. Following this, in the second pass from the tree-stack bottom, the automaton reads the information of w again and reproduces each character—either A or B—kept within each node from bottom to top.

Path Given a run μ, we extract the uniquely determined rule sequence by $\mathsf{path}(\mu)$, as per Lemma 3.4. We call this sequence the *underlying path* of μ. Then, when we say a *path*, usually denoted by ϕ, we are referring to the underlying path of some run μ. Conversely, we say μ is the *host run* of ϕ or simply that μ hosts ϕ.

Example 3.9. Reconsider the Example 3.7. The paths of its initial runs, as depicted in Figure 3.3, are obtained by extracting the specified rules δ_i in the figure. Specifically, they are seen in Figure 3.5.

A configuration $\mathscr{c}$ is said to *enable* a path ϕ if there is a run μ starting with $\mathscr{c}$ such that $\phi = \mathsf{path}(\mu)$. Conversely, we write $\mathsf{host}(\mathscr{c}, \phi)$ to get μ, which is by definition unique. On the other hand, a path does not uniquely determine a run – it might be enabled by different starting configurations.

$(q_f, m_E, \bot)$	$\to$	(q_f, m_F, up_γ)	Start first w
(q_f, m_E, γ)	$\xrightarrow{p_A}$	(q_f, m_A, up_γ)	Produce an A in the first w
(q_f, m_E, γ)	$\xrightarrow{p_B}$	(q_f, m_B, up_γ)	Produce a B in the first w
(q_f, m_E, γ)	$\xrightarrow{p_T}$	$(q_d, m_T, down)$	Finish generating the first w
(q_d, m_A, γ)	$\to$	$(q_d, m_A, down)$	Back to $\bot$
(q_d, m_B, γ)	$\to$	$(q_d, m_B, down)$	Back to $\bot$
$(q_d, m_F, \bot)$	$\to$	(q_s, m_S, up_γ)	Start the second w
(q_s, m_A, γ)	$\to$	(q_s, m_R, up_γ)	Reproduce A by record
(q_s, m_B, γ)	$\to$	(q_s, m_R, up_γ)	Reproduce B by record
(q_s, m_T, γ)	$\to$	$(q_d, m_R, down)$	Succeed in generation
(q_d, m_R, γ)	$\to$	$(q_d, m_R, down)$	Back to $\bot$ for termination
$(q_d, m_S, \bot)$	$\to$	$(q_d, m_R, down)$	Termination

Figure 3.4: Rules of Example 3.8

$\delta_0\, \delta_1^n\, \delta_2\, \delta_3^n$	Generating A n times and returning *down* to bottom
$+\!\!+\ \delta_4\, \delta_5^m\, \delta_6\, \delta_7^m$	Generating B m times and returning *down* to bottom
$+\!\!+\ \delta_8\, \delta_9^n\, \delta_{10}\, \delta_3^n$	Generating C n times and returning *down* to bottom
$+\!\!+\ \delta_{11}\, \delta_{12}^m\, \delta_{13}\, \delta_7^m$	Generating D m times and returning *down* to bottom
$+\!\!+\ \delta_{14}$	Termination

Figure 3.5: Paths of Initial Runs of Example 3.7

Further, given a finite run μ, let the starting and ending configurations be c_s and c_e, then we may sometimes write $c_s \xrightarrow{\phi} c_e$. And we may naturally say c_e is the *resulting* configuration of *executing* ϕ from c_s.

Furthermore, given a *finite* path ϕ, the *weight* thereof, denoted by $wt(\phi)$, is the product of the transition probabilities of the rules contained therein. Similarly, the *transition length* thereof, denoted by $\ell(\phi)$, is defined as the length of the path, such that $\ell(\phi) := |\phi|$. When extending to the *total accumulated reward* of it, in the absence of information regarding the configurations involved, the definition is restricted to cases where the reward function f is *simple*. Under these circumstances, the value, denoted by $\mathsf{rwd}_f(\phi)$, is calculated as: $\mathsf{rwd}_f(\phi) := \sum_{\delta \in \phi} f(\delta)$.

It is evident that these three definitions, applicable to any path, are fundamentally equivalent to those for any run that hosts it. That is, if $\mathsf{path}(\mu) = \phi$, then $wt(\mu) = wt(\phi)$, $\ell(\mu) = \ell(\phi)$, and $\mathsf{rwd}_f(\mu) = \mathsf{rwd}_f(\phi)$ for any *simple* reward function f.

3.2.2 Restrictions

It is readily apparent that a PTSA exhibits (probabilistic) Turing completeness. By constraining the tree to a single branch ($|\Gamma| = 1$), we derive a (probabilistic) Turing machine equipped with a one-way tape.

Hence, in order to analyse this model, we impose a constraint called k-restriction to the model. Intuitively, such a restriction constrains the number of visit *from below* to any node on the tree-stack.

Restriction for Run Given a(n) (infinite) run $\mu = c_1 \ldots$, we say that it is *k-restricted* iff for any sequence of stack symbols ρ and any stack symbol γ, there are *at most* k pairs of contiguous configurations of the form $(-,-,\rho)(-,-,\rho\gamma)$ in μ. The *smallest restriction number* of μ is the highest count of such pairs observed for any single ρ-γ combination across the entire run. Intuitively, the k-restriction says that for any node at the tree-stack, there are at most k visits from its parent node. Apparently, a k-restricted run is also $(k+1)$- and $(k+n)$-restricted for any $n \geq 0$. The number k is then called a *restriction number*.

k-Restricted PTSA A PTSA $\mathcal{A}$ is termed k-restricted if all of its initial runs are k-restricted. This concept, known as the *local* k-restriction, is sufficient for most discussions. As a broader concept, the *global* k-restriction requires that all runs, not just the initial ones, be k-restricted. This distinction is particularly important in global model checking, where every configuration, including those not reachable from the initial configuration, must be considered.

For most purposes, including termination analysis and local model checking algorithms, considering only the local restrictions is adequate. Thus, in the subsequent discussion, unless specified otherwise, k-restriction will refer to the local restriction. Furthermore, in Chapter 7, a validation algorithm addressing both local and global restrictions is presented.

A k-restricted PTSA can also be denoted as k-PTSA. Likewise, k is called a *restriction number* of $\mathcal{A}$.

Definition 3.10 (Restricted PTSA)**.** A PTSA $\mathcal{A}$ is a *restricted Probabilistic Tree-Stack Automaton* (rPTSA), iff it is k-restricted for some natural number k.

3.3 Termination Properties

The termination properties are fundamental properties to inquire when given a probabilistic system. We here introduce these properties for rPTSA.

Termination Probability For PTSA $\mathcal{A}$, the *termination probability* $tp(\mathcal{A})$ is given by:

$$tp(\mathcal{A}) := \sum_{\mu \in \mathbf{Run}_{ter}} wt(\mu) \tag{3.11}$$

Expected Termination Time For PTSA $\mathcal{A}$, the *expected termination time* $ett(\mathcal{A})$ is the expected times of *transitions*, which is given by:

$$ett(\mathcal{A}) := \sum_{\mu \in \mathbf{Run}_{ter}} wt(\mu) \cdot \ell(\mu) \tag{3.12}$$

The *conditional* expected termination time, which intuitively represents the expected length of transitions conditioning on the terminating runs, is obtained by dividing the expected termination time by the termination probability. Formally, when $tp(\mathcal{A}) > 0$:

$$cett(\mathcal{A}) := \frac{ett(\mathcal{A})}{tp(\mathcal{A})} \tag{3.13}$$

Expected Total Accumulated Reward Then, for PTSA $\mathcal{A}$ and a given reward function f, the *expected total accumulated reward* of $\mathcal{A}$ is given by:

$$etar_f(\mathcal{A}) := \sum_{\mu \in \mathbf{Run}_{ter}} wt(\mu) \cdot \mathsf{rwd}_f(\mu) \tag{3.14}$$

Clearly, the expected termination time is simply an expected total accumulated reward with the reward function set to return a constant 1 – that:

$$ett(\mathcal{A}) = etar_{\lambda(-,-).1}(\mathcal{A}) \tag{3.15}$$

(P)AST A given rPTSA $\mathcal{A}$ is said to achieve *almost sure termination* (AST) if $tp(\mathcal{A}) = 1$, and is said to achieve *positive almost sure termination* (PAST) if it is AST and has a finite expected termination time, such that $ett(\mathcal{A}) < \infty$.

Termination Problems To analyse the termination properties of a given rPTSA $\mathcal{A}$ (including both termination probability and expected termination time), we generally consider three main types of termination problems: *qualitative*, *quantitative*, and *approximation* problems.

Qualitative Problems Qualitative problems focus on the nature of the termination probabilities. They include determining whether $\mathcal{A}$ is AST, never terminates ($tp(\mathcal{A}) = 0$), or has a termination probability between 0 and 1 ($0 < tp(\mathcal{A}) < 1$). For expected termination times (or more broadly, expected total accumulated reward), the questions include whether $ett(\mathcal{A}) < \infty$ (or $etar_f(\mathcal{A}) < \infty$ for a specific function f). Additionally, one might inquire whether $\mathcal{A}$ is PAST.

Quantitative Problems Quantitative problems, often referred to as threshold problems, involve comparing the termination probability and expected termination time (or more generally, expected total accumulated reward) against a specific threshold. These problems ask whether $tp(\mathcal{A})$ or $ett(\mathcal{A})$ (or $etar_f(\mathcal{A})$ for a given function f) meets a comparison condition relative to a given real number $c \in \mathbb{R}_{\geq 0}$, i.e., whether $\sim c$ holds, with $\sim$ being a comparison operator among $\leq, <, =, \neq, >, \geq$.

Approximation Problems Approximation problems aim to compute an approximate value for the termination probability $tp(\mathcal{A})$, the expected termination time $ett(\mathcal{A})$, and the expected total accumulated reward $etar_f(\mathcal{A})$ for a given reward function f.

3.4 Model Checking

3.4.1 ω-Regular Property

An ω-regular property is a type of property that describes sets of infinite sequences, which are fundamental in the analysis of systems with long-term behaviours [72]. There are various methods to define an ω-regular property, such as using Büchi automata [72], Streett automata [188], or parity automata [189]. Each type of automaton offers a unique perspective and utility in different contexts of formal verification and model checking.

For simplicity, we adopt the *Deterministic Rabin Automaton* (DRA) in this context, primarily due to its deterministic nature and its ability to fully express any ω-regular property. This choice aims to reduce the complexity of the state space, making both theoretical analysis and demonstration of our model checking algorithm more straightforward.

Definition 3.16 (DRA)**.** A ***Deterministic Rabin Automaton*** (DRA) $\mathcal{D}$ is a tuple $(\mathfrak{D}, \Sigma, \mathfrak{T}, \mathscr{d}_{init}, \mathscr{R})$, where:

- $\mathfrak{D}$ is a *finite* set of DRA states;
- Σ is a *finite* alphabet;
- $\mathfrak{T} : \mathfrak{D} \times \Sigma \to \mathfrak{D}$ is the transition function;
- $\mathscr{d}_{init} \in \mathfrak{D}$ is the initial DRA state;
- $\mathscr{R}$ contains pairs of the form (E, F), where $E, F \subseteq \mathfrak{D}$.

Running Behaviour Given a DRA $\mathcal{D} = (\mathfrak{D}, \Sigma, \mathfrak{T}, \mathscr{d}_{init}, \mathscr{R})$, and an *infinite* input sequence $s = \mathsf{a}_1\mathsf{a}_2 \cdots \in \Sigma^\omega$, an *infinite* sequence $s_\mathscr{d} = \mathscr{d}_0\mathscr{d}_1\mathscr{d}_2 \cdots \in \mathfrak{D}^\omega$ is called the *run* of $\mathcal{D}$ on the input sequence $\mathsf{a}_1\mathsf{a}_2 \ldots$, if: $\mathscr{d}_0 = \mathscr{d}_{init}$, and for all $i \in \{1, 2, \ldots\}$, $\mathscr{d}_i = \mathfrak{T}(\mathscr{d}_{i-1}, \mathsf{a}_i)$. The states that appear infinitely often in $s_\mathscr{d}$ are denoted $\mathsf{Inf}(s_\mathscr{d})$. We say that the input sequence $\mathsf{a}_1\mathsf{a}_2 \ldots$ and its run $\mathscr{d}_0\mathscr{d}_1\mathscr{d}_2 \ldots$ are *accepted* by $\mathcal{D}$, denoted $s \ldots \models \mathcal{D}$, if there exists a pair $(E, F) \in \mathscr{R}$ such that: $E \cap \mathsf{Inf}(s_\mathscr{d}) = \emptyset$ and $F \cap \mathsf{Inf}(s_\mathscr{d}) \neq \emptyset$. Intuitively, this means that *all* states in E must appear only *finitely* often, while there must *exist* a state in F that appears *infinitely* often.

Model Checking Given a Markov chain $\mathscr{M}$ $(\mathbf{St}, \mathcal{P})$, a DRA $\mathcal{D} = (\mathfrak{D}, \Sigma, \mathfrak{T}_\mathcal{D}, \mathscr{d}_{init}, \mathscr{R})$ is said to describe a class of trajectories of $\mathscr{M}$ via an *interpretation map* $f : \mathbf{St} \to \Sigma$ that maps the possibly infinite MC states to the finite alphabet of $\mathcal{D}$. Given an MC

state $\mathscr{s} \in \mathbf{St}$, we denote the probability of the infinite trajectories being described by the $\mathcal{D}$ as $\mathbb{P}[\mathscr{s} \models_f \mathcal{D}]$, which is given by:

$$\mathbb{P}[\mathscr{s} \models_f \mathcal{D}] := \mathbb{P}\left[\left\{\mathscr{u} \in \mathsf{InfTraj}(\mathscr{s}) \,\middle|\, f(\mathscr{u}) \models \mathcal{D}\right\}\right] \tag{3.17}$$

Here, $f(\mathscr{u})$ denotes a pointwise application of f to the sequence.

By employing the concept of denumerable Markov chain of a given rPTSA $\mathcal{A}$, a DRA $\mathcal{D}$ $(\mathfrak{D}, \Sigma, \mathfrak{T}_{\mathcal{D}}, \mathscr{d}_{init}, \mathscr{R})$ can describe a class of runs of $\mathcal{A}$ via an interpretation map f that maps the set of all configurations $\mathcal{C}^{(\mathcal{A})^\top}$, including the terminating configuration $\top$ of $\mathcal{A}$, to the DRA alphabet Σ. Notably, the model checking does not operate solely on the sequences of the control states of $\mathcal{A}$; instead, it generally considers those of the entire configuration or, at least, the sequences of statuses.

3.4.2 Logical Specifications

In order to define the LTL and branching-time model checking, we here first define the formulas. Throughout the paper, we will use Φ to denote the *state* formulas that represent a proposition against (Markov chain) states (e.g. configuration of rPTSA); we use φ to denote the *path* formulas that represent a proposition against (Markov chain) trajectories (e.g. runs of rPTSA).

We here fix a finite set of *atomic propositions* $\mathbb{A}$, which is ranged over by p, to be used throughout this work.

Then, the syntax of a well-formed PCTL* formula Φ is given inductively by:

$$\begin{aligned}\Phi &:= \mathsf{p} \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \mathbb{P}^{\sim c}[\varphi] \\ \varphi &:= \Phi \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathcal{X}\varphi \mid \varphi_1 \mathcal{U} \varphi_2\end{aligned}$$

where p ranges over the atomic propositions, $\sim \in \{\leq, <, \neq, =, >, \geq\}$ and, $c \in [0, 1]$ is a real number.

The LTL formulas are the PCTL* formulas without the probability operator, which hence is:

$$\varphi := \mathsf{p} \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathcal{X}\varphi \mid \varphi_1 \mathcal{U} \varphi_2$$

Moreover, the *qualitative* fragment of PCTL*, abbreviated as qPCTL*, encompasses only the probabilistic queries of form $= 1$ rather than the general threshold comparison [1].

Definition 3.18 (Valuation)**.** Given a Markov chain $\mathscr{M}$ $(\mathbf{St}, \mathcal{P})$, a *valuation* ν based on $\mathscr{M}$ is a function $\mathbb{A} \to 2^{\mathbf{St}}$ that maps the atomic propositions to the subsets of the state set of the Markov chain.

[1] The query of $\mathbb{P}^{=0}[-]$ can be represented by taking $\mathbb{P}^{=1}[\neg -]$.

Semantics Given a Markov chain $\mathscr{M} = (\mathbf{St}, \mathcal{P})$, and a valuation ν that is based on it, the *evaluation* of a state formula results in a set of Markov chain states, while that of a path formula produces a set of *infinite* trajectories. They are represented by two functions: $[\![-]\!]^{\mathsf{s}}_{\nu}$ and $[\![-]\!]^{\mathsf{p}}_{\nu}$, as follows:

The evaluation of states $[\![-]\!]^{\mathsf{s}}_{\nu}$ is given by:

$$\begin{aligned}
[\![\mathsf{p}]\!]^{\mathsf{s}}_{\nu} &:= \nu(\mathsf{p}) \\
[\![\neg\Phi]\!]^{\mathsf{s}}_{\nu} &:= \mathbf{St} \setminus [\![\Phi]\!]^{\mathsf{s}}_{\nu} \\
[\![\Phi_1 \wedge \Phi_2]\!]^{\mathsf{s}}_{\nu} &:= [\![\Phi_1]\!]^{\mathsf{s}}_{\nu} \cap [\![\Phi_2]\!]^{\mathsf{s}}_{\nu} \\
[\![\mathbb{P}^{\sim c}[\varphi]]\!]^{\mathsf{s}}_{\nu} &:= \left\{ \mathscr{s} \in \mathbf{St} \;\middle|\; \mathbb{P}\left[[\![\varphi]\!]^{\mathsf{p}}_{\nu}\right] \sim c \right\}
\end{aligned}$$

And the path formula evaluation is given by:

$$\begin{aligned}
[\![\Phi]\!]^{\mathsf{p}}_{\nu} &:= \left\{ \mathsf{InfTraj}(\mathscr{s}) \;\middle|\; \mathscr{s} \in [\![\Phi]\!]^{\mathsf{s}}_{\nu} \right\} \\
[\![\neg\varphi]\!]^{\mathsf{p}}_{\nu} &:= \mathsf{InfTraj} \setminus [\![\varphi]\!]^{\mathsf{p}}_{\nu} \\
[\![\varphi_1 \wedge \varphi_2]\!]^{\mathsf{p}}_{\nu} &:= [\![\varphi_1]\!]^{\mathsf{p}}_{\nu} \cap [\![\varphi_2]\!]^{\mathsf{p}}_{\nu} \\
[\![\mathcal{X}\varphi]\!]^{\mathsf{p}}_{\nu} &:= \left\{ \mathscr{u} \in \mathsf{InfTraj} \;\middle|\; \mathscr{u}_{[1..]} \in [\![\varphi]\!]^{\mathsf{p}}_{\nu} \right\} \\
[\![\varphi_1 \mathcal{U} \varphi_2]\!]^{\mathsf{p}}_{\nu} &:= \left\{ \mathscr{u} \in \mathsf{InfTraj} \;\middle|\; \exists j. \mathscr{u}_{[j..]} \in [\![\varphi_2]\!]^{\mathsf{p}}_{\nu} \wedge \forall 0 \leq i < j. \mathscr{u}_{[i..]} \in [\![\varphi_1]\!]^{\mathsf{p}}_{\nu} \right\}
\end{aligned}$$

where $\mathscr{u}_{[i..]}$ represents the sub-infinite-list starting from the i-th index of the infinite list $\mathscr{u}$.

We say a state $\mathscr{s}$ *satisfies* a state formula Φ with respect to valuation ν, denoted by $\mathscr{s} \models_{\nu} \Phi$, when $\mathscr{s} \in [\![\Phi]\!]^{\mathsf{s}}_{\nu}$. Likewise, we say an *infinite* trajectory $\mathscr{u}$ *satisfies* a path formula φ, denoted $\mathscr{u} \models_{\nu} \varphi$, to denote $\mathscr{u} \in [\![\varphi]\!]^{\mathsf{p}}_{\nu}$.

When we say we are to model check a state formula Φ as a specification against an rPTSA $\mathcal{A}$ via a valuation ν, we mean that $\mathscr{c}^{\mathcal{A}}_{init} \models_{\nu} \Phi$ with the underlying Markov chain being the denumerable Markov chain of $\mathcal{A}$.

3.4.3 Regular Valuation

In the preceding part, we introduced the concept of *valuation* as a function mapping atomic propositions to the power set of the given Markov chain.

In this study, we specifically consider the denumerable Markov chain derived from rPTSA. Hence, in the following discussion, when we refer to valuations, we implicitly assume them pertaining to the Markov chain $\mathscr{M}(\mathcal{A})$.

It is evident that with the general concept of valuation, the model checking problem of rPTSA against the specifications is *undecidable*, since the valuation itself might not be computable. Therefore, we will restrict our focus to more concrete and finitely representable concepts – namely, *simple valuation* and *regular valuation*. Only by employing these constrained concepts, can we precisely present the model checking problems we focus on.

We first define simple valuation.

Definition 3.19 (Simple Valuation)**.** Intuitively, a simple valuation means the result of the valuation depends solely on the current status of the given configuration.

More precisely, given a valuation ν, we say it is a *simple valuation* if and only if: for every two rPTSA configurations $\mathscr{c}$ and $\mathscr{c}'$, if $\mathsf{status}(\mathscr{c}) = \mathsf{status}(\mathscr{c}')$, then for all p, it holds that $\mathscr{c} \in \nu(\mathsf{p}) \iff \mathscr{c}' \in \nu(\mathsf{p})$. Given this property, we may sometimes write a simple valuation ν as a function from the set of atomic propositions $\mathbb{A}$ to a set of statuses and possibly the termination configuration $\top$.

Next, we explore the more complex idea of regular valuation. This idea stems from the observation that a configuration can be represented as a tree, where the control state is appended to the label of the tree-stack node pointed at by the stack pointer. In this way, we can represent a regular set of configurations using something called a *finite-state tree automaton*, which is a tool known to describe a regular language of trees [190]. This is similar to how a *finite-state automaton* is used to describe a regular language of strings [72]. To formalise the sense, let us first introduce the concept of a *tree automaton*.

Definition 3.20 (Finite-State Tree Automaton)**.** A ***Finite-State Top-Down Tree Automaton***, or simply *Tree Automaton* (TA), $\mathcal{T}$ is a tuple $(T, \Sigma, \mathscr{T}, T_{init})$, where: T is a *finite* set of tree automaton states; Σ is a finite indexed alphabet; $\mathscr{T}$ is the transition relation, that: $\mathscr{T} \subseteq T \times \mathrm{dom}(\Sigma) \times (\mathsf{Idx}(\Sigma) \rightharpoonup T)$ such that for any triple $(t, \mathtt{a}, f) \in \mathscr{T}$ and $\mathtt{a} \in \mathrm{dom}(\Sigma)$, $\mathrm{dom}(f) = \Sigma(\mathtt{a})$; and, $T_{init} \subseteq T$ is a *non-empty* set with the elements called the initial states. We may sometimes write the tuple $(t, \mathtt{a}, f)$ in the form of a transition rule that $t \xrightarrow{\mathtt{a}} f$, especially when displaying.

Analogous to the automaton for strings, where the input is a sequence of letters from an alphabet and the transition relation is of the form $(t, \mathtt{a}, t')$, indicating that when a state t reads a letter $\mathtt{a}$, the next state becomes t', the Tree Automaton (TA) in our case reads inputs that are $\mathcal{I}$-indexed trees, where letters are attached to each node as labels. Thus, the transition rule for the TA is of the form $(t, \mathtt{a}, f)$, where t is the current state, $\mathtt{a}$ is the label of the current node, and f is a partial function mapping indices from $\mathsf{Idx}(\Sigma(\mathtt{a}))$ to states in the automaton. After reading the label $\mathtt{a}$ at a node while in state t, the TA reads the sub-trees rooted at indices from $\mathrm{dom}(f)$, which, by definition, is required to match exactly the set of indices associated with $\mathtt{a}$ in the indexed alphabet Σ. Each sub-tree is then read from the state specified by $f(\imath)$ for each index $\imath$.

More specifically, fix a TA $\mathcal{T}$ $(T, \Sigma, \mathscr{T}, T_{init})$, given a $(\mathsf{Idx}(\Sigma), \mathrm{dom}(\Sigma))$-tree $\mathscr{t}$, we say that it is *accepted* by $\mathcal{T}$ from a state t if and only if: there exists a triple $(t, \mathscr{t}(\varepsilon), f) \in \mathscr{T}$ such that: (1) the set of all branches one level above the root on $\mathscr{t}$ is exactly $\mathrm{dom}(f)$, that $\{\imath \mid [\imath] \in \mathrm{dom}(\mathscr{t})\} = \mathrm{dom}(f)$; and, (2) for every $\imath \in \mathrm{dom}(f)$, the sub-tree of $\mathscr{t}$ from $[\imath]$ is accepted by $\mathcal{T}$ from $f(\imath)$. Additionally, we say t is an *accepting state* of tree $\mathscr{t}$ in $\mathcal{T}$.

Then, we say a tree $\mathscr{t}$ is *accepted* by the TA $\mathcal{T}$, iff it is accepted from a state within T_{init}.

Example 3.21. Consider a TA $(T, \Sigma, \mathscr{T}, T_{init})$, where T contains two states t_1 and t_2; Σ also includes two symbols: s_1, associated with an empty index set $\emptyset$, and s_2, associated with a binary index set $\{\ell, r\}$. The transition relation $\mathscr{T}$ is defined by

the following rules:

$$\begin{array}{lll} & & \ell \quad r \\ t_1 & \xrightarrow{s_2} & t_1, \quad t_2 \\ t_1 & \xrightarrow{s_2} & t_2, \quad t_2 \\ t_2 & \xrightarrow{s_1} & \end{array}$$

For simplicity, the rules are presented in tabular form, showing the corresponding states for indices ℓ and r on the right-hand side of each rule. For example, the first line of the table represents the transition rule $(t_1, s_2, \{\ell \mapsto t_1, r \mapsto t_2\})$, while the last line represents the rule $(t_2, s_1, \emptyset)$. Finally, let $T_{init} = \{t_1\}$ be the set of initial states.

This TA intuitively describes binary trees where leaves are labelled by s_1, and internal nodes are labelled by s_2, with the constraint that each internal node must have two branches, and only the left branch may be non-leaf.

As an example, consider the tree depicted in Figure 3.6 (which is NOT an rPTSA configuration). At the root, the first line of the transition rule applies. For the sub-tree rooted at the path $[\ell]$, the second line applies, with the sub-trees at the leaves governed by the last line.

Notably, the tree is accepted because each node has an applicable corresponding transition rule. Especially, if any of these rules were absent, the tree would not be accepted. For instance, if the first rule were missing, the only applicable rule at the root would be the second line, considering the label s_2, which would impose the state t_2 on the node at $[\ell]$, which is then not a proper accepting state of this sub-tree.

Remark 3.22 (On Tree Automata)**.** The given definition is a *non-deterministic top-down* automaton. We here do not additionally introduce the deterministic version of the automaton as the deterministic top-down automata are not as expressive as this non-deterministic version.

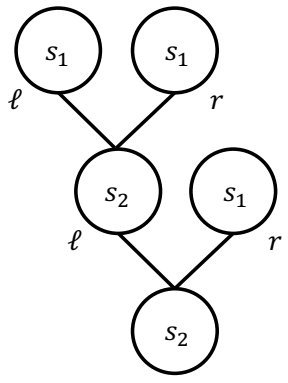

Figure 3.6: Illustrative Tree in Example 3.21

Recall that the concept of tree automaton is used to represent sets of configurations of PTSA, referred to as *regular* sets. To achieve this representation, it is necessary to transform these configurations into a tree format that a TA can process. This involves mapping the configurations to a tree structure compatible with the TA's requirements.

Then, we formally represent a configuration $\mathscr{c}$ as a tree, which we refer to as the *configuration tree*, denoted by $\mathsf{tree}(\mathscr{c})$. The overall concept is straightforward and can be illustrated with an example, as shown in Figure 3.7. Consider a configuration $\mathscr{c}_s$ on the LHS, which has two possible stack symbols, namely, ℓ and r; its corresponding configuration tree, $\mathsf{tree}(\mathscr{c}_s)$, is depicted on the RHS. In the configuration tree, the current control state of the configuration is attached to the node pointed to by the stack pointer,

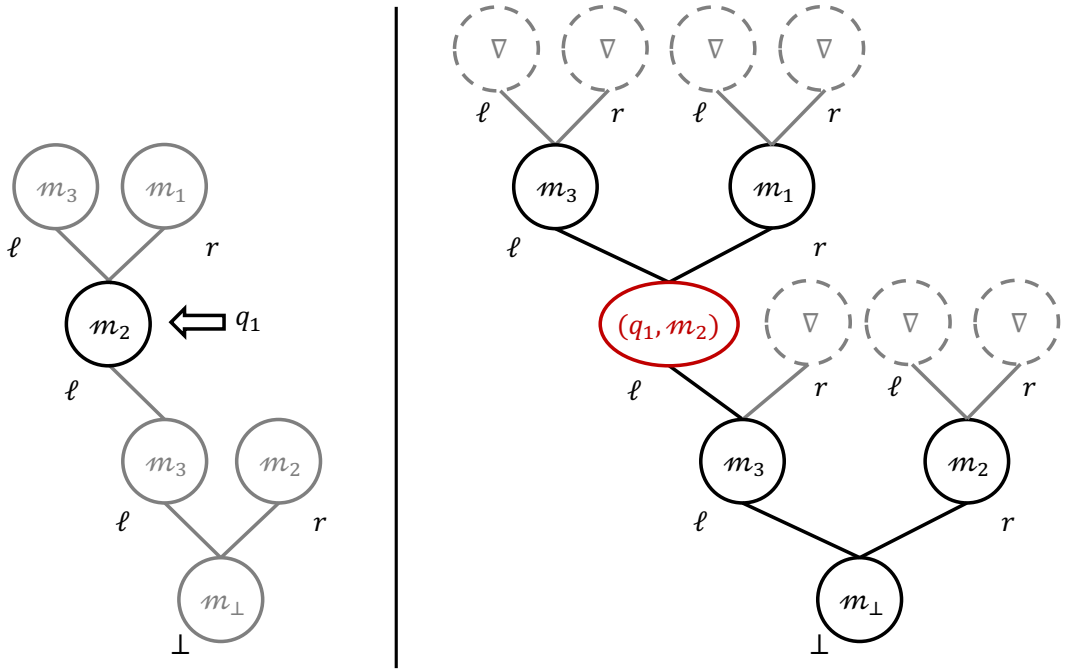

Figure 3.7: Configuration c_s and its Configuration Tree $\mathsf{tree}(c_s)$

displayed as the red oval node. Additionally, we introduce a special mark ∇ to maintain format correctness so that every internal node is guaranteed to have the set of indices being exactly Γ, with the leaves nodes being all labelled by ∇. This ensures that the tree automaton accepting the tree $\mathsf{tree}(c)$ can have an indexed alphabet $\Sigma^{\mathcal{A}}$, which contains three types of elements:

- A pair $(q, m) \in \mathcal{Q} \times \mathcal{M}$, denoting the exact location where the stack pointer points, along with the control state information. For this case, we have that $\Sigma^{\mathcal{A}}((q, m)) = \Gamma$, allowing further branching.
- A usual local memory $m \in \mathcal{M}$, representing a typical internal node not pointed at by the stack pointer, for which we also have $\Sigma^{\mathcal{A}}(m) = \Gamma$.
- The special element ∇, serving as a placeholder for uncreated nodes, where $\Sigma^{\mathcal{A}}(\nabla) = \emptyset$.

Formally, for a given configuration $c = (q, t, \rho)$ of the rPTSA $\mathcal{A}$, its configuration tree $\mathsf{tree}(c)$ is constructed through the following steps:

- Insert the current control state at the location specified by the stack pointer, namely, creating a new tree $t' := t[\rho \mapsto (q, t(\rho))]$.
- Further to t', we insert the placeholder ∇ to all uncreated nodes that are one level above the created ones to technically adhere to the Γ-indexed structure. More specifically, consider all those ρ' and γ such that: $\rho' \in \mathrm{dom}(t)$ but $\rho'\gamma \notin \mathrm{dom}(t)$. We introduce a new mapping $\rho'\gamma \mapsto \nabla$.

We may refer to the configuration tree concept directly without explicitly mentioning the specific configuration it is based on.

A TA $\mathcal{T} = (T, \Sigma, \mathscr{T}, T_{init})$ is said to describe a *regular* set of configurations if and only if $\Sigma = \Sigma^{\mathcal{A}}$. The set of configurations it describes is denoted by $\mathcal{C}(\mathcal{T})$ and is defined as:

$$\mathcal{C}(\mathcal{T}) := \left\{ c \in \mathcal{C} \,\middle|\, \mathsf{tree}(c) \text{ is accepted by } \mathcal{T} \right\}$$

Note that the TA does not necessarily include solely those corresponding to a proper configuration – for example, the elements of $(q, m) \in \mathcal{Q} \times \mathcal{M}$ might not be unique in the accepted trees.

Definition 3.23. A valuation ν is called a *regular* over the rPTSA $\mathcal{A}$ if for each atomic proposition $\mathsf{p} \in \mathbb{A}$, there exists a TA $\mathcal{T}_{\mathsf{p}}$ such that $\nu(\mathsf{p})$ is the set of configurations $\mathcal{C}(\mathcal{T}_{\mathsf{p}})$, possibly including the termination configuration $\top$.

3.4.4 Model Checking Problems

We now introduce the model checking problems. This work focuses specifically on the following issues:

Quantitative ω-Regular Model Checking Consider a DRA $\mathcal{D}$ with alphabet Σ and a *simple* interpretation map f. This map, mimicking the concept of simple valuation, assigns configurations with the same status to the same element in Σ. The model checking problem is quantitative, asking whether the probability $\mathbb{P}\left[c_{init} \models_f \mathcal{D}\right] \sim c$ holds for a given constant $c \in [0, 1]$, and a comparison operator $\sim$, ranging over $\{\leq, <, \neq, =, >, \geq\}$.

Global Qualitative LTL Model Checking The *global* qualitative model checking problem requires computing *all* configurations c of $\mathcal{A}$ such that $c \models_\nu \mathbb{P}^{=1}\left[\varphi\right]$ for a given LTL formula φ and a *regular* valuation ν.

Qualitative Branching-Time Model Checking This model checking concerns the qualitative fragment of the PCTL* properties against rPTSA. It asks whether, given a qPCTL* formula Φ, $c^{\mathcal{A}}_{init} \models_\nu \Phi$ holds for a given *regular* valuation ν.

The problems of interest can be classified into two categories: linear-time model checking for the ω-regular and LTL properties, which will be addressed in Chapter 5, and branching-time model checking for the qualitative PCTL* property, detailed in Chapter 6.

We can only see a short distance ahead, but we can see plenty there that needs to be done.

— Alan Turing

4 Termination Analysis

Contents

Termination analysis plays a crucial role in the study of probabilistic models. Understanding the property is not only essential for the practical use of the model but also lays the groundwork for more advanced analyses. In this chapter, we explore this analysis for rPTSA, concentrating on the termination probability, the expected time to termination, and, in a broader sense, the expected total accumulated reward.

It is noteworthy that the technique used for analysing termination, specifically the method of decomposing rPTSA runs, will also serve as a key technique in subsequent parts.

To facilitate exposition, we fix an rPTSA $\mathcal{A} = (\mathcal{Q}, \mathcal{M}, \Gamma, \Delta, q_{init}, m_{init})$ subject

$$
\begin{array}{lcccl}
\delta_0 & = & (q_f, m_{init}, \bot) & \to & (q_f, m_F, up_\gamma) \\
\delta_1 & = & (q_f, m_{init}, \gamma) & \xrightarrow{p_A} & (q_f, m_A, up_\gamma)(\star) \\
\delta_2 & = & (q_f, m_{init}, \gamma) & \xrightarrow{p_B} & (q_f, m_B, up_\gamma)(\star) \\
\delta_3 & = & (q_f, m_{init}, \gamma) & \xrightarrow{p_T} & (q_d, m_T, down)(\star) \\
\delta_4 & = & (q_d, m_A, \gamma) & \to & (q_d, m_A, down) \\
\delta_5 & = & (q_d, m_B, \gamma) & \to & (q_d, m_B, down) \\
\delta_6 & = & (q_d, m_F, \bot) & \to & (q_s, m_S, up_\gamma) \\
\delta_7 & = & (q_s, m_A, \gamma) & \xrightarrow{p_A} & (q_s, m_R, up_\gamma)(\star) \\
\delta_8 & = & (q_s, m_B, \gamma) & \xrightarrow{p_B} & (q_s, m_R, up_\gamma)(\star) \\
\delta_9 & = & (q_s, m_T, \gamma) & \to & (q_d, m_R, down) \\
\delta_{10} & = & (q_d, m_R, \gamma) & \to & (q_d, m_R, down) \\
\delta_{11} & = & (q_d, m_S, \bot) & \to & (q_d, m_R, down)
\end{array}
$$

Figure 4.1: Rules of $\mathcal{A}_{COPY}$

to a restriction number $k \in \mathbb{N}$ throughout the chapter.

In the following, we begin by providing an overview of the methodology in Section 4.1, which extends the approaches used for pPDA and RMC by decomposing the infinitely many runs into finite, manageable "patterns". Subsequently, Section 4.2 offers a detailed introduction to these patterns and the principles of their inductive decomposition. Following this, two straightforward applications of these principles are presented: the analysis of the termination probability in Section 4.3, and the analysis of the expected termination time in Section 4.4.

We fix an example that will be progressively analysed throughout the chapter, thereby illustrating each concept introduced in a more concrete and intuitive manner.

Example 4.1 (Random COPY Language)**.** We consider a variant of generating the COPY language, $\{w\#w \mid w \in (A|B)^*\}$, in which a random printer generates the symbols A, B, and $\#$ with probabilities p_A, p_B, and p_T, respectively. Our objective is to determine the probability that when printing infinitely, the printer outputs a prefix of the form $w\#w$, where w is an arbitrary sequence (including the empty sequence) of As and Bs. In addition, we aim to calculate the expected length of such prefixes.

This problem can be formulated as the termination problem of a randomised Probabilistic Tree Stack Automaton (rPTSA), denoted as $\mathcal{A}_{COPY}$, whose transition rules are provided in Figure 4.1. Let $q_{init} = q_f$. These rules modify the standard COPY example (i.e., Example 3.8) by introducing probabilistic deviations during the traversal of the tree-stack. Specifically, if the next character to be printed does not correspond to the expected symbol recorded in the current node, the process diverges – this is represented by transitions marked in red. As in the previous examples (Examples 3.7 and 3.8), any unspecified probability of transitions is assigned to transitioning to an implicit dummy stack symbol γ_Ω, which symbolises divergence.

The terminating runs of this construction precisely mimic those introduced

earlier in Example 3.8. Therefore, there is a one-to-one correspondence between the terminating runs μ of the automaton and the successful printing of a string $s = w\#w$. Furthermore, by introducing deviations when mismatches occur, the weight of each run $wt(\mu)$ corresponds to the probability of printing a string s in the COPY language. Consequently, the termination probability $tp(\mathcal{A})$ exactly equals the probability of printing such prefixes.

Furthermore, in order to answer the question regarding the expected length, we distinctly mark the rules denoting the generation of characters (A, B or $\#$) with a $\star$, meaning that only these specific rules are counted as transition steps, which then ensures that the conditional expected termination time of the automaton exactly corresponds to the expected length of the desired prefixes. From a technical perspective, this can be implemented by defining a simple reward function that assigns a value of 0 to rules without the $\star$ mark and a value of 1 to those with it.

The subsequent analysis, which includes detailed solutions to the problems, will be presented in the following sections. In particular, Example 4.45 deals with the termination probability problem, which is the probability of printing the desired prefixes; while Example 4.51 addresses the (conditional) expected termination time, which refers to the expected length of such prefixes.

The results, along with the corresponding equation systems, have been verified using the PTSV tool, which will be introduced in Chapter 9. For instance, when $p_A = 0.7$, $p_B = 0.2$, and $p_T = 0.1$, PTSV yields the precise results $tp(\mathcal{A}_{COPY}) = 10/47$ and $ett(\mathcal{A}_{COPY}) = 1530/2209$, thus giving $cett(\mathcal{A}_{COPY}) = 153/47$.

4.1 Methodology Overview

The methodology we adopt for the analysis can be viewed as an extension of those used for RMC [109] and pPDA [87], as briefly introduced in Section 2.1.2. To gain a preliminary understanding of this methodology, one may first refer to the example provided therein, particularly the equation system displayed in Equation (2.7).

The methodology for RMC can be summarised as follows. The potentially infinite runs are grouped into a finite number of classes, with their weights *inductively related*. The weight of a class x is expressed as a polynomial equation involving the weights of other classes, including x itself — which we referred to as the *decomposition equation* of x. For example, in Section 2.1.2, the weights are expressed by subscripted x terms. Specifically, the weight of the runs from the entry of component **B** to its exit, $x_{(En_B,Ex_B)}$, is equated to $x_{(N_4,Ex_B)}$, the weight of the runs from node N_4 to the exit of **B**. These decomposition relations form a system of equations, e.g., Equation (2.7).

This decomposition relation is inherently inductive — x may be expressed as a polynomial involving itself or another class y, which is decomposed to a polynomial involving x. For example, $x_{(N_4,Ex_B)}$ is represented by the second polynomial in Equation (2.7), which includes $x_{(En_B,Ex_B)}$.

Once the equation system is constructed, it can be solved using the Existential fragment of the first-order arithmetic Theory of the Reals (EToR) (i.e., Theorem 2.8), yielding the target value, such as the probability of exiting component

B in Section 2.1.2.

In summary, the overarching approach can be outlined in three steps:

1. Identify a *finite* set of distinct classes of runs, $x_1, \ldots, x_n$, each corresponding to a unique set of runs, referred to henceforth as "patterns", which can be *inductively decomposed*. Specifically, each pattern x_i can be inductively expressed in terms of patterns, including itself, through a formula of the form $f_i(x_1, \ldots, x_n)$.

2. Construct a system of polynomial equations based on this inductive decomposition for the necessary quantities. Consider the weight as an illustrative case. For each pattern, its weight decomposition equation is obtained by applying the weight operator to both sides, yielding $wt(x_i) = wt(f_i(x_1, \ldots, x_n))$, where the right-hand side can be further transformed to be a polynomial whose variables are the weights of the patterns, $wt(x_1), \ldots, wt(x_n)$.

3. Solve the resulting polynomial system using the EToR method.

To handle rPTSA, we adopt a similar methodology; however, a significant challenge arises in the first step when we aim to identify a finite set of patterns of running behaviours of the automaton that could be inductively related. Unlike the case of RMC, where the patterns are naturally and intuitively derived from the structure of the formalism, the situation is more complex in rPTSA. This complexity is evident from the fact that, when moving upwards to visit a node on the tree-stack, the automaton's behaviour changes as the local memories (in this node and/or the nodes above) change.

This observation necessitates considering the behaviours within and above a node together in a *synchronised* way. In other words, the elements of the patterns we devise are not individual runs describing isolated visits to a node on the tree-stack until leaving it, but rather sequences of synchronised "run segments", where each segment describes such a visit. This sequence of run segments, taken together, captures *all* the behaviours within and above this node in a run. Hence, in such a synchronised sequence of run segments, the former segments affecting the potential behaviors of the latter. Let us present an example to fully illustrate this concept.

4.1.1 Motivating Example

To illustrate the concept behind the patterns adopted in this work, consider a typical terminating run as depicted in Figure 4.2, which intuitively produces the string $ABAA\#ABAA$. To clarify the idea, the figure emphasises the movement of the stack pointer with arrows, while the nodes on the tree-stack are represented by horizontal (dashed and undashed) lines.

Using this illustrative figure, let us focus on the node two levels above the root, which is highlighted by a box encapsulating its stack status and an undashed line. We will refer to this as the "subject node". The run segments within and above the subject node are emphasised by arrows with a deeper colour, including all

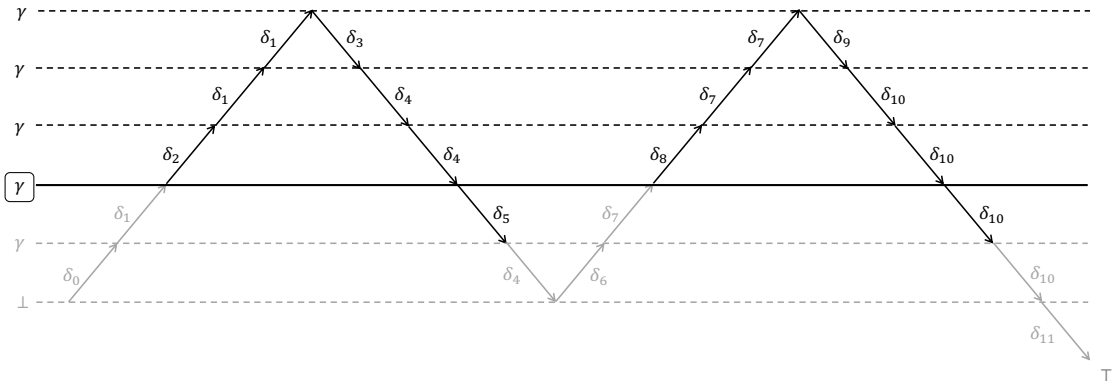

Figure 4.2: An Illustrative Run of $\mathcal{A}_{COPY}$

transitions above the line and two additional transitions with an immediate *down* from the line. Formally, the sequence of paths for the two run segments is:

$$\begin{bmatrix} [\delta_2, \delta_1, \delta_1, \delta_3, \delta_4, \delta_4, \delta_5], \\ [\delta_8, \delta_7, \delta_7, \delta_9, \delta_{10}, \delta_{10}, \delta_{10}] \end{bmatrix} \tag{4.2}$$

Notably, the latter segment is influenced by the former segment: the first three rules in the latter segment, δ_8, δ_7 and δ_7, are applicable due to the execution of δ_2, δ_1 and δ_1 in the former segment, as they set the local memories to m_B, m_A and m_A for the subject node and the node above it, respectively. In contrast, for the former segment, the applicable rules are predetermined, as the subject node is newly created, and thus the subtree from this node must be a single node with local memory m_{init}.

Given the 2-restriction of the automaton $\mathcal{A}_{COPY}$, it becomes evident that, for every node, the total number of possible run segments is *finite*. Extending this further, for any k-restricted rPTSA, the possible number of segments for each node must be $\leq k$.

This sequence of run segments will be referred to as a *pre-run* below, with each segment within it named a "trip" — intuitively capturing the notion of a visit from entering the node to leaving it. Moreover, the underlying rule sequence, i.e., the path for each segment within, will be termed a *pre-path*, of which Equation (4.2) is a suitable example.

4.1.2 Patterns: From pPDA to rPTSA

Returning to our methodology with the illustrative example in mind, we utilise the sets of pre-runs, which capture the running segments within and above a node, as patterns. Further, to inductively decompose these patterns for the construction of an equation system for the desired quantities, we must further qualify such sets of pre-runs occurring within and above a node by their "running interfaces", as per the practice for pPDA.

Recall that in pPDA, a pattern is of the form $\langle q\gamma q' \rangle$, which describes a set of runs within and above a node marked by γ, beginning with q and concluding with

the *first* descent below γ with state q'. Intuitively, pPDA patterns further qualify the behaviour of a node by stipulating the entering and leaving "interface" as q and q'. In a similar vein, in our case, we may further qualify the behaviours by specifying the "interfaces" of the visits collectively. More specifically, we can state that the pre-run in the illustrative example of Figure 4.2 (whose rules are in Figure 4.1) falls within the pattern, expressed in the same spirit as pPDA patterns, as:

$$\langle q_f \gamma q_d, q_s \gamma q_d \rangle$$

More generally, the patterns for rPTSA can be understood as follows:

$$\langle q_1 \gamma q_2, q_3 \gamma q_4, \ldots, q_{2n-1} \gamma q_{2n} \rangle \tag{4.3}$$

These new patterns now represent sets of pre-runs, where each pre-run takes the form $[\mu_1, \ldots, \mu_n]$. Each run μ_i, analogous to the runs in the pPDA pattern $\langle q_{2i-1} \gamma q_{2i} \rangle$, begins at the node marked by γ with state q_{2i-1} and ends with the first descent from the node with state q_{2i}. However, there is an additional natural requirement: each μ_i must start with a configuration identical to that at the end of the preceding run μ_{i-1}, with the starting configuration of μ_1 representing the creation of the subject node.

Taking one step further, by isolating the common stack symbol γ and transforming the control state sequence into a single, continuous sequence, we can rewrite the above representation as Equation (4.3) in a more succinct and intuitive way, namely, $\mathit{Trips}^{\gamma}_{[q_1,\ldots,q_{2n}]}$, which is exactly the form we adopt below.

Additionally, although not utilised in this chapter for termination analysis, we introduce the pattern $\mathit{Up}^{\gamma}_{[q_1,\ldots,q_{2n},q_{2n+1}]}$ for further analysis below. This pattern describes the behaviour where, after the final *up* visit to the subject node with state q_{2n+1}, the runs remain infinitely within and above the subject node without descending from it. This corresponds to the $\langle q\gamma \uparrow \rangle$ patterns in the pPDA literature. This type of pattern is introduced due to its similarity in decomposition to the *Trips* patterns, and will be heavily utilised in subsequent chapters, such as in model checking.

4.1.3 Chapter Outline

Finally, by identifying the set of patterns, this chapter delves into the inductive decomposition of these patterns to construct equation systems for desired quantities, such as weights for termination probability. In the remainder of this chapter, we first formally present the patterns and their decomposition principles in Section 4.2, and then apply these principles to resolving the termination probability in Section 4.3, concluding with an application to the analysis of the expected termination time in Section 4.4.

4.2 Decomposing rPTSA Runs

In this section, we formalise the patterns of pre-runs introduced earlier and explore their inductive relations. We begin with the formalisms of pre-runs, patterns, and related concepts in Section 4.2.1.

Next, we discuss the decomposition of these patterns in Section 4.2.2, where we formulate a decomposition equation for patterns. A key concept, *synchronisation paths*, emerges here and will be used extensively throughout this chapter and the work. Thus, we provide a clearer formalism for synchronisation paths in Section 4.2.3.

Finally, we present a proof of the decomposition principles in Section 4.2.4, and briefly introduce the decomposition equation system derived from the patterns' decomposition equations in Section 4.2.5.

4.2.1 Pre-Runs and Patterns for rPTSA

With the significance mentioned above, we formally present the key formalisms to instantiate our methodology. Let us begin with the fundamental concept – the pre-runs.

Pre-Runs A *pre-run*, denoted by $\vec{\mu}$, is a *finite* sequence of *non-terminating* runs. It is termed a *finite pre-run* if every run within it is finite.

The ***weight*** of a *finite* pre-run, denoted by $wt(\vec{\mu})$, is calculated as the product of the weights of each run in the sequence:

$$wt(\vec{\mu}) := \prod_{\mu \in \vec{\mu}} wt(\mu)$$

An empty sequence is assigned a default weight of 1. [1] Additionally, when a pre-run is finite, its *transition length* and *total accumulated reward*, denoted by $\ell(\vec{\mu})$ and $\mathsf{rwd}_f(\vec{\mu})$ (for a given reward function f), respectively, are determined by summing the corresponding values of each run within it.

Moreover, we define that $\vec{\mu}$ is *k-restricted* if and only if the *cumulative sum* of the smallest restriction numbers *across all runs* within it is $\leq k$. This means that the k-restriction for a pre-run is not on an individual run basis but on the aggregate level by viewing all the runs as a whole. Such a requirement is intuitive if one views a pre-run as essentially an incomplete run, leaving gaps between consecutive run segments – as intuitively presented in the overview.

For sets of pre-runs, we delineate the following definitions.

Definition 4.4. For a set S of *finite* pre-runs, the *weight*, denoted by $wt(S)$, is determined by summing the weights of all constituent elements:

$$wt(S) := \sum_{\vec{\mu} \in S} wt(\vec{\mu}).$$

Furthermore, the *mean hitting time*, denoted by $mht(S)$, is defined as:

$$mht(S) := \sum_{\vec{\mu} \in S} wt(\vec{\mu}) \cdot \ell(\vec{\mu}).$$

Similarly, given a reward function f, the *expected total accumulated reward*, denoted by $etar_f(S)$, is calculated as:

$$etar_f(S) := \sum_{\vec{\mu} \in S} wt(\vec{\mu}) \cdot \mathsf{rwd}_f(\vec{\mu}).$$

[1] The elements within a pre-run should not be empty (otherwise violating the definition of a run), but a pre-run itself may be empty.

Pre-Paths Additionally, we introduce the concept of a *pre-path*, represented by $\vec{\phi}$, which constitutes a finite sequence of paths. In parallel to runs and paths, the function $\mathsf{path}(\vec{\mu})$ extracts the *underlying pre-path* from a given pre-run $\vec{\mu}$, thereby designating $\vec{\mu}$ as the *host pre-run* of the said pre-path. A prepath $\vec{\phi}$ is termed *finite* if every path within it is finite. Given a *finite* pre-path $\vec{\phi}$, its *weight* $wt(\vec{\phi})$ is defined as the product of the weights of all paths it encompasses. Similarly, its *transition length* $\ell(\vec{\phi})$ and *total accumulated reward* $\mathsf{rwd}_f(\vec{\phi})$, with respect to a *simple* reward function f, are naturally extended as the sum of the corresponding attributes across all the paths included in $\vec{\phi}$.

Further, for a set S of *finite* pre-paths, its *weight* $wt(S)$ is the sum of weights across all elements inside. While its *mean hitting time* $mht(S)$ is the weighted sum of the transition lengths across the elements, that:

$$mht(S) := \sum_{\vec{\phi} \in S} wt(\vec{\phi}) \cdot \ell(\vec{\phi})$$

As an extension, its *expected total accumulated reward* $etar_f(S)$ wrt a *simple* reward function is then given by summing the weighted rewards:

$$etar_f(S) := \sum_{\vec{\phi} \in S} wt(\vec{\phi}) \cdot \mathsf{rwd}_f(\vec{\phi})$$

Cohesiveness We introduce several key auxiliary definitions to further qualify the types of pre-runs and pre-paths necessary for defining patterns.

Definition 4.5. A *trip* refers to a finite run from a configuration $(q, t, \rho\gamma)$ to a configuration (q', t', ρ), where the stack pointer in each configuration is distinct from ρ, except in the final configuration. A *soar* is a run, finite or infinite, from a configuration $(q, t, \rho\gamma)$ that never reaches a configuration with stack pointer ρ. A path is called a *trip path* if it is hosted by some trip; similarly, a *soar path* is hosted by some soar.

Definition 4.6. A pre-run $\vec{\mu}$ consists of a sequence of trips, each starting from the same stack pointer ρ, and may optionally end with a soar from ρ. The pre-run $\vec{\mu}$ is called *cohesive* if, for every pair of contiguous runs μ and μ', the tree-stack of the final configuration of μ is the same as the tree-stack of the starting configuration of μ'. A *cohesive pre-path* is a pre-path $\vec{\phi}$ hosted by some cohesive pre-run $\vec{\mu}$. The starting configuration of the first run in $\vec{\mu}$ is called an enabling configuration for $\vec{\phi}$. Especially, an empty pre-path is defined to be enabled by any configuration.

These definitions yield several straightforward yet significant observations: Every host of a trip path must be a trip, and every host of a soar path must be a soar. [2] However, a cohesive pre-path can be hosted by a non-cohesive pre-run. For instance, in the forthcoming example of a pattern (Figure 4.3), if the bottom memory of the first configuration in the depicted second runs is modified to m_A, it will indeed

[2]Notably, given a path, one can track the relative position of the stack pointer with respect to the starting configuration.

host the same paths as shown in the figure, but the overall pre-runs become non-cohesive. Given a fixed enabling configuration c, the cohesive pre-run (starting from c) hosting a particular cohesive pre-path is unique, since the uniqueness of rule application and the definition of cohesiveness determine every configuration.

Interface Sequence Furthermore, we define the *interaction interface* (or simply *interface*) of a pre-run $\vec{\mu}$, which consists of a finite number of trips and possibly ends with a soar — where only the final soar may be infinite — denoted $\mathsf{Interface}(\vec{\mu})$, as the sequence of control states determined by the start and end configurations of each trip (or solely the start configuration if the run is a soar) within the pre-run. Formally:

$$\mathsf{Interface}(\vec{\mu}) := \mathsf{concat}\left(\left[s \;\middle|\; \mu \in \vec{\mu}, s := \begin{cases} [q^{(\mathsf{hd}(\mu))}, q^{(\mathsf{last}(\mu))}] & \mu \text{ is a trip} \\ [q^{(\mathsf{hd}(\mu))}] & \mu \text{ is a soar} \end{cases}\right]\right)$$

This function is naturally extended to *cohesive* pre-paths – $\mathsf{Interface}(\vec{\phi})$ is simply defined to be $\mathsf{Interface}(\vec{\mu})$ for any pre-run $\vec{\mu}$ that hosts $\vec{\phi}$, note that it is straightforward to verify that they all yeild the same result.

Pre-Run Patterns As introduced above in the overview, a pre-run pattern describes the behaviours of the rPTSA within and above a node on the tree-stack, further qualified by the interaction interfaces of each run segment. Formally, a pre-run pattern is a collection of pre-runs that is k-restricted, characterised by a stack symbol γ for the stack symbol of the node, and a finite sequence of control states D with length $< 2k$, referred to as the *(downward) interaction sequence*, representing the interaction interfaces.

Specifically, there are two types of patterns: *Trips* patterns, ranged over by $Trips^{\gamma}_{D}$, and *Up* patterns, ranged over by Up^{γ}_{D}, with both collectively ranged over by $TripsOrUp^{\gamma}_{D}$. A pattern $TripsOrUp^{\gamma}_{D}$ consists of cohesive k-restricted pre-runs $\vec{\mu}$ that satisfy: (1) each run starts from the common stack pointer $[\gamma]$; (2) $\mathsf{Interface}(\vec{\mu}) = D$; and (3) the starting tree-stack of the first run is $\{\varepsilon \mapsto m_{init}, [\gamma] \mapsto m_{init}\}$. In a $Trips^{\gamma}_{D}$ pattern, D is of *even* length and the pre-runs consist only of trips, while in a Up^{γ}_{D} pattern, D is of *odd* length and the pre-runs end with an infinite soar.

Although the pattern type can be inferred from the parity of the interaction sequence D, we explicitly distinguish between the two, as they play fundamentally different roles in the analysis.

Example 4.7. Consider the automaton from Example 3.7. Let us examine the pattern $Trips^{\gamma_{AC}}_{[q_A, q_{back}, q_C, q_{back}]}$. It encompasses pre-runs of the form illustrated in Figure 4.3, where the arrangement of upper and lower runs typifies a standard pre-run in the pattern.

It is important to note that these pre-runs are *NOT* part of the initial runs of the original rPTSA, as depicted in Figure 3.3. Specifically, a pre-run in $Trips^{\gamma_{AC}}_{[q_A, q_{back}, q_C, q_{back}]}$ initiates with a configuration wherein the stack pointer is positioned one level above the root, pointing to the subject node, with the local memory of the root fixed to m_{init}. In the first run of the pre-run, the automaton produces A repeatedly for n iterations and returns to the *root* node.

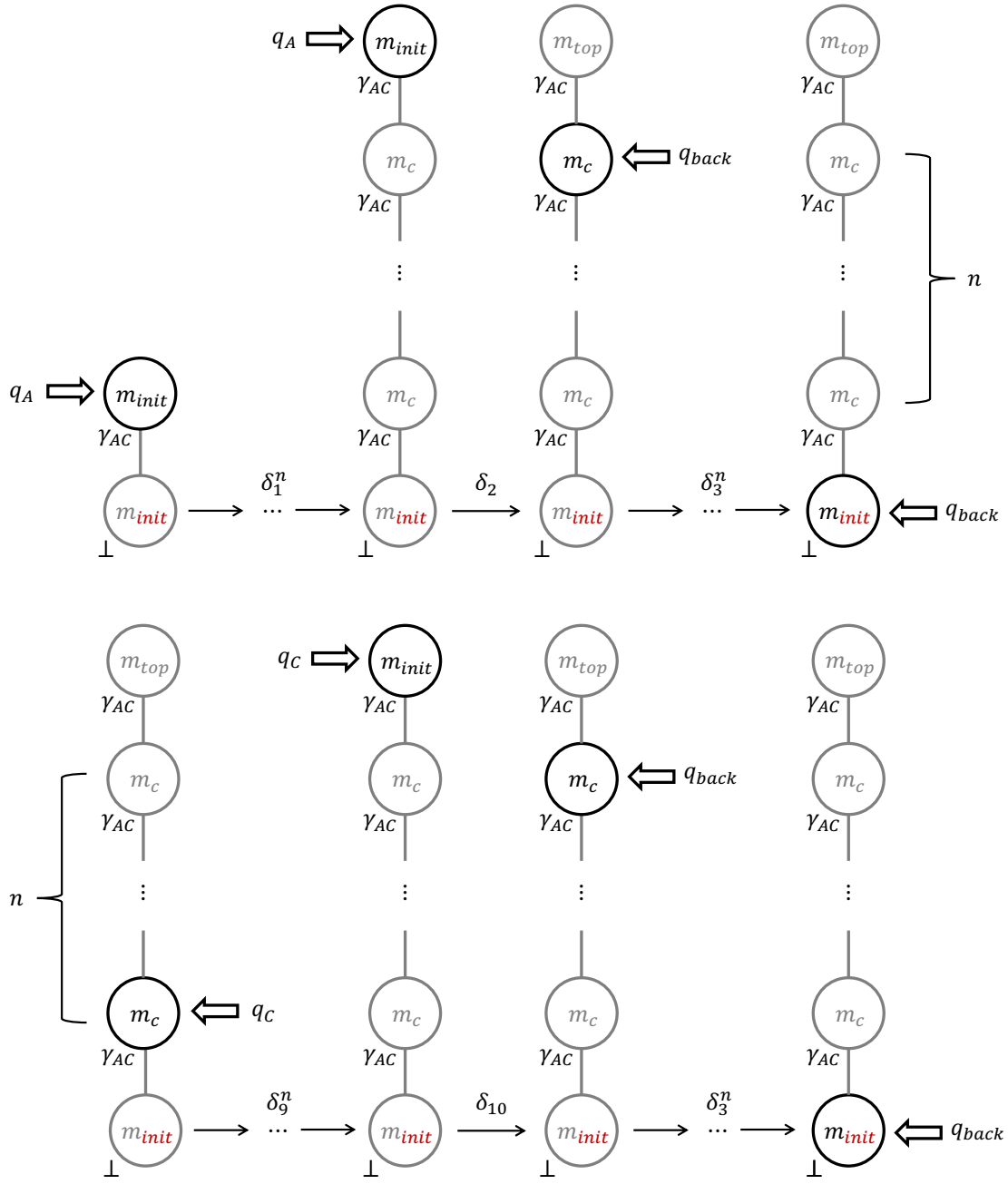

Figure 4.3: Pattern $Trips^{\gamma_{AC}}_{[q_A, q_{back}, q_C, q_{back}]}$ of Example 3.7

Thereafter, the second run of the pre-run commences with the identical tree-stack as at the conclusion of the first run, the stack pointer positions again at one level above the root in direction γ_{AC}, with the new control state q_C, as specified by the definition of patterns. The second run then traverses the branch γ_{AC} once more, generating C in the same quantity n as A, and returns to the root node, concluding the second run.

Weights of Patterns Then, we inspect the weight of a given pattern $\mathit{TripsOrUp}_D^\gamma$. Firstly, note that as a set of finite pre-runs, the weight of any *Trips* pattern is already well-defined (i.e., Definition 4.4).

Conversely, for a Up_D^γ pattern, its weight has not yet been established due to the terminal run of every pre-run within being *infinite*. Herein, a more generalised definition of *weight* is presented for a particular category of infinite pre-run sets.

Definition 4.8. Given a set of infinite pre-runs S, the concept of weight is only defined under the following conditions:

1. For every pre-run in S, exactly the final run therein is infinite, while all preceding runs are finite; and,
2. For every finite pre-run $\vec{\mu}$, all the last infinite runs postpending $\vec{\mu}$, namely, $S_{\vec{\mu}} = \{\mu \mid \vec{\mu} +\!\!+ [\mu] \in S\}$, must start with the same configuration.

Should these conditions be met, the weight of S is delineated as follows:

$$wt(S) := \sum_{\vec{\mu} \text{ is finite}} wt(\vec{\mu}) \cdot \mathbb{P}\left[\{\mu \mid \vec{\mu} +\!\!+ [\mu] \in S\}\right] \tag{4.9}$$

It is pertinent to note that the probability measure is well-defined pursuant to the second condition.

Given the definition of *Up* patterns, the previously mentioned conditions are invariably satisfied, thereby rendering their weights as well-defined. Moreover, when this definition is tailored specifically to the *Up* patterns, the following more intuitive and explicit definition can be posited for patterns denoted as $\mathit{Up}_{D +\!\!+ [\varphi]}^\gamma$:

$$wt(\mathit{Up}_{D +\!\!+ [\varphi]}^\gamma) := \sum_{\vec{\mu} \in \mathit{Trips}_D^\gamma} wt(\vec{\mu}) \cdot \mathbb{P}\left[\left\{\mu \,\middle|\, \vec{\mu} +\!\!+ [\mu] \in \mathit{Up}_{D +\!\!+ [\varphi]}^\gamma\right\}\right] \tag{4.10}$$

Patterns of Pre-Paths Given the uniform starting configuration of pre-runs within a specific pattern, a direct one-to-one correspondence emerges between $\mathit{TripsOrUp}_D^\gamma$ and the set produced by applying path pointwise, i.e., $\{\mathsf{path}(\vec{\mu}) \mid \vec{\mu} \in \mathit{TripsOrUp}_D^\gamma\}$. This indicates that path functions as a *bijection* between these two sets. Following the conventions, this pointwise applied set is referred to as $\mathsf{path}(\mathit{TripsOrUp}_D^\gamma)$, and we name it the *pattern of pre-paths*. The observation of bijection can be formalised as below.

Lemma 4.11. *For any pattern $\mathit{TripsOrUp}_D^\gamma$, and any pre-path $\vec{\phi}$ within its corresponding set of pre-paths $\mathsf{path}(\mathit{TripsOrUp}_D^\gamma)$, there exists exactly one pre-run $\vec{\mu} \in \mathit{TripsOrUp}_D^\gamma$ such that $\vec{\phi} = \mathsf{path}(\vec{\mu})$.*

Proof. Recall that a fixed starting configuration coupled with a path uniquely determines a run. Consider a pre-run $\vec{\mu} \in \mathit{TripsOrUp}_D^\gamma$ with $\vec{\phi} = \mathsf{path}(\vec{\mu})$. This principle immediately ensures that the first run in $\vec{\mu}$ is uniquely determined as the starting configuration of all pre-runs in a pattern is fixed. In other words, no pre-run in the same pattern hosting $\vec{\phi}$ has a different first run. To inspect inductively, if

the prefix until the ith run in $\vec{\mu}$ is uniquely determined, given that the starting configuration of the $(i+1)$th run in $\vec{\mu}$ is uniquely determined by this prefix, this run at the $(i+1)$th position is also uniquely determined. □

Directly following from the result, for any $Trips_D^\gamma$ pattern, the following equations hold:

$$wt(Trips_D^\gamma) = wt(\mathsf{path}(Trips_D^\gamma)) \tag{4.12}$$
$$mht(Trips_D^\gamma) = mht(\mathsf{path}(Trips_D^\gamma)) \tag{4.13}$$
$$etar_f(Trips_D^\gamma) = etar_f(\mathsf{path}(Trips_D^\gamma)) \tag{4.14}$$

where f is any *simple* reward function.

Example 4.15. Consider the pattern $Trips^{\gamma_{AC}}_{[q_A, q_{back}, q_C, q_{back}]}$ as introduced above in Example 4.7. Its pre-path pattern, that $\mathsf{path}(Trips^{\gamma_{AC}}_{[q_A, q_{back}, q_C, q_{back}]})$, contains the following pre-path for all $n \geq 0$:

$$[\delta_1^n \delta_2 \delta_3^n, \delta_9^n \delta_{10} \delta_3^n]$$

4.2.2 Inductively Decomposing Patterns

We now explore the decomposition relationships between patterns, with the aim of representing one pattern in terms of others. This allows us to quantify the necessary attributes of the patterns, such as weights or expected total accumulated rewards, using a system of polynomial equations. Each equation is derived from the decomposition relations.

However, the decomposition method is not immediately intuitive, due to the complexity of the formalisms involved. To clarify the approach, we begin by illustrating the rough idea through a *by-level* decomposition of a single pre-run. We then proceed with a more general discussion of the overarching principles of decomposition, which will highlight the technical difficulties in decomposing patterns themselves. This challenge motivates our shift in focus towards the decomposition of pre-path patterns, for which we will subsequently establish a decomposition equation, i.e., Equation (4.17), as the main result and the focus of the remainder of this section.

Decomposing a Single Pre-Run

Initially, we describe the decomposition of a single pre-run within a given pattern, as a preliminary step towards the broader decomposition of an entire pattern. We will focus on a specific pattern $TripsOrUp_D^{\gamma_0}$ and a particular pre-run $\vec{\mu} = [\mu_1, \ldots, \mu_n]$ contained within it. Recall that a pattern describes a set of pre-runs *within-and-above* a specific node on the path $[\gamma_0]$. We refer to this node as the *subject node*.

The decomposition is based on the observation that a pre-run $\vec{\mu}$ can be divided into two segments: the transitions occurring *within* the subject node $\vec{\mu}_s$, and those *above* it in each direction γ, denoted $\vec{\mu}_\gamma$. This concept can be formalised as follows:

$$\vec{\mu} = \mathsf{SyncR}(\vec{\mu}_s, \{\gamma \mapsto \vec{\mu}_\gamma \mid \gamma \in \Gamma\}) \tag{4.16}$$

Here, the function SyncR serves as a "recovery" function to the intuitive split process — it collects the transitions within and above the subject node and produces the original pre-run.

In the following, we provide a more formal account of each function and prove this decomposition equation. These formalisations closely mirror the intuitive understanding and serve as a foundation for the formal proof of the decomposition.

Subject Node Transitions To be more specific, $\vec{\mu}_s$ denotes the pre-run obtained by extracting all transitions occurring *within* the subject node leading to *leave* it. Formally, we define $\vec{\mu}_s$ as $\mathsf{CurNode}(\vec{\mu})$, with

$$\mathsf{CurNode}(\vec{\mu}) := \mathsf{concat}\Big([\mathsf{CurNode}'(\mu) \mid \mu \in \vec{\mu}]\Big)$$

where $\mathsf{CurNode}'(\mu)$, applied to each run μ in $\vec{\mu}$, is described as follows: First, we identify all occurrences of configurations in μ with the stack pointer $[\gamma_0]$, enabling it to be expressed as:

$$c_1 c'_1 \dots c_2 c'_2 \dots c_i c'_i \dots c_n c'_n \dots$$

Here, the configurations $\{c_i\}_1^n$ represents all occurrences of configurations with the stack pointer $[\gamma_0]$. It is ensured by the pre-run requirements of patterns that the sequence begins with the first occurrence at the start of μ and that c'_n exists. Furthermore, the existence of such a finite number n is affirmed by the k-restriction. Subsequently, $\mathsf{CurNode}'(\mu)$ is defined as $\Big[[c_1, c'_1], [c_2, c'_2], \dots, [c_n, c'_n]\Big]$.

Upward Branches Transitions Conversely, for each $\gamma \in \Gamma$, $\vec{\mu}_\gamma$ comprises runs initiating from the node above the subject node in direction γ (hence encapsulating transitions occurring at or above the node at stack pointer $[\gamma_0, \gamma]$) and either terminating in the subject node for the first time or bypassing the subject node (wherein the run is infinite). Formally, we define $\vec{\mu}_\gamma := \mathsf{Abv}(\vec{\mu}, [\gamma_0, \gamma])$, where $\mathsf{Abv}(\vec{\mu}, x)$ extracts transitions of $\vec{\mu}$ made within and above the node at route x. We express $\mathsf{Abv}(\vec{\mu}, x)$ as:

$$\mathsf{Abv}(\vec{\mu}, x) := \mathsf{concat}\Big([\mathsf{Abv}'(\mu, x) \mid \mu \in \vec{\mu}]\Big)$$

This formulation applies Abv' to each run in $\vec{\mu}$ collectively. The function $\mathsf{Abv}'(\mu, x)$, applying for a single run, is defined as follows:

If $x = \varepsilon$, then, it becomes simply the run itself, because every transition must happen within and above the bottom. Otherwise, let x be denoted as $\rho\gamma$, then we initially identify all occurrences of contiguous configurations in the form $(-, -, \rho)(-, -, \rho\gamma)$, thereby allowing the run μ to be written as:

$$\dots c_1 c'_1 \mu_1 c_2 c'_2 \mu_2 \dots c_i c'_i \mu_i \dots c_n c'_n \mu_n$$

where the contiguous configuration pairs $c_i c'_i$ for $i \in \{1, \dots, n\}$ signify all identified occurrences, and μ_i represents the run ensuing the ith occurrence of such pairs until the subsequent occurrence or the remainder of the run after the final occurrence.

Similar to the previous scenario, the existence of a finite number n (specifically, $n \leq k$) is assured by the k-restriction.

Subsequently, we define a series of runs μ'_i for $i \in \{1, \ldots, n-1\}$ as follows: If μ_i excludes any configuration with the stack pointer pointing to the node at route ρ, then $\mu'_i := \mu_i \mathscr{c}_{i+1}$; otherwise, μ'_i is the prefix of μ_i up to and including the first occurrence of a configuration with the stack pointer being ρ.

Moreover, we specify μ'_n such that: if μ_n encompasses at least one configuration with the stack pointer ρ, then μ'_n is the finite prefix of μ_n up to and including the first occurrence of such a configuration; otherwise, μ'_n is simply μ_n. Note that the latter case signifies that μ_n is an infinite run as the postfix of μ, which concludes the $\vec{\mu}$ in an *Up* pattern.

Finally, $\mathsf{Abv}'(\mu, x)$ is defined as $[\mathscr{c}'_1 :: \mu'_1, \ldots, \mathscr{c}'_n :: \mu'_n]$.

The Recovery Function The function $\mathsf{SyncR}(\vec{\mu}', f)$ effectively reverses the decomposition process above, ensuring Equation (4.16) holds. To formalise this, it is necessary to first delineate certain properties of its parameters, inferred from the decomposition procedure. They are at the same time prerequisites of calling the function – that arguments not adhering to these prerequisites yield the special value $\mathsf{Undefined}$, thereby rendering the function total. Further, to ease the symbolic burden, we denote the function initialises the call of SyncR, that $\{\gamma \mapsto \vec{\mu}_\gamma \mid \gamma \in \Gamma\}$ to be $f_{\vec{\mu}}$.

- The argument $\vec{\mu}'$ is expected to take the form $\Big[[\mathscr{c}_1, \mathscr{c}'_1], \ldots, [\mathscr{c}_n, \mathscr{c}'_n]\Big]$ for some finite n, where each pair $[\mathscr{c}_i, \mathscr{c}'_i]$ satisfies $\rho^{(\mathscr{c}_i)}$ is not empty, and let it be written as $\rho\gamma$, then $\rho^{(\mathscr{c}'_i)}$ is either ρ or $\rho\gamma\gamma'$ for some $\gamma' \in \Gamma$.

  For the initial call, i.e., $\mathsf{SyncR}(\vec{\mu}_s, f_{\vec{\mu}})$, this property is directly derived from the definitions of $\mathsf{CurNode}$ and the k-restriction.

- Given $\vec{\mu}'$ as $\Big[[\mathscr{c}_1, \mathscr{c}'_1], \ldots, [\mathscr{c}_n, \mathscr{c}'_n]\Big]$, then for each $\gamma \in \Gamma$, the following two conditions hold:

  Length Match the magnitude $|f(\gamma)|$ is precisely the count of $\mathscr{c}'_i$ in $\vec{\mu}'$ where $\rho^{(\mathscr{c}'_i)} = \rho^{(\mathscr{c}_i)}\gamma$. Essentially, the number of up_γ operations in $\vec{\mu}'$ equals the length of $f(\gamma)$.

  Beginning Configuration Match denoting the jth up_γ operation in $\vec{\mu}'$ by $[\mathscr{c}_{j_\gamma}, \mathscr{c}'_{j_\gamma}]$, then $f(\gamma)[j]$ commences with $\mathscr{c}'_{j_\gamma}$.

  The legitimacy of the initial call, that $\mathsf{SyncR}(\vec{\mu}_s, f_{\vec{\mu}})$, is affirmed through an analysis of $\vec{\mu}_s$ and $\vec{\mu}_\gamma$ definitions. Noticing that both functions aggregate results from individual runs, their properties can be validated by scrutinising the definitions of $\mathsf{CurNode}'$ and AtDir' for each run.

$\mathsf{SyncR}(\vec{\mu}', f)$ is then defined through a case analysis on $\vec{\mu}'$. In the base case, $\mathsf{SyncR}(\varepsilon, -)$ returns the empty sequence ε. While in the inductive case, guided by the first property, the scenario is of the form $\mathsf{SyncR}([\mathscr{c}, \mathscr{c}'] :: \vec{\mu}'_s, f)$. Its result is determined by the transition direction from $\mathscr{c}$ to $\mathscr{c}'$:

- If $\rho^{(c')} \sqsubseteq \rho^{(c)}$, indicating a *down* operation, the result is simply: $[c, c'] :: \mathsf{SyncR}(\vec{\mu}'_s, f)$.

  Here, the call $\mathsf{SyncR}(\vec{\mu}'_s, f)$ upholds the declared properties, verifiable directly. The first property remains valid as $\vec{\mu}'_s$ is merely a postfix of $\vec{\mu}'$, and the second, by inherence, as $\vec{\mu}'_s$ retains the identical count of *up* operations to each direction in Γ.

- Alternatively, there must exist a $\gamma' \in \Gamma$ such that $\rho^{(c')} = \rho^{(c)}\gamma'$, indicating an *up* operation to direction γ', and by the second property, it is feasible to decompose $f(\gamma')$ as $\mu' :: \vec{\mu}'$ with μ' starting with c'.

  Subsequently, the result of $\mathsf{SyncR}(\vec{\mu}'_s, f[\gamma' \mapsto \vec{\mu}'])$ determines the next action:

  - If the result is $\mu_r :: \vec{\mu}_r$, then it is $((c :: \mu') +\!\!+ \mu_r) :: \vec{\mu}_r$;
  - otherwise, if the result is empty, it becomes $[c :: \mu']$.

  The recursive call $\mathsf{SyncR}(\vec{\mu}'_s, f[\gamma' \mapsto \vec{\mu}'])$ ensures consistency and avoids producing $\mathsf{Undefined}$, leveraging the properties inherited from the inputs. The first property remains valid as $\vec{\mu}'_s$ constitutes a postfix of $\vec{\mu}'$, preserving the structure and sequence of operations.

  Regarding the second property, with $f' := f[\gamma' \mapsto \vec{\mu}']$, both the sequence $\vec{\mu}'_s$ and the length of $f'(\gamma')$ see a reduction in the count of $up_{\gamma'}$ operations by one relative to the original inputs, hence maintaining the length matching condition for γ'. This deliberate reduction, achieved by removing the first occurrence of an $up_{\gamma'}$ operation alongside the corresponding initial element in $f(\gamma')$, preserves the relative positions of subsequent $up_{\gamma'}$ operations, so upholding the beginning configuration matching condition for γ'. The case for any γ'' distinct from γ' holds straightforwardly as they remain unchanged.

With these definitions, we present the following proof to substantiate the constructions:

Proof (Equation (4.17)*).* This can be demonstrated through straightforward induction on the number of transitions within the subject node in $\vec{\mu}$ – that is, the length of $\vec{\mu}_s$. □

The Decomposition Principles

Building upon the decomposition of a single pre-run formalised in Equation (4.16), we now extend our focus to the broader principles underpinning our methodology for patterns — the *by-level* decomposition principle. Notably, our focus here will be on conveying the intuition behind the principles, reserving the technical proofs and constructions for subsequent discussions.

In essence, in the same spirit as Equation (4.16), we decompose a pattern into two components: a set denoting transitions at the subject node, and sets of pre-runs above it for each upward direction $\gamma \in \Gamma$. This process is inherently recursive, as each set of pre-runs for a direction γ is further decomposed based on transitions within the

child node at that direction and the pre-runs further above it. In this way, we obtain a method to represent pre-runs in a pattern in a by-level decomposing manner.

Central to inductively characterising this decomposition with patterns is the concept of "interaction interface", referring to the control states at child node visits and returns, analogous to $\mathsf{Interface}(\vec{\mu}_\gamma)$ for all γ in Equation (4.16). The key observation is that when such interfaces for each direction are fixed, the behaviours within the sub-tree rooted in each child node and those in the subject node are essentially *non-interfering*. This indicates that the combination of any subject node behaviours and any upward behaviours for each γ form a proper pre-run still within the pattern. Notice that the latter essentially corresponds to the same intuition as a pattern — the behaviours within and above a node with contracted interface D and stack symbol γ.

Moreover, we may formalise this concept of upward interfaces as a function $\mathbb{D}$, assigning to each γ an interaction interface D. The k-restriction ensures D's length is $\leq 2k$, guaranteeing a finite number of possible $\mathbb{D}$. Thus, when extending for patterns, the by-level decomposition principle suggests that we represent a pattern as an aggregation of local behaviours in the subject node satisfying a given $\mathbb{D}$, along with a pattern $\mathit{TripsOrUp}^\gamma_{\mathbb{D}(\gamma)}$ for each γ.

However, this formulation faces a technical challenge: the pre-runs $\vec{\mu}_\gamma$ do not directly belong to $\mathit{TripsOrUp}^\gamma_{\mathsf{Interface}(\vec{\mu}_\gamma)}$, as they neither qualify as cohesive pre-runs nor start from the required fixed initial configuration. This challenge is then addressed by the observation that the "behaviours within-and-above" a node are *fully* characterised by the underlying rule sequences of the pre-runs — the pre-paths, as demonstrated by Lemma 4.11. Crucially, the pre-paths are capable of describing these behaviours independent of the specific starting configuration, thereby circumventing the technical challenge posed by the stringent requirements on initial configurations. Formally, for each γ:

$$\mathsf{path}(\vec{\mu}_\gamma) \in \mathsf{path}(\mathit{TripsOrUp}^\gamma_{\mathsf{Interface}(\vec{\mu}_\gamma)})$$

This insight leads us to focus on pre-path patterns, which encapsulate all necessary information to characterise the patterns themselves. Notably, this contrasts with the cases in pPDA and RMC where patterns directly correspond to runs.

By applying the by-level decomposition principle to pre-path patterns, we now arrive at our final destination: a pre-path pattern can be decomposed into sets characterised by the upward interfaces $\mathbb{D}$. The pre-paths of each set are further decomposed into a set of pre-paths describing the transitions within the subject node that fulfill the interfaces $\mathbb{D}$, known as the *synchronisation paths*, and the pre-path patterns $\mathit{TripsOrUp}^\gamma_{\mathbb{D}(\gamma)}$ for all γ.

Decomposing a Single Pre-Path

Equipped with the general principle discussed, we now examine the decomposition of a single pre-path pattern. It is achieved by applying the function $\mathsf{path}(-)$ and adapting the recovery function for pre-paths.

$$\mathsf{path}(\vec{\mu}) = \mathsf{Sync}\Big(\mathsf{path}(\vec{\mu}_s), \mathsf{path} \circ f_{\vec{\mu}}\Big) \tag{4.17}$$

Recalling the function composition, $\mathsf{path} \circ f_{\vec{\mu}}$ is defined as $\{\gamma \mapsto \mathsf{path}(\vec{\mu}_\gamma) \mid \gamma \in \Gamma\}$.

Our objective then is to define the function Sync satisfying the following equation, from which the validity of Equation (4.17) is immediately apparent.

$$\mathsf{path}\Big(\mathsf{SyncR}(\vec{\mu}_s, f_{\vec{\mu}})\Big) = \mathsf{Sync}(\mathsf{path}(\vec{\mu}_s), \mathsf{path} \circ f_{\vec{\mu}}) \tag{4.18}$$

Accordingly, the function $\mathsf{Sync}(\vec{\phi}, f)$, now tailored for pre-paths, comes also with certain prerequisites (/ properties) for its arguments, following those of SyncR. Any violation of these prerequisites results in the function returning $\mathsf{Undefined}$. Additionally, we stipulate that $\mathsf{path}(\mathsf{Undefined}) := \mathsf{Undefined}$.

- The first argument, $\vec{\phi}$, must be of the form $\Big[[\delta_1], [\delta_2], \ldots, [\delta_n]\Big]$ for some n.
- For any stack symbol γ, the pre-path $\vec{\phi} = f(\gamma)$ must be *cohesive* with the length matching the number of rules with the up_γ operation in $\vec{\phi}$.

The validity of the initial call, i.e., $\mathsf{Sync}(\mathsf{path}(\vec{\mu}_s), \mathsf{path} \circ f_{\vec{\mu}})$, is derived directly from the properties of $\vec{\mu}_s$ and $\vec{\mu}_\gamma$ for all γ, as previously established when defining SyncR.

We then detail the definition $\mathsf{Sync}(\vec{\phi}, f)$, again by a case analysis on the structure of $\vec{\phi}$. In the base case, $\mathsf{Sync}(\varepsilon, -)$ yields the empty sequence ε. For the inductive case, by the first prerequisite, it takes the form $\mathsf{Sync}([\delta] :: \vec{\phi}'_s, f)$:

- If $op^{(\delta)}$ represents a *down* operation, the outcome is $[\delta] :: \mathsf{Sync}(\vec{\phi}'_s, f)$.
- If $op^{(\delta)}$ signifies an *up* operation towards direction γ', let $f(\gamma')$ be $\phi' :: \vec{\phi}'$ (it is valid given the second prerequisite), then the outcome of the recursive call $\mathsf{Sync}(\vec{\phi}'_s, f[\gamma' \mapsto \vec{\phi}'])$ determines the subsequent step.
  - If the outcome is $\phi_r :: \vec{\phi}_r$, then the final result is $((\delta :: \phi') +\!\!+ \phi_r) :: \vec{\phi}_r$;
  - otherwise, (when the outcome is ε) the final result is $[\delta :: \phi']$.

The validity of the recursive calls above can be straightforwardly verified. So does Equation (4.18), where a straightforward induction on the length of $\vec{\mu}_s$ suffices.

Essentially, this method inductively decomposes any potential pre-path in a *by-level* manner, which can be visualised by the following example.

Example 4.19. See Figure 4.4. The figure depicts a pre-path with two paths embedded within. Each horizontal line, distinguished by a unique color, denotes a node on the tree-stack. Each arrow represents a rule to execute and the direction of traversal upon applying this rule. The arrows, matching the colour of their respective lines, illustrate the transitions occurring at the node (denoted as $\vec{\phi}_s$ above), known as the *synchronisation path* to be introduced below. The node line along with the synchronisation paths intuitively forms a shape like a "fish bone" (by observing each colour individually, especially the blue node).

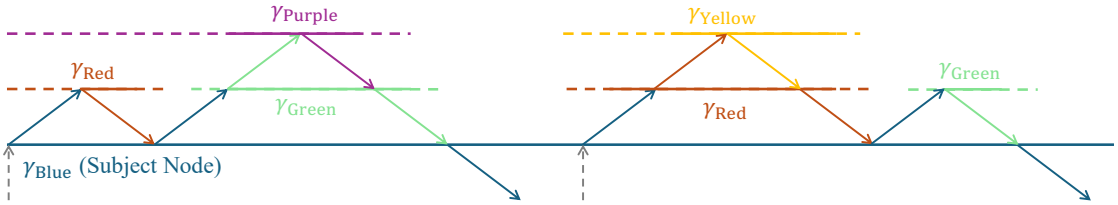

Figure 4.4: Visual Example of By-Level Decomposition

This pre-path delineates the following sequence: Initially, an *up* transition occurs from the blue node to the red node, subsequently followed by a *down* transition back to the blue node. Next, the blue node performs an *up* transition to the green node, which then follows with an *up* transition further to the purple node. From the purple node, there is a series of *down* transitions back to the parent of the blue node, marking the end of the first path in the pre-path.

Then, in the second path, the blue subject node transitions again to the red node via an *up*, where the red node interacts once with a new yellow node and then back to the blue one. Following this, the second path ends with an interaction with the green node again.

To be more formal, following the aforementioned decomposition principle, we are essentially decomposing this pre-run "by color" as follows:

$$\mathsf{Sync}\left(\vec{\phi}_s^{(\text{Blue})}, \left\{\begin{array}{l} \gamma_{\text{Red}} \mapsto \mathsf{Sync}(\vec{\phi}_s^{(\text{Red})}, \{\gamma_{\text{Yellow}} \mapsto \vec{\phi}^{(\text{Yellow})}\}) \\ \gamma_{\text{Green}} \mapsto \mathsf{Sync}(\vec{\phi}_s^{(\text{Green})}, \{\gamma_{\text{Purple}} \mapsto \vec{\phi}^{(\text{Purple})}\}) \end{array}\right\}\right)$$

Note that the function Sync will reorder the pre-paths to the correct sequence, thereby correctly recovering the original pre-path as illustrated in the graph.

Decomposing Patterns of Pre-Paths

It is not hard to recognise the following fact:

Lemma 4.20. *For every $\gamma \in \Gamma$ and every path $\vec{\mu}$ in some pattern, let $\vec{\mu}_\gamma := \mathsf{AtDir}(\vec{\mu}, \gamma)$, then $\mathsf{path}(\vec{\mu}_\gamma)$ belongs to $\mathsf{path}(\mathit{TripsOrUp}^{\gamma}_{\mathsf{Interface}(\vec{\mu}_\gamma)})$.*

This lemma stems from the straightforward observation that transitions confined to or above a specific node depend solely on the tree structure at or above that node for enabling paths. A detailed proof is provided later.

Thus, any pre-path within $\mathsf{path}(\mathit{TripsOrUp}^{\gamma_0}_D)$ can be dissected into the internal pre-path (e.g., $\mathsf{path}(\vec{\mu}_s)$) and the pre-paths for each direction within the corresponding pre-paths pattern, identified by the interface sequence (e.g., for each $\gamma \in \Gamma$, the $\mathsf{path}(\mathit{TripsOrUp}^{\gamma}_{\mathsf{Interface}(\vec{\mu}_\gamma)})$).

Expanding on this concept, the principle of decomposing a pattern of pre-paths emerges. It is to say, the pattern can be partitioned based on all possible combinations of *interface sequences* (e.g., $\mathsf{Interface}(\vec{\mu}_\gamma)$ for each $\gamma \in \Gamma$), where each combination equates to a non-empty set if there exists a pre-path that "synchronises" among these interfaces, e.g., $\mathsf{path}(\vec{\mu}_s)$.

We formalise this concept into the decomposition equation, starting with defining the "combination of interface sequences" through *upward interactions assigning functions*.

Given a function $\mathbb{D} : \Gamma \to \mathcal{Q}^*$, it is said to *match* a *Trips* pattern if all values of $\mathbb{D}$ are even in length and no greater than $2k$, with k being the restriction number of the rPTSA $\mathcal{A}$. Alternatively, it matches an *Up* pattern if (1) all lengths of $\mathbb{D}$ values are $\leq 2k$, and (2) exactly one $\gamma \in \Gamma$ has an odd length in $\mathbb{D}$, while all other $\gamma' \in \Gamma \setminus \{\gamma\}$ have even lengths. Such a function that matches either pattern type, is termed an *interaction assigning function*, where that $\mathbb{D}$ matches a pattern v is denoted $\mathbb{D} \sim : v$.

The decomposition then unfolds as follows, for $v := \mathit{TripsOrUp}^{\gamma_0}_D$:

$$\mathsf{path}(v) = \biguplus_{\mathbb{D} \sim : v} \left\{ \mathsf{Sync}\Big(\vec{\phi}_s, \{\gamma \mapsto \vec{\phi}_\gamma \mid \gamma \in \Gamma\}\Big) \;\middle|\; \begin{array}{c} \vec{\phi}_s \in \mathit{SynPaths}^{\mathbb{D}}_{(\gamma_0, D)} \\ \forall \gamma \in \Gamma . \vec{\phi}_\gamma \in \mathsf{path}(V^\gamma_{\mathbb{D}}) \end{array} \right\} \tag{4.21}$$

where $V^\gamma_{\mathbb{D}} := \mathit{TripsOrUp}^{\gamma}_{\mathbb{D}(\gamma)}$, and $\mathit{SynPaths}^{\mathbb{D}}_{(\gamma_0, D)}$ signifies the set of *synchronisation paths* capable of bridging the upward interactions by $\mathbb{D}$ with the downward interaction by D at γ_0, e.g., $\mathsf{path}(\vec{\mu}_s)$.

The notion of synchronisation paths plays a pivotal role throughout this study. Hence, in the following, we first present a detailed formal introduction, followed by a proof of the decomposition equation (Equation (4.21)).

Example 4.22. Consider a concrete example drawing from Example 4.7. A path in $\mathsf{path}(\mathit{Trips}^{\gamma_{AC}}_{[\boldsymbol{q}_A, \boldsymbol{q}_{back}, \boldsymbol{q}_C, \boldsymbol{q}_{back}]})$, in the form of: $[\delta_1^n \delta_2 \delta_3^n, \delta_9^n \delta_{10} \delta_3^n]$ can be decomposed as: $\mathsf{Sync}(\vec{\mu}_s, \{\gamma_{AC} \mapsto \vec{\mu}_{AC}, \gamma_{BD} \mapsto \varepsilon\})$ where, either $\vec{\mu}_s$ is $[[\delta_1], [\delta_3], [\delta_9], [\delta_3]]$ with $\vec{\mu}_{AC}$ being $[\delta_1^{n-1} \delta_2 \delta_3^{n-1}, \delta_9^{n-1} \delta_{10} \delta_3^{n-1}]$ or $\vec{\mu}_s := [[\delta_2], [\delta_{10}]]$ with $\vec{\mu}_{AC}$ being ε.

Observing this, we have the decomposition equation as follows:

$$\begin{aligned} &\mathsf{path}(\mathit{Trips}^{\gamma_{AC}}_{[\boldsymbol{q}_A, \boldsymbol{q}_{back}, \boldsymbol{q}_C, \boldsymbol{q}_{back}]}) \\ &= \{\mathsf{Sync}([[\delta_2], [\delta_{10}]], \{\gamma_{AC} \mapsto \varepsilon, \gamma_{BD} \mapsto \varepsilon\})\} \\ &\uplus \{\mathsf{Sync}(\vec{\mu}_s, \{\gamma_{AC} \mapsto \vec{\mu}_{AC}, \gamma_{BD} \mapsto \varepsilon\}) \mid \vec{\mu}_{AC} \in \mathsf{path}(\mathit{Trips}^{\gamma_{AC}}_{[\boldsymbol{q}_A, \boldsymbol{q}_{back}, \boldsymbol{q}_C, \boldsymbol{q}_{back}]})\} \end{aligned}$$

where $\vec{\mu}_s := [[\delta_1], [\delta_3], [\delta_9], [\delta_3]]$, and, in the first case, $\mathbb{D}$ is simply $\lambda - .\varepsilon$, while in the second case, $\mathbb{D}$ assigns γ_{AC} the value of $[\boldsymbol{q}_A, \boldsymbol{q}_{back}, \boldsymbol{q}_C, \boldsymbol{q}_{back}]$ and γ_{BD} as ε.

4.2.3 Synchronisation Paths

Re-Formalising Synchronisation Paths The concept of synchronisation paths is of key importance not only in the analysis of termination but also in the general analysis of various properties in subsequent chapters. At the same time, their representation has been quite limited:

$$\Big[[\delta_1], [\delta_2], \ldots, [\delta_n]\Big] \tag{4.23}$$

To highlight this concept and address this unique structure, we introduce a distinct formalism to represent synchronisation paths. The term *synchronisation path* will henceforth exclusively denote this specific variant of a pre-path, as aligned

with the formalism presented herein. This new formalism is designed to facilitate more streamlined discussions, offering a refined method to describe the characteristics of the pre-path, as illustrated in Equation (4.23). When examining the properties or structure of synchronisation paths, this formalism will be used to promote a more focused and precise discussion.

Importantly, these synchronisation paths remain essentially a specialised kind of pre-paths and are *fully compatible* with conventional pre-path applications.

Reaction Unit To define the formalism of synchronisation paths, we first present its fundamental components, that *reaction unit*s.

A reaction unit ζ is a tuple (q, q', m, κ), where:

- $q \in \mathcal{Q}$ is the *starting (control) state* or *source (control) state* of the unit;
- $q' \in \mathcal{Q}$ is the *reacting (control) state* of the unit;
- $m \in \mathcal{M}$ is the *reacting (local) memory* of the unit; and
- $\kappa \in \mathcal{K} = \{\Downarrow\} \uplus \Gamma$ is the *reacting direction* of the unit.

Synchronisation Path Then, a synchronisation path is a *finite* sequence of reaction units additionally equipped with the information of the starting local memory and the stack status. Specifically, a *synchronisation path* π is of the form $[\zeta_1, \ldots, \zeta_n]_{(m_0, g)}$, where each reaction unit $\zeta_i = (q_i, q'_i, m_i, \kappa)$ at position i within it signifies δ_i in Equation (4.23):

$$(q_i, m_{i-1}, g) \xrightarrow{p} op(\zeta_i)$$

where the *underlying operation* represented by the reaction unit $op(\zeta_i)$ is given by:

$$op(\zeta_i) := \begin{cases} (q'_i, m_i, down) & \text{if } \kappa = \Downarrow \\ (q'_i, m_i, up_\kappa) & \text{if } \kappa \in \Gamma \end{cases}$$

Note that m_0 is the subscript mark of the whole path. The whole rule is then referred to as the rule *represented* by the reaction unit at i in π, denoted ${\delta_i}^{(\pi)}$, in the spirit of Equation (4.23).

Note that the definition does not require the whole synchronisation path to be *complete* – especially, it might not start from the creation of the node (implying $m_0 = m_{init}$) and its end does not intuitively imply the completion of the overall interactions within the subject node. Further, we may directly use $\delta \in \pi$ to extract the underlying rules of a given synchronisation path π in order.

Filtration We may obtain the interaction sequence of given direction κ by filtering a given synchronisation path π. This, denoted by $\pi \restriction_{\kappa_0} \kappa$, is defined by:

$$\begin{aligned} \pi \restriction_{\kappa_0} \kappa := & \left[\; q_1 \;\middle|\; \kappa = \kappa_0 \; \right] \\ & +\!\!+ \; \mathsf{concat}\left(\left[\; [q'_i, q_{i+1}] \;\middle|\; i \in \{1, \ldots, n-1\}, \kappa_i = \kappa \; \right] \right) \\ & +\!\!+ \left[\; q'_n \;\middle|\; \kappa_n = \kappa \; \right] \end{aligned}$$

where κ_0 is called the *source direction* of the path. We may abbreviate $\restriction_{\Downarrow}$ to just $\restriction$. Besides, we specifically define that $\varepsilon \restriction_{\kappa_0} \kappa = \varepsilon$ for all κ_0 and κ.

Attributes of Synchronisation Path The *weight* for a given synchronisation path π, denoted by $wt(\pi)$, is determined by multiplying the weights of all the underlying rules each unit of reaction represents.

The *mean hitting time* for π, denoted by $mht(\pi)$, is calculated as the product of its weight and length.

The *total accumulated reward* for π with respect to a *simple* reward function f, denoted by $\mathsf{rwd}_f(\pi)$, is the sum of the rewards for all rules involved. Recall that, a *simple* reward function implies that the reward value is directly determined by the transition rule.

Empty Synchronisation Path When no reaction unit is present, meaning no transition is performed at the current node, we have an empty synchronisation path of the form $[]_{(m,\mathscr{g})}$. The key distinction from the representation in Equation (4.23) is that this form should additionally convey the local memory and stack status. This adjustment is more intuitive, as such information must persist independently of existence or extinction of transitions. However, when this additional information is irrelevant, we technically denote it by simply ε, maintaining compatibility with the original representation.

Set of Synchronisation Paths The set of synchronisation paths for the rPTSA $\mathcal{A}$ under analysis is represented by *SynPaths*. The notation $SynPaths^{\mathbb{D}}_{(\mathscr{g},D)}$ specifies the set of synchronisation paths that:

1. occur within the given stack status $\mathscr{g} \in \Gamma \uplus \{\bot\}$,
2. are aligned with the specified downward interaction sequence D, and
3. conform to the given upward interaction assigning function $\mathbb{D}$,

formally defined as:

$$SynPaths^{\mathbb{D}}_{(\mathscr{g},D)} := \left\{ \pi \in SynPaths \;\middle|\; \begin{array}{c} \pi = [\dots]_{(m_{init},\mathscr{g})} \\ \pi \restriction \Downarrow = D \\ \forall \gamma \in \Gamma . \pi \restriction \gamma = \mathbb{D}(\gamma) \end{array} \right\} \tag{4.24}$$

Especially, if $D = \varepsilon$, $\mathbb{D}(\gamma) = \varepsilon$ for all γ, the set contains simply ε.

Given the k-restriction, it is evident that the length of any synchronisation path does not exceed $k \cdot (|\Gamma| + 1)$, as exceeding this would imply a $(k+1)$th visit to a node, leading to a contradiction of the k-restriction. This observation underscores the finiteness (and hence, the decidability) of *SynPaths* and its subsets.

Example 4.25. Consider the Example 4.1 and its pattern $Trips^{\gamma}_{[\mathscr{q}_f,\mathscr{q}_d,\mathscr{q}_s,\mathscr{q}_d]}$. Then, let $D = [\mathscr{q}_f, \mathscr{q}_d, \mathscr{q}_s, \mathscr{q}_d]$, we may inductively decompose the pre-paths pattern $\mathsf{path}(Trips^{\gamma}_{D})$ by the following cases on the upward interactions $\mathbb{D}$:

- There is no upward interaction, such that $\mathbb{D} = \lambda - .\varepsilon$. Then, in the set $\mathit{SynPaths}^{\mathbb{D}}_{(\gamma,D)}$, there is only one possible synchronisation path π_T denoting reaching the # point:

$$[(q_f, q_d, m_T, \Downarrow), (q_s, q_d, m_R, \Downarrow)]_{(m_{init},\gamma)}$$

- The interactions with $\mathbb{D}(\gamma) = [q_f, q_d, q_s, q_d]$. In this case, there are two possible synchronisation paths in $\mathit{SynPaths}^{\mathbb{D}}_{(\gamma,D)}$:

  The path for generating firstly an A-pair in the first and second w, respectively, named π_A:

$$[(q_f, q_f, m_A, \gamma), (q_d, q_d, m_A, \Downarrow), (q_s, q_s, m_R, \gamma), (q_d, q_d, m_R, \Downarrow)]_{(m_{init},\gamma)}$$

  The path then for the B-pairs, termed π_B:

$$[(q_f, q_f, m_B, \gamma), (q_d, q_d, m_B, \Downarrow), (q_s, q_s, m_R, \gamma), (q_d, q_d, m_R, \Downarrow)]_{(m_{init},\gamma)}$$

It is easily verifiable that these two cases already cover all meaningful cases with all the other corresponding pre-paths sets being empty.

Hence, we have the following formula for decomposing $\mathsf{path}(\mathit{Trips}^{\gamma}_{[q_f,q_d,q_s,q_d]})$:

$$\begin{aligned}\mathsf{path}(\mathit{Trips}^{\gamma}_{[q_f,q_d,q_s,q_d]}) = {} & \{\mathsf{Sync}(\pi_T, \lambda - .\varepsilon)\} \\ & \uplus \left\{ \mathsf{Sync}(\pi, \{\gamma \mapsto \vec{\phi}\}) \;\middle|\; \begin{array}{c} \pi \in \{\pi_A, \pi_B\} \\ \vec{\phi} \in \mathsf{path}(\mathit{Trips}^{\gamma}_{[q_f,q_d,q_s,q_d]}) \end{array} \right\} \end{aligned} \tag{4.26}$$

4.2.4 Proof of the Decomposition Equation

The decomposition equation (e.g., Equation (4.21)) can be established by demonstrating both its soundness (LHS $\subseteq$ RHS) and completeness (LHS $\supseteq$ RHS), encapsulated in the following formal lemmas.

Lemma 4.27 (Soundness of the Decomposition)**.** *For any $v := \mathit{TripsOrUp}^{\gamma_0}_D$, it holds that:*

$$\mathsf{path}(v) \subseteq \biguplus_{\mathbb{D} \sim: v} \left\{ \mathsf{Sync}\Big(\pi_s, \Big\{\gamma \mapsto \vec{\phi}_\gamma \;\Big|\; \gamma \in \Gamma\Big\}\Big) \;\middle|\; \begin{array}{c} \pi_s \in \mathit{SynPaths}^{\mathbb{D}}_{(\gamma_0,D)} \\ \forall \gamma \in \Gamma. \vec{\phi}_\gamma \in \mathsf{path}(\mathit{TripsOrUp}^{\gamma}_{\mathbb{D}(\gamma)}) \end{array} \right\}$$

Lemma 4.28 (Completeness of the Decomposition)**.** *For any $v := \mathit{TripsOrUp}^{\boxed{\gamma_0}}_D$, it holds that:*

$$\mathsf{path}(v) \supseteq \biguplus_{\mathbb{D} \sim: v} \left\{ \mathsf{Sync}\Big(\pi_s, \Big\{\gamma \mapsto \vec{\phi}_\gamma \;\Big|\; \gamma \in \Gamma\Big\}\Big) \;\middle|\; \begin{array}{c} \pi_s \in \mathit{SynPaths}^{\mathbb{D}}_{(\gamma_0,D)} \\ \forall \gamma \in \Gamma. \vec{\phi}_\gamma \in \mathsf{path}(\mathit{TripsOrUp}^{\gamma}_{\mathbb{D}(\gamma)}) \end{array} \right\}$$

Remark 4.29 (Traceable Symbols)**.** To mitigate complexity and prevent confusion from similar symbols, we will employ colour-coded links for key symbols. Certain symbols will be boxed and assigned a specific colour in its definition, such as $\boxed{x}$, with each instance appearing in the same colour and hyperlinked to its definition, e.g., x, to ensure ease of reference and clarity.

Notably, in the above completeness lemma, we have specially marked γ_0 for use in the proof of completeness to avoid ambiguity.

Proof of Soundness

The development of the decomposition equation underpins the proof of soundness, focusing on demonstrating Lemma 4.20.

To establish the lemma, we first introduce several key concepts and prove some auxiliary lemmas. We define the *suprapointer sub-tree* of a configuration $(-, t, \rho)$ as the sub-tree of t rooted at ρ. Two configurations are said to be *suprapointer bisimular* if they share the same suprapointer sub-tree and control state.

Additionally, we introduce the concept of a *relative tree route*, which facilitates the comparison between two paths ρ and ρ' in the tree. The relative path from ρ to ρ' is expressed as a pair $\mathcal{y} = (d, \rho_u)$, intuitively describing "how to walk from ρ to ρ'": d is a natural number representing the number of *down* operations required, and ρ_u denotes the path along the lower branch. Formally, let ρ_b be the *longest* common prefix of ρ and ρ'. Then, $d = |\rho| - |\rho_b|$, and ρ_u is such that $\rho_b +\!\!+ \rho_u = \rho'$. For a given configuration c, we define may directly refer to relative path $\mathcal{y}$ as the tree-stack route from the stack poitner of c with $\mathcal{y}$.

For a given non-terminating run $c \xrightarrow{\phi} c'$, the *relative stack-pointer movement* refers to the relative tree route from the stack pointer of c to that of c'. The same concept applies to paths, where the relative path is also unique.

Lemma 4.30. *For a given path ϕ, its relative stack-pointer movement is unique.*

Proof. We prove this by showing that for any two runs $c_1 \xrightarrow{\phi} c_1'$ and $c_2 \xrightarrow{\phi} c_2'$, the relative stack-pointer movements are identical. This result follows directly from a straightforward induction on the length of ϕ. □

We now prove the following lemma concerning infinite paths.

Lemma 4.31. *An infinite path ϕ is enabled by a configuration c if and only if c enables every finite prefix of ϕ.*

Proof. The forward direction is straightforward. For the reverse direction, we can construct an infinite run μ starting from c such that the $(i+1)$th configuration in the run, for each $i \in \{1, 2, \dots\}$, is exactly the unique configuration c_i where $c \xrightarrow{\phi[..i]} c_i$ holds.

Observe that for every consecutive pair of configurations, c_n and c_{n+1} in μ, the transition $c_n \xrightarrow{\phi[n]} c_{n+1}$ must hold. If not, the finite prefix $\phi[..n]$ would not be enabled by c, contradicting the assumption that every finite prefix of ϕ is enabled by c. □

We now prove the following key lemma, which will be used extensively in subsequent sections. Intuitively, it asserts that "transplanting" the suprapointer sub-tree has no impact on the behaviors of trip and soar paths.

Lemma 4.32. *Any two suprapointer bisimilar configurations share the same set of trip paths and soar paths.*

Specifically, given any finite trip or soar path ϕ enabled by two such configurations $c = (-, t, \rho)$ and $c' = (-, t', \rho')$, let $c_e = (-, t_e, \rho_e)$ and $c'_e = (-, t'_e, \rho'_e)$ be the resulting configurations obtained by executing ϕ from c and c', respectively. Then:

1. *the sub-tree of t_e rooted at ρ is identical to the sub-tree of t'_e rooted at ρ'; and,*

2. *the entire tree-stack t is equivalent to t_e except for the sub-tree rooted at ρ, similarly, t' is equivalent to t'_e except for the sub-tree rooted at ρ';*

3. *the relative tree-stack route from ρ to ρ_e is identical to that from ρ' to ρ'_e.*

Proof. Since the configurations c and c' are arbitrary, we show that any trip or soar path enabled by c is also enabled by c', and that any finite trip or soar path satisfies the conditions stated above.

The proof for finite paths proceeds by straightforward induction on the length of any finite soar path ϕ enabled by c, which is trivial, with the third condition following directly from Lemma 4.30. For trip paths, observe that any trip path consists of a soar path followed by a rule involving the *down* operation. Since the condition holds for soar paths, it holds trivially for trip paths as well.

By the fact that any finite soar path is enabled by c', it follows from Lemma 4.31 that any infinite soar path enabled by c is also enabled by c'. □

This forms the basis for the key lemma on cohesive pre-paths.

Lemma 4.33. *Any two suprapointer bisimilar configurations enable the same set of cohesive pre-paths.*

Proof. This follows straightforwardly from the definitions of cohesive pre-run and cohesive pre-path, by Lemma 4.32. □

Then, we are ready to prove provide the soundness proof.

Proof (Lemma 4.20). Consider $\vec{\mu}_\gamma = [c_1 \ldots, c_2 \ldots, \ldots, c_n \ldots]$. We demonstrate that a pre-run $\vec{\mu}' \in \mathit{TripsOrUp}^{\gamma}_{\mathsf{Interface}(\vec{\mu}_\gamma)}$ exists such that $\mathsf{path}(\vec{\mu}_\gamma) = \mathsf{path}(\vec{\mu}')$. Construct $\vec{\mu}'$ as $[c'_1 \ldots, \ldots, c'_n \ldots]$, where each $c'_i := (q'_i, t'_i, [\gamma])$ corresponds to $c_i = (q_i, t_i, \rho_i)$ with ρ_i being $\gamma_0\gamma$, $q'_i = q_i$, and the sub-tree of the tree-stack t' from $[\gamma]$ is the same as that of t from $[\gamma_0, \gamma]$. The root of the tree consistently remains m_{init} in pattern construction.

It remains to show that each c'_i enables the path underlying the run $c_i \ldots$ in $\vec{\mu}_\gamma$. This follows directly from Lemma 4.32, since c'_i and c_i share the same suprapointer subtree. □

The soundness is then established by:

Proof (Lemma 4.27). For any $\vec{\phi} \in \mathsf{path}(v)$, a corresponding $\vec{\mu} \in v$ exists such that $\vec{\phi} = \mathsf{path}(\vec{\mu})$. Applying Equation (4.17) allows for the decomposition of $\vec{\phi}$. By Lemma 4.20, every pre-path $\mathsf{path}(\vec{\mu}_\gamma)$ is included in $\mathsf{path}(\mathit{TripsOrUp}^{\gamma}_{\mathsf{Interface}(\vec{\mu}_\gamma)})$ for all $\gamma \in \Gamma$, with $\vec{\mu}_\gamma := \mathsf{AtDir}(\vec{\mu}, \gamma)$.

Constructing $\mathbb{D}$ such that $\mathbb{D}(\gamma) = \mathsf{Interface}(\vec{\mu}_\gamma)$ for every $\gamma \in \Gamma$, it remains to demonstrate that $\mathsf{path}(\mathsf{CurNode}(\vec{\mu}))$ falls within $\mathit{SynPaths}^{\mathbb{D}}_{(\gamma_0, D)}$, which is evident. □

Proof of Completeness

Next, we address the completeness aspect, which is more complicated than the previous case. To begin with, we present some more auxiliary concepts and supportive lemmas.

Auxiliary Structures Two tree-stacks t_1 and t_2 are said to be *bisimular* (or, bisimulating each other) if the differences in their domains all correspond to the initial memory m_{init}. Formally, for every $\rho \in \mathrm{dom}(t_1) \setminus \mathrm{dom}(t_2)$, we have $t_1(\rho) = m_{init}$, and for every $\rho \in \mathrm{dom}(t_2) \setminus \mathrm{dom}(t_1)$, we have $t_2(\rho) = m_{init}$. Two configurations c_1 and c_2 are said to be *tree-stack bisimular* if their tree-stacks are bisimilar and they share the same control state and stack pointer, or they are both termination configurations. By definition, it is evident that two tree-stack bisimilar configurations share the same set of enabled rules, enabled paths, and enabled cohesive pre-paths. Moreover, after the execution of any enabled path in both configurations, the resulting configurations remain tree-stack bisimilar.

Completeness Proof After presenting the above auxiliary structures and the supportive lemmas stating their properties, we are then able to turn our attention to the proof of completeness. We first present an important lemma to be used in the proof.

Lemma 4.34. *Given a synchronisation path π and a function f assigning for each upward direction $\gamma \in \Gamma$ a* cohesive *pre-path, if $\pi \restriction \gamma = \mathsf{Interface}(f(\gamma))$ for all γ, then there is at most one γ_0 such that $f(\gamma_0)$ ends with a soar path, in which case, π ends with a reaction unit with reacting direction being exactly γ_0.*

Proof. It suffices to notice the following straightforward facts regarding the parity from the definitions that:

- If π ends with reacting direction *down*, then $|\pi \restriction \gamma|$ must be of even length for all γ.
- If π ends with a reacting direction γ_0, then $|\pi \restriction \gamma_0$ must be of odd length while for all other $\gamma \in \Gamma \setminus \{\gamma_0\}$, $|\pi \restriction \gamma|$ must still be of even length.
- The interface of cohesive pre-path is of even length iff it contains only trips path, and of odd length iff it ends with a soar path.

□

Then, we are ready to present the following central lemma closely connected to the completeness proof.

Lemma 4.35. *Given a synchronisation path π for a stack status $\gamma \in \Gamma$, and a function f that maps stack symbols to* cohesive *pre-paths, for which the function* Sync *is defined, the resulting sequence $\vec{\mu} = \mathsf{Sync}(\pi, f)$ must form a cohesive pre-path that is enabled by any configuration $c = (-, t, \rho)$ satisfying the following conditions:*

- *let $\pi = [\dots]_{(m,\gamma)}$, then ρ ends with γ and $t(\rho) = m$;*

- *for each $\gamma \in \Gamma$, there exists a configuration $\mathscr{c}_\gamma$ enabling $f(\gamma)$ such that the sub-tree at $\rho\gamma$ is equivalent to the suprapointer sub-tree of $\mathscr{c}_\gamma$.*

Proof. In this proof, our aim is to construct a cohesive pre-run $\vec{\mu}$ from any configuration $\mathscr{c}$ that satisfies the conditions above, ensuring $\mathsf{path}(\vec{\mu}) = \mathsf{Sync}(\pi, f)$. For clarity, we define $f_0 = f$, $\pi_0 = \pi$, and $\mathscr{c}_0 = \mathscr{c}$, freeing these symbols for further use. Additionally, let the stack pointer of $\mathscr{c}_0$ be denoted as ρ_0.

For this construction, we define an auxiliary function SyncM that returns the desired cohesive pre-run. Formally, our objective is to define a function SyncM that satisfies the following condition:

$$\mathsf{Sync}(\pi, f) = \mathsf{path}(\mathsf{SyncM}(\mathscr{c}, \pi, f)) \tag{4.36}$$

where Sync is based on π and f. Then, we construct the pre-run $\vec{\mu}$ as $\mathsf{SyncM}(\mathscr{c}_0, \pi_0, f_0)$.

Before presenting the definition, let us first specify the complete "contract" for the input and output of $\mathsf{SyncM}(\mathscr{c}, \pi, f)$, which the function is promised to satisfy and must uphold during recursive calls, analogous to the concept of refinement types [191, 192].

1. If π is non-empty, the starting rule ${\delta_1}^{(\pi)}$ must be enabled by $\mathscr{c}$.
2. The configuration $\mathscr{c}$ must have the same stack pointer as $\mathscr{c}_0$, i.e., ρ_0.
3. For each direction $\gamma \in \Gamma$, $\pi \restriction \gamma = \mathsf{Interface}(f(\gamma))$.
4. For every $\gamma \in \Gamma$, $f(\gamma)$ must either be empty or a cohesive pre-path enabled by the configuration $(\mathscr{q}, \mathscr{t}^{(\mathscr{c})}, \rho_0\gamma)$, where $\mathscr{q}$ is the first control state of the initial sequence in $f(\gamma)$. Further, we will refer to this configuration as the *phantom configuration* for γ.
5. The final result must satisfy Equation (4.36).
6. The result must either be a cohesive pre-run starting from $\mathscr{c}$ or an empty sequence.

It is important to note that the input conditions all hold at the initial call $\mathsf{SyncM}(\mathscr{c}_0, \pi_0, f_0)$ based on the assumptions.

We are now ready to define the function SyncM.

In the base case, for $\mathsf{SyncM}(-, \varepsilon, -)$, the result is simply the empty sequence ε. This coincides with $\mathsf{Sync}(\varepsilon, -)$, serving as the base case for $\mathsf{Sync}(-, -)$.

For the inductive case, we consider $\mathsf{SyncM}(\mathscr{c}, \boxed{\pi}, f)$. Here, let

$$\pi = [\boxed{\zeta_1}, \zeta_2, \ldots, \zeta_n]_{(\mathscr{m}_0, \gamma)}$$

and $\boxed{\mathscr{c}'}$ be the configuration such that $\mathscr{c} \xrightarrow{{\delta_1}^{(\pi)}} \boxed{\mathscr{c}'}$. Let $\boxed{\mathscr{m}_1}$ the reacting memory of ζ_1, and define $\boxed{\pi'} := [\zeta_2, \ldots, \zeta_n]_{(\mathscr{m}_1, \gamma)}$. We then proceed with a case analysis based on the operation type $\mathit{op}(\zeta_1)$:

- When $op(\zeta_1)$ is a *down* operation, the outcome is $[\mathscr{c}, \mathscr{c}'] :: \mathsf{SyncM}(\mathscr{c}'', \pi', f)$. Here, $\mathscr{c}''$ is defined as $(\mathscr{q}_2, t^{(\mathscr{c}')}, \rho_0)$, where:
  - If $n \geq 2$, $\mathscr{q}_2$ is the starting state of ζ_2.
  - Otherwise, $\mathscr{q}_2 = \mathscr{q}_{init}$, serving as a placeholder.

  Notably, when $n \geq 2$, the recursive call involves a non-empty synchronisation path, ensuring that ${\delta_1}^{(\pi'')}$ is enabled by $\mathscr{c}''$. The second condition also holds by the definition of $\mathscr{c}''$. The third condition remains unchanged, inheriting its validity from $\mathscr{c}$. For the fourth condition, since $\mathscr{c}$ and $\mathscr{c}''$ share the same sub-tree from $\rho_0\gamma$ for each γ, the required enablement is guaranteed due to the suprapointer bisimilarity between the phantom configuration for γ constructed from $\mathscr{c}''$ and the original one from $\mathscr{c}$, as established by Lemma 4.33.

  Regarding the fifth condition, which relates to Equation (4.36), the guarantee ensures that: $\mathsf{Sync}(\pi', \mathsf{path} \circ f) = \mathsf{path}(\mathsf{SyncM}(\mathscr{c}'', \pi', f))$, implying that the result matches $\mathsf{path}([\mathscr{c}, \mathscr{c}'])$, which corresponds to $[{\delta_1}^{(\pi)}]$.

  Finally, for the sixth condition concerning the construction of a cohesive pre-run starting from $\mathscr{c}$, notice that the pair $[\mathscr{c}, \mathscr{c}']$ already forms a trip. Additionally, by the guarantee, $\mathsf{SyncM}(\mathscr{c}'', \pi', f)$ produces a cohesive pre-run $\vec{\mu}'$ starting from $\mathscr{c}''$, prepending the trip $[\mathscr{c}, \mathscr{c}']$ to $\vec{\mu}'$ results in a cohesive pre-run.

- When $op(\zeta_1)$ is an *up* operation to the direction γ_1: According to the third condition, $f(\gamma_1)$ must be non-empty and can be denoted as $\phi_1 :: \vec{\phi}_1$. Additionally, $\mathscr{c}'$ must have the control state being the first source state of ϕ_1, making $\mathscr{c}'$ suprapointer bisimilar to the configuration described in the fourth condition, thus enabling the entire sequence $\phi_1 :: \vec{\phi}_1$ by Lemma 4.32.

  Let μ_1' denote the run starting from $\mathscr{c}$, extending from $\mathscr{c}'$ through ϕ_1. The result then depends on the nature of ϕ_1.

  - If ϕ_1 is a soar path, this implies that ζ_1 is the final reaction unit in π by Lemma 4.34, marking the conclusion of the process, and the result is $[\mu_1']$.

    In this case, for the fifth condition, the underlying pre-path matches $[{\delta_1}^{(\pi)} :: \phi_1]$, which is equivalent to $\mathsf{Sync}([\zeta_1]_{(m_0,\gamma)}, f)$.

    Moreover, since ϕ_1 is a soar path, μ_1' must also be a soar from $\mathscr{c}$, meaning $[\mu_1']$ is a valid cohesive pre-run starting from $\mathscr{c}$, satisfying the sixth condition.
  - If ϕ_1 is a trip path, let $\mathscr{c}''$ be the configuration such that $\mathscr{c} \xrightarrow{\phi_1} \mathscr{c}''$, and define f' as $f[\gamma_1 \mapsto \vec{\mu}_1]$.

    The subsequent behaviour depends on the return of $\mathsf{SyncM}(\mathscr{c}'', \pi', f')$:

    * If it returns ε, the result is $[\mu_1']$.

* Otherwise, by the sixth condition, it must return a structure of the form $\vec{\mu}'' = (\mathscr{c}'' :: \mu_r) :: \vec{\mu}_r$. In this case, the result becomes $(\mu_1' +\!\!+ \mu_r) :: \vec{\mu}_r$.

Next, we verify the satisfaction of the conditions:

1. First, by Lemma 4.30, since the relative pointer movement of a trip path must be $(1, \varepsilon)$, the stack pointer of $\mathscr{c}''$ must be ρ_0. Let the tree-stack of $\mathscr{c}''$ be $\mathscr{t}''$ and that of $\mathscr{c}'$ be $\mathscr{t}'$. Therefore, $\mathscr{t}''(\rho_0) = \mathscr{t}'(\rho_0) = \mathscr{m}_1$, as the transitions only affect $\rho_0\gamma_1$ and above, formally guaranteed by Lemma 4.32. Furthermore, the control state of $\mathscr{c}''$ must be the final reacting state of ϕ_1, which, by the third condition, matches the initial state of ζ_2 (if it exists). If π' is non-empty, $\mathscr{c}''$ must enable ${\delta_1}^{(\pi')}$.
2. The second condition follows directly from the previous discussion.
3. The third condition holds because both $\pi' \restriction \gamma_1$ and $\mathsf{Interface}(f(\gamma_1))$ are derived by removing the first two states from $\pi \restriction \gamma_1$ and $\mathsf{Interface}(f(\gamma_1))$, respectively, while for all $\gamma \neq \gamma_1$, no changes occur.
4. For the fourth condition, $f'(\gamma_1)$ becomes $\vec{\phi}_1$, which must be cohesive since it is a postfix of a cohesive pre-path. Moreover, the phantom configuration constructed from $\mathscr{c}''$ with f' for γ_1 must enable $\vec{\phi}_1 = f'(\gamma_1)$, by the properties of tree-stack bisimilarity.
   For each $\gamma \neq \gamma_1$, the sub-tree of $\mathscr{c}''$ at $\rho_0\gamma$ remain unchanged from $\mathscr{c}'$ by Lemma 4.32, which are also equivalent to those of $\mathscr{c}$. Hence, the phantom configuration $(\mathscr{q}, \mathscr{t}'', \rho_0\gamma)$ is suprapointer bisimilar to the original one $(\mathscr{q}, \mathscr{t}^{(c)}, \rho_0\gamma)$, enabling $f(\gamma)$ if it is non-empty by Lemma 4.33.
5. Regarding Equation (4.36), it follows directly from the definition of Sync and the guarantee from the recursive call.
6. As for the cohesiveness of the result, we are prepending a non-descending run wrt. ρ_0 from $\mathscr{c}$ to the first run of the cohesive pre-run $\vec{\mu}''$, which, by definition, forms a cohesive pre-run starting from $\mathscr{c}$.

By the output guarantees provided, it is straightforward to obtain that the initial call, $\mathsf{SyncM}(\mathscr{c}_0, \pi_0, f_0)$ must be a cohesive pre-run starting from $\mathscr{c}_0$ whose underlying pre-path is exactly $\mathsf{Sync}(\pi_0, f_0)$, concluding the proof. □

The completeness proof unfolds as follows:

Proof (Lemma 4.28). Fixing a specific $\mathbb{D}$, we select $\pi \in \mathit{SynPaths}^{\mathbb{D}}_{(\gamma_0, D)}$. For each $\gamma \in \Gamma$, we choose $\vec{\mu}_\gamma \in \mathit{TripsOrUp}^{\gamma}_{\mathbb{D}(\gamma)}$. Let f be defined by $f(\gamma) = \mathsf{path}(\vec{\mu}_\gamma)$. Our goal is to show that $\mathsf{Sync}(\pi, f)$ belongs to $\mathsf{path}(\upsilon)$.

We now apply the results of the key lemma above. By Lemma 4.35, the cohesive pre-path $\vec{\phi} = \mathsf{Sync}(\pi, f)$ must be enabled by a configuration $\mathscr{c} = (\mathscr{q}, \mathscr{t}, [\gamma_0])$, where $\mathscr{q} = \mathsf{hd}(D)$, and $\mathscr{t}$ has a domain containing precisely ε, $[\gamma_0]$, and $[\gamma_0, \gamma]$ for all γ, with each mapped to $\mathscr{m}_{\mathit{init}}$. This holds because $f(\gamma)$ must be enabled by a configuration whose suprapointer sub-tree is $\{\varepsilon \mapsto \mathscr{m}_{\mathit{init}}\}$, according to the definition of patterns.

Observe that the configuration $\mathscr{c}$ bisimulates the fixed pattern configuration $\mathscr{c}' = (\mathsf{hd}(D), \{\varepsilon \mapsto m_{init}, [\gamma_0] \mapsto m_{init}\}, [\gamma_0])$. By the property of tree-stack bisimilarity, $\vec{\phi}$ must also be enabled by $\mathscr{c}'$. Finally, by the definition of patterns, the unique cohesive pre-run hosting $\vec{\phi}$ from $\mathscr{c}'$ must lie in v. Thus, $\vec{\phi} \in \mathsf{path}(v)$.

□

Proving Equation (4.21) Finally, the target proof is presented as follows.

Proof (Equation (4.21)*).* The result follows directly from both the soundness declared in Lemma 4.27 and the completeness as declared in Lemma 4.28. □

4.2.5 The Decomposition Equation System

As discussed in the methodology section, following the inductive decomposition, our next step involves addressing the equation system generated from the inductive relations of pattern attributes, which are typically concerned with specific target quantities like weights or the mean hitting time. Here, we first introduce the equation system for the attribute of the pre-paths set, which serves as a generalisation of such quantities. Subsequently, we will explore the characterisation of *Trips* patterns via the derived equation system, which is crucial for the forthcoming concrete termination analysis.

The Equation System

With the k-restriction, we can safely consider a finite set of patterns where the length of the interaction sequence is $\leq 2k$, thus also rendering the set of decomposition equations for these patterns finite (e.g., Equation (4.21)). To form an equation system, a patterns-indexed vector of distinct variables $\vec{x}$ is introduced, and each pre-path pattern $\mathsf{path}(\mathit{TripsOrUp}^{\gamma}_{D})$ is substituted by the corresponding variable $\vec{x}[\mathit{TripsOrUp}^{\gamma}_{D}]$ throughout all the decomposition equations for all patterns. This transformation results in a new system of equations expressed in terms of the variables $\vec{x}$, where each variable $\vec{x}[\mathit{TripsOrUp}^{\gamma}_{D}]$ uniquely represents a specific pre-path pattern identified by its dimension. By this substitution, we retain the original relationships and abstract the system for further analysis.

More specifically, the equation system could be expressed in terms of a function $\mathcal{F}_{\mathsf{path}}$ that maps a patterns-indexed vector of pre-paths sets to another such vector, defined as follows. For every pattern $\mathit{TripsOrUp}^{\gamma}_{D}$, $\mathcal{F}_{\mathsf{path}}(\vec{x})[\mathit{TripsOrUp}^{\gamma}_{D}]$ is the RHS of Equation (4.21) for this pattern with the symbols $\mathsf{path}(\mathit{TripsOrUp}^{\gamma'}_{\mathbb{D}(\gamma')})$ replaced by the corresponding variable $\vec{x}[\mathit{TripsOrUp}^{\gamma'}_{\mathbb{D}(\gamma')}]$. That is, for all patterns $v := \mathit{TripsOrUp}^{\gamma}_{D}$:

$$
\begin{aligned}
&\mathcal{F}_{\mathsf{path}}(\vec{x})[v] \\
:= &\bigcup_{\mathbb{D} \sim: v} \left\{ \mathsf{Sync}\Big(\pi_s, \Big\{\gamma' \mapsto \vec{\phi}_{\gamma'} \,\Big|\, \gamma' \in \Gamma\Big\}\Big) \,\middle|\, \begin{matrix} \pi_s \in \mathit{SynPaths}^{\mathbb{D}}_{(\gamma,D)} \\ \forall \gamma' \in \Gamma . \vec{\phi}_{\gamma'} \in \vec{x}[\mathit{TripsOrUp}^{\gamma'}_{\mathbb{D}(\gamma')}] \end{matrix} \right\}
\end{aligned}
$$

Note that since the input set is arbitrary, there is no guarantee of the result being disjoint for different $\mathbb{D}$. Also, given that Sync requires specific prerequisites for its inputs, failure to meet these prerequisites may result in the value Undefined. Notably, for every Sync call in $\mathcal{F}_{\mathsf{path}}$, the first prerequisite is inherently satisfied by the synchronisation paths found. A violation may occur if the argument of $x_{\mathit{TripsOrUp}^{\gamma}_{D}}$ includes some pre-paths $\vec{\phi}$ whose length does not match $\lfloor(|D|+1)/2\rfloor$. Thus, more precisely, $\mathcal{F}_{\mathsf{path}}$ produces a patterns-indexed vector of either sets of pre-paths or the value Undefined.

Through this process, we derive an equation system of the form $\vec{x} = \mathcal{F}_{\mathsf{path}}(\vec{x})$. Let the vector of all pre-paths patterns be $\vec{\Psi}_{\mathsf{path}(\mathit{TripsOrUp})}$, i.e.,

$$\vec{\Psi}_{\mathsf{path}(\mathit{TripsOrUp})}[\mathit{TripsOrUp}^{\gamma}_{D}] := \mathsf{path}(\mathit{TripsOrUp}^{\gamma}_{D})$$

then, evidently, it is a solution to the equation system and thus also a *fixed point* of the function $\mathcal{F}_{\mathsf{path}}$. Similarly, we define $\vec{\Psi}_{\mathsf{path}(\mathit{Trips})}$ as the vector of all pre-paths *Trips* patterns.

Characterising *Trips* Values

The *Trips* patterns, particularly their corresponding pre-path patterns, are essential for the termination analysis. Here, we further present a characterisation of the *Trips* pre-paths patterns' values via the obtained equation system.

Focusing on the *Trips* patterns, it is observed that in the RHS of every decomposition equation (Equation (4.21)), the *Up* patterns are never involved. Thus, we denote $\mathcal{F}_{\mathsf{path}(\mathit{Trips})}$ as the component corresponding solely to the *Trips* patterns, with the input value also constrained to the vector of *Trips* patterns.

The characterisation follows from the intuitive observation that the RHS of the decomposition equation (Equation (4.21)) for a *Trips* pattern essentially *gathers information from one more level above*. The target value of *Trips* patterns can then be obtained by infinitely gathering information from above – by infinitely applying $\mathcal{F}_{\mathsf{path}(\mathit{Trips})}$ to the vector of empty sets as the initial condition ($\vec{\emptyset}$), which assumes no information about this level.

This concept can be formally captured by introducing the notion of *sub-Trips-patterns*.

An h-height sub-*Trips*-pattern, denoted by $\langle \mathit{Trips}^{\gamma}_{D}\rangle^{\uparrow h}$, is essentially a subset of the given pattern $\mathit{Trips}^{\gamma}_{D}$, with the maximum length of stack pointers in the configurations limited to k. Naturally, $\langle \mathit{Trips}^{\gamma}_{D}\rangle^{\uparrow 0} = \emptyset$, and $\mathit{Trips}^{\gamma}_{D} = \lim_{h\to\infty} \langle \mathit{Trips}^{\gamma}_{D}\rangle^{\uparrow h}$.

A sub-pattern can then be decomposed via the following equation:

$$\mathsf{path}\Big(\langle \mathit{Trips}^{\gamma_0}_{D}\rangle^{\uparrow(h+1)}\Big) = \biguplus_{\mathbb{D}\sim:\, v} \left\{ \mathsf{Sync}\Big(\pi, \{\gamma \mapsto \vec{\phi}_{\gamma} \mid \gamma \in \Gamma\}\Big) \;\middle|\; \begin{array}{c} \pi \in \mathit{SynPaths}^{\mathbb{D}}_{(\gamma_0, D)} \\ \forall \gamma \in \Gamma. \vec{\phi}_{\gamma} \in \mathsf{path}(V^{\gamma}_{\mathbb{D}}) \end{array} \right\}$$

where $v = \mathit{Trips}^{\gamma_0}_{D}$, $V^{\gamma}_{\mathbb{D}} := \left\langle \mathit{Trips}^{\gamma}_{\mathbb{D}(\gamma)} \right\rangle^{\uparrow h}$.

The proof mirrors that for the original decomposition equation for pre-path patterns (Equation (4.21)), with the additional consideration that every run,

excluding the current level, should be one level less than the input (i.e., from $h+1$ to h) when transitioning to the level above.

Letting the vector of h-height sub-*Trips*-patterns be $\vec{V}_h$, i.e., for every pattern $v = \mathit{Trips}^{\gamma}_{D}$, $\vec{V}_h[v] := \mathsf{path}(\langle v \rangle^{\uparrow h})$, and we can express the above equation in terms of $\mathcal{F}_{\mathsf{path}(\mathit{Trips})}$ as:

$$\vec{V}_{h+1}[v] = \mathcal{F}_{\mathsf{path}(\mathit{Trips})}(\vec{V}_h)[v]$$

Given that v is an arbitrary *Trips* pattern, generalising this to the vector of h-height sub-*Trips*-patterns, we get:

$$\vec{V}_{h+1} = \mathcal{F}_{\mathsf{path}(\mathit{Trips})}(\vec{V}_h)$$

This leads to the following characterisation of $\vec{V}_h$:

$$\vec{V}_h = \mathcal{F}^{h}_{\mathsf{path}(\mathit{Trips})}(\vec{V}_0)$$

where $\mathcal{F}^{h}_{\mathsf{path}(\mathit{Trips})}$ denotes the function $\mathcal{F}_{\mathsf{path}(\mathit{Trips})}$ applied h times.

Finally, we arrive at the characterisation of the *Trips* patterns' values:

$$\vec{\Psi}_{\mathsf{path}(\mathit{Trips})} = \lim_{h\to\infty} \vec{V}_h = \lim_{h\to\infty} \mathcal{F}^{h}_{\mathsf{path}(\mathit{Trips})}(\vec{V}_0) = \lim_{h\to\infty} \mathcal{F}^{h}_{\mathsf{path}(\mathit{Trips})}(\vec{\emptyset})$$

4.3 Analysis of Termination Probability

We now turn to the problem of solving the termination probability.

For clarity and without loss of generality, we impose the following constraint on $\mathcal{A}$ for the remainder of this chapter: only two types of rules are permitted at the stack bottom — $(q_{init}, m_{init}, \bot) \to (q_{init}, m_{end}, up_{\gamma_\bot})$ and $(q, m_{end}, \bot) \to (q, m_{end}, \mathit{down})$. Here, $\gamma_\bot \in \Gamma$ is a special stack symbol, $m_{end} \neq m_{init}$ is a special local memory, and $q \in Q$ ranges over any control states.[3]

This restriction allows us to directly link the target value to the patterns. For instance, the termination probability could be equated to the sum of $wt(\mathit{Trips}^{\gamma_\bot}_{[q_{init},q]})$ across all q.

Any general rPTSA can be modified to fit this form: reassign rules involving $\bot$ to a newly introduced $\gamma_\bot$, using only the specified rules for $\bot$. With a specially designed reward function (which disregards the two types of transitions for $\bot$ mentioned above), these effects can be ignored when calculating the expected total accumulated reward. This constraint simplifies our analysis significantly.

To facilitate exposition, for the Example 4.1, we also modify it to accommodate the assumption. Henceforth, $\mathcal{A}_{COPY}$ will denote the modified example below.

[3]For completeness, there are also rules for the status $(q, m, \bot)$ where m is other than m_{init} or m_{end}. These are assumed to divert to the dummy divergence direction γ_Ω as they are never utilised.

Example 4.37 (Modified $\mathcal{A}_{COPY}$)**.** We adapt the automaton $\mathcal{A}_{COPY}$ from Example 4.1, which terminates upon the successful generation of a string in the COPY language $\{w\#w \mid w \in (A|B)^*\}$, to align with the assumption. Specifically, the rules in Figure 4.1 are modified such that all $\bot$ symbols are replaced with $\gamma_\bot$, and two types of bottom rules are added accordingly. Henceforth, throughout the remainder of this chapter, any reference to $\mathcal{A}_{COPY}$ pertains to this modified version.

Then, recall the three types of problems delineated for termination analysis; our focus in this section is on the *quantitative problem*. Specifically, we will demonstrate the following theorem.

Theorem 4.38 (Quantitative Analysis of Termination Probability)**.** *Given any non-negative constant c, and any comparator $\sim \in \{\leq, <, =, \neq, >, \geq\}$, the problem of whether $tp(\mathcal{A}) \sim c$ is in EXPSPACE.*

The qualitative problem follows as a direct corollary.

Theorem 4.39 (Almost Sure Termination)**.** *The problem of whether $\mathcal{A}$ terminates almost surely, that is $tp(\mathcal{A}) = 1$, is in EXPSPACE.*

To address the problem, we here adopt a more adapted version of the general methodology mentioned above. First, we establish a polynomial equation system from which the value of $tp(\mathcal{A})$ can be delineated as a polynomial combination of a specific solution to the system – thanks to the imposed restriction, it is the sum of the weights of the *Trips* patterns, as mentioned. Next, based on the equation system, we resolve the quantitative problem by encoding it into an inquiry expressed in the first-order arithmetic theory of the reals using only existential quantification, which is decidable and in PSPACE, as per Theorem 2.8.

4.3.1 Equation System of Weights

We then proceed to work on the mentioned equation, which fundamentally constitutes the equation system of the weights of the patterns.

The Decomposition Equation

Subsequently, the equation system of weights can be directly derived from the decomposition of pre-paths, i.e., Equation (4.21), by applying the function $wt(-)$ to both sides. For a given pattern $v := \mathit{Trips}^{\gamma_0}_D$:

$$wt(v) = wt(\mathsf{path}(v)) = wt\Big(\biguplus_{\mathbb{D} \sim : v} S_{\mathbb{D}}\Big)$$

where $S_{\mathbb{D}}$ is defined locally in this section to reduce the use of complex notation.

$$S_{\mathbb{D}} := \left\{ \mathsf{Sync}\Big(\pi_s, \{\gamma \mapsto \vec{\phi}_\gamma \mid \gamma \in \Gamma\}\Big) \;\middle|\; \begin{array}{c} \pi_s \in \mathit{SynPaths}^{\mathbb{D}}_{(\gamma_0, D)} \\ \forall \gamma \in \Gamma . \vec{\phi}_\gamma \in \mathsf{path}(\mathit{Trips}^{\gamma}_{\mathbb{D}(\gamma)}) \end{array} \right\}$$

We shall then examine the RHS. As the set is *partitioned* disjointly by the various $\mathbb{D}$, we have:

$$wt\Big(\biguplus_{\mathbb{D}\sim:v} S_{\mathbb{D}}\Big) = \sum_{\mathbb{D}\sim:v} wt(S_{\mathbb{D}})$$

Next, we investigate $wt(S_{\mathbb{D}})$ for a specified $\mathbb{D}$.

Each combination of $\pi \in \mathit{SynPaths}^{\mathbb{D}}_{(\gamma_0,D)}$ and $\vec{\phi}_\gamma \in \mathsf{path}(\mathit{Trips}^{\gamma}_{\mathbb{D}(\gamma)})$ for every $\gamma \in \Gamma$ yields a distinct $\mathsf{Sync}(\pi, \{\gamma \mapsto \vec{\phi}_\gamma \mid \gamma \in \Gamma\})$. Therefore, $wt(S_{\mathbb{D}})$ is the sum of $\mathsf{Sync}(\pi, \{\gamma \mapsto \vec{\phi}_\gamma \mid \gamma \in \Gamma\})$ for all possible combinations of π and $\vec{\phi}_\gamma$ for every γ. Denoting Γ as $\{\gamma_1, \ldots, \gamma_n\}$, we have:

$$wt(S_{\mathbb{D}}) = \sum_{\pi\in \mathit{SynPaths}^{\mathbb{D}}_{(\gamma_0,D)}} \sum_{\vec{\phi}_{\gamma_1}\in\mathsf{path}(\mathit{Trips}^{\gamma_1}_{\mathbb{D}(\gamma_1)})} \cdots \sum_{\vec{\phi}_{\gamma_n}\in\mathsf{path}(\mathit{Trips}^{\gamma_n}_{\mathbb{D}(\gamma_n)})} X$$

with X being:

$$wt(\mathsf{Sync}(\pi, \{\gamma_1 \mapsto \vec{\phi}_{\gamma_1}, \ldots, \gamma_n \mapsto \vec{\phi}_{\gamma_n}\}))$$

Further noting that Sync simply concatenates the rules in a specific order from both its parameters without erasing or duplicating any rules [4], we have:

$$wt(\mathsf{Sync}(\pi, f)) = wt(\pi) \cdot \prod_{\gamma\in\Gamma} wt(f(\gamma)) \tag{4.40}$$

This observation can be more rigorously proved through simple induction on the length of π.

By the principle that:

$$\sum_{x_1\in X_1} \cdots \sum_{x_m\in X_m} f_1(x_1)\cdots f_m(x_m) = \Big(\sum_{x_1\in X_1} f_1(x_1)\Big)\cdots\Big(\sum_{x_m\in X_m} f_m(x_m)\Big)$$

We deduce that:

$$\begin{aligned} wt(S_{\mathbb{D}}) &= wt(\mathit{SynPaths}^{\mathbb{D}}_{(\gamma_0,D)}) \cdot \prod_{\gamma\in\Gamma} wt(\mathsf{path}(\mathit{Trips}^{\gamma}_{\mathbb{D}(\gamma)})) \\ &= wt(\mathit{SynPaths}^{\mathbb{D}}_{(\gamma_0,D)}) \cdot \prod_{\gamma\in\Gamma} wt(\mathit{Trips}^{\gamma}_{\mathbb{D}(\gamma)}) \end{aligned}$$

Recalling that the first equality is valid as the sum of weights across all pre-paths in the set is just the weight of the set itself.

In summary, we arrive at the following decomposition equation for the weights of patterns.

$$wt(\mathit{Trips}^{\gamma}_{D}) = \sum_{\mathbb{D}\sim:\,\mathit{Trips}^{\gamma}_{D}} wt(\mathit{SynPaths}^{\mathbb{D}}_{(\gamma,D)}) \cdot \prod_{\gamma'\in\Gamma} wt(\mathit{Trips}^{\gamma'}_{\mathbb{D}(\gamma')}) \tag{4.41}$$

[4] As the input under discussion trivially satisfies the prerequisites, the result shall never be Undefined.

The Equation System of Weights of Patterns

Like for the decomposition of pre-paths patterns, given a *Trips*-patterns-indexed vector of variables $\vec{x}$, by substituting all the occurrences of $wt(\mathit{Trips}^{\gamma}_{D})$ with the corresponding variable $\vec{x}[\mathit{Trips}^{\gamma}_{D}]$ in the RHS of the decomposition equation for all *Trips* patterns, and indexing them by the corresponding pattern, we derive a function $\mathcal{F}_{wt(\mathit{Trips})}$ and an equation system $\vec{x} = \mathcal{F}_{wt(\mathit{Trips})}(\vec{x})$ where the variable vector $\vec{x}$ is indexed by patterns of non-negative real numbers. For each pattern $\mathit{Trips}^{\gamma}_{D}$, its corresponding dimension in $\mathcal{F}_{wt(\mathit{Trips})}$ is defined as:

$$\mathcal{F}_{wt(\mathit{Trips})}(\vec{x})[\mathit{Trips}^{\gamma}_{D}] := \sum_{\mathbb{D} \sim: \mathit{Trips}^{\gamma}_{D}} wt(\mathit{SynPaths}^{\mathbb{D}}_{(\gamma,D)}) \cdot \prod_{\gamma' \in \Gamma} \vec{x}[\mathit{Trips}^{\gamma'}_{\mathbb{D}(\gamma')}]$$

Letting the value $\vec{\Psi}_{wt(\mathit{Trips})}$ be defined as $\vec{\Psi}_{wt(\mathit{Trips})}[\mathit{Trips}^{\gamma}_{D}] := wt(\mathit{Trips}^{\gamma}_{D})$ for all patterns $\mathit{Trips}^{\gamma}_{D}$, it clearly constitutes a solution to the aforementioned equation system and also a fixed point of $\mathcal{F}_{wt(\mathit{Trips})}$.

To alleviate the symbolic burden, henceforth in this section, we shall default the symbol $\mathcal{F}$ to $\mathcal{F}_{wt(\mathit{Trips})}$ unless explicitly specified otherwise. Similarly, we shall default $\vec{\Psi}$ to $\vec{\Psi}_{wt(\mathit{Trips})}$.

Remark 4.42 (Derivation of Equation (4.41))**.** During the step-by-step derivation from Equation (4.21) to Equation (4.41), two critical properties of the pre-path patterns enabled this progression, which are worth highlighting:

1. The set of pre-paths is disjointly partitioned by differing $\mathbb{D}$.
2. For a given $\mathbb{D}$, each combination of $\pi \in \mathit{SynPaths}^{\mathbb{D}}_{(\gamma_0,D)}$ and for every $\gamma \in \Gamma$ the $\vec{\phi}_{\gamma} \in \mathsf{path}(\mathit{Trips}^{\gamma}_{\mathbb{D}(\gamma)})$ yields a distinct $\mathsf{Sync}(\pi, \{\gamma \mapsto \vec{\phi}_{\gamma} \mid \gamma \in \Gamma\})$ outcome.

4.3.2 Analysis via the Equation System

Finally, we are one step away from deriving the termination probability, and the remaining task is to analyse the generated system of equations. However, the first hurdle, as might be somehow unexpected, is the problem of determining "which fixed point we want".

Identifying the Target Fixed Point More specifically, merely recognising $\vec{\Psi}$ as a fixed point of $\mathcal{F}$ is insufficient; we must further ascertain which fixed point it is to analyse the value appropriately. To this effect, we present the following theorem:

Theorem 4.43. *$\vec{\Psi}$ is the Least Fixed Point (LFP) of $\mathcal{F}$.*

Proof. We revisit the utilisation of sub-patterns according to their heights, denoted $\langle \mathit{Trips}^{\gamma}_{D} \rangle^{\uparrow h}$ introduced in Section 4.2.5. Also, define the vector of h-height sub-*Trips*-patterns as $\vec{V}_h$, that is, for each $\mathit{Trips}^{\gamma}_{D}$, $\vec{V}_h := \mathsf{path}(\langle \mathit{Trips}^{\gamma}_{D} \rangle^{\uparrow h})$. Following the derivation of Equation (4.41), it can be readily verified that the following holds for all h [5]:

$$wt(\mathcal{F}_{\mathsf{path}(\mathit{Trips})}(\vec{V}_h)) = \mathcal{F}(wt(\vec{V}_h))$$

[5] As both aforementioned properties in Remark 4.42 are satisfied.

Consequently, it is straightforward to deduce that:

$$\vec{\Psi} = \lim_{h\to\infty} wt(\vec{V_h}) = \lim_{h\to\infty} wt(\mathcal{F}^h_{\mathsf{path}(\mathit{Trips})}(\vec{V_0})) = \lim_{h\to\infty} \mathcal{F}^h(wt(\vec{V_0})) = \lim_{h\to\infty} \mathcal{F}^h(\vec{0}) \tag{4.44}$$

Given that $\mathcal{F}$ is both monotone and continuous, the LFP of $\mathcal{F}$ is thus $\bigsqcup_{h=0}^{\infty} \mathcal{F}^h(\vec{0})$, which involves taking the least upper bound of the sequence $\mathcal{F}^h(\vec{0})$ from when h is 0 to infinity. In our case, it precisely equates to $\lim_{h\to\infty} \mathcal{F}^h(\vec{0})$, aligning with $\vec{\Psi}$. □

Analysing Termination Probability We analyse the termination probability of $\mathcal{A}$, which, under the assumptions concerning $\mathcal{A}$, can be expressed as $\sum_{q\in\mathcal{Q}} wt(\mathit{Trips}^{\gamma_\perp}_{[q_{init},q]})$.

Consequently, we can prove Theorem 4.38.

Proof (Theorem 4.38). We can encode the fixed points of $\mathcal{F}$ succinctly as follows:

$$\exists \vec{x}.\vec{x} \geq \vec{0} \wedge \vec{x} = \mathcal{F}(\vec{x})$$

However, this does not solely capture the (non-negative) LFP, our primary focus. Therefore, in general, to accurately identify the LFP of $\mathcal{F}$, it may be necessary to also employ $\forall$ quantifiers, extending beyond the scope of the existential fragment.

Nonetheless, for the specific case of the quantitative problem, it is feasible to resolve it using only EToR. Given a (non-negative) constant c, and if the operator $\sim$ belongs to $\{<, \leq\}$, we consider the following existential formula:

$$\exists \vec{x}.\vec{x} \geq \vec{0} \wedge \vec{x} = \mathcal{F}(\vec{x}) \wedge \sum_{q\in\mathcal{Q}} \vec{x}[\mathit{Trips}^{\gamma_\perp}_{[q_{\text{init}},q]}] \sim c$$

If the EToR procedure affirms, it indicates the existence of at least one non-negative fixed point of $\mathcal{F}(\vec{x})$ with the target sum less than or equal to c, thus resolving the quantitative question.

Conversely, when $\sim$ is either $>$ or $\geq$, the situation is more complex. For the $>$ comparison, we examine the following existential formula, adjusting our approach:

$$\exists \vec{x}.\vec{x} \geq \vec{0} \wedge \vec{x} = \mathcal{F}(\vec{x}) \wedge \sum_{q\in\mathcal{Q}} \vec{x}[\mathit{Trips}^{\gamma_\perp}_{[q_{\text{init}},q]}] \;\leq\; c$$

By inquiring $\leq$ instead of $>$, if EToR responds with NO, it implies the absence of any fixed point with the target sum $\leq c$, signifying that the target sum over the LFP must exceed c, hence effectively addressing the inquiry for $>$. A similar approach applies for $\geq$, where we inquire about $<$ in place of $\leq$.

Lastly, let us examine the complexity of the aforementioned procedure. Initially, the number of distinct *Trips* patterns to explore is at most $|\Gamma| \cdot \sum_{i=0}^{k} |Q|^{2i} \leq |Q|^{2k+2} \cdot |\Gamma|$. For each *Trips* pattern, the number of distinct summands (kinds of different $\mathbb{D}$) in the RHS of Equation (4.41) is up to $|Q|^{(2k+2)|\Gamma|}$. Each summand involves $|\Gamma|$ variables and calculates $wt(\mathit{SynPaths}^{\mathbb{D}}_{(\gamma,D)})$, entailing the following complexity.

Each path in $\mathit{SynPaths}^{\mathbb{D}}_{(\gamma,D)}$ has at most $|D|/2$ (up to k) *down* operations and at most $|\mathbb{D}(\gamma)|/2$ (up to k) up_γ operations for each $\gamma \in \Gamma$, representing all possible

interleavings with the last operation being a *down*. This results in at most $\frac{(k-1+k|\Gamma|)!}{(k-1)!\cdot(k!)^{|\Gamma|}}$ variations. Given $k(|\Gamma|+1)$ transitions per path, with each offering up to $|\mathcal{M}|$ choices for the next (reacting) local memory, the scale of $\mathit{SynPaths}^{\mathbb{D}}_{(\gamma,D)}$ is at most $\frac{(k-1+k|\Gamma|)!}{(k-1)!\cdot(k!)^{|\Gamma|}} \cdot |\mathcal{M}|^{k(|\Gamma|+1)}$.

Summing over all *Trips* patterns, the overall computational complexity becomes:

$$\begin{aligned}
&|\Gamma| \cdot |Q|^{2k+2} \cdot |Q|^{(2k+2)|\Gamma|} \cdot \left(\frac{(k-1+k|\Gamma|)!}{(k-1)! \cdot (k!)^{|\Gamma|}} \cdot |\mathcal{M}|^{k(|\Gamma|+1)} \right) \\
\leq\ &|\Gamma| \cdot |Q|^{(2k+2)(|\Gamma|+1)} \cdot \frac{(k+k|\Gamma|)!}{k! \cdot (k!)^{|\Gamma|}} \cdot |\mathcal{M}|^{k(|\Gamma|+1)} \\
=\ &|\Gamma| \cdot |Q|^{(2k+2)(|\Gamma|+1)} \cdot \frac{(k(|\Gamma|+1))!}{(k!)^{|\Gamma|+1}} \cdot |\mathcal{M}|^{k(|\Gamma|+1)}
\end{aligned}$$

To simplify the factorial term, we apply Stirling's approximation [193]:

$$n! \approx n^n e^{-n} \sqrt{2\pi n}$$

Applying this to each factorial:

$$\frac{(k(|\Gamma|+1))!}{(k!)^{|\Gamma|+1}} \approx \frac{N^N e^{-N}}{(k^k e^{-k})^{|\Gamma|+1}} \cdot \frac{\sqrt{2\pi N}}{\left(\sqrt{2\pi k}\right)^{|\Gamma|+1}}$$

where $N = k(|\Gamma|+1)$.

We focus on the dominant exponential terms for complexity approximation, ignoring the minor factors on the right:

$$\frac{N^N e^{-N}}{(k^k e^{-k})^{|\Gamma|+1}} = \frac{N^N}{k^{k(|\Gamma|+1)}} = \frac{(k(|\Gamma|+1))^{k(|\Gamma|+1)}}{k^{k(|\Gamma|+1)}} = (|\Gamma|+1)^{k(|\Gamma|+1)}$$

Incorporating this back into the overall complexity, we obtain an approximation of the following form:

$$|\Gamma| \cdot |Q|^{(2k+2)(|\Gamma|+1)} \cdot (|\Gamma|+1)^{k(|\Gamma|+1)} \cdot |\mathcal{M}|^{k(|\Gamma|+1)}$$

Expressing exponential terms using natural logarithms, and neglecting the left-most term, which contributes to the exponent only logarithmically, we derive:

$$\exp\left((\ln |Q|^{2k+2} + \ln |\mathcal{M}|^k + \ln(|\Gamma|+1)^k)(|\Gamma|+1) \right)$$

Simplifying the exponents, we obtain:

$$\exp\left(\left((2k+2)\ln|Q| + k\ln|\mathcal{M}| + k\ln(|\Gamma|+1)\right)(|\Gamma|+1) \right)$$

For simplicity, let us introduce the rPTSA size $|\mathcal{A}|$ as the sum $|\mathcal{A}| = |\mathcal{Q}| + |\mathcal{M}| + |\Gamma|$. Thus, we may safely claim that the overall complexity of constructing the equation system is bounded by $O(\exp(\mathrm{poly}(|\mathcal{A}|+k)))$.

Next, we consider the final application of EToR [194], which is PSPACE-complete with respect to the size of the logical formula involved. The size of this formula, primarily derived from the equation system, scales linearly with the number of *Trips* patterns, which is bounded by $|\Gamma| \cdot |\mathcal{Q}|^{2k+2}$. Since $|\mathcal{Q}|^{2k+2} = \exp((2k+2) \cdot \ln|\mathcal{Q}|)$, it grows exponentially with respect to k and $\ln|\mathcal{Q}|$. Consequently, the formula size — and thus the required space — remains bounded by an exponential function in $\text{poly}(|\mathcal{A}| + k)$, placing it firmly within the EXPSPACE complexity class.

In conclusion, due to the exponential growth in both the construction size and the logical formula, the problem belongs to EXPSPACE. □

The Approximation Problem Finally, we address the approximation problem, which is commonly a concern in practice.

The equation in the proof of Theorem 4.43(Equation (4.44)) instantly provides a method for primitive Kleene iteration. By iterating $\mathcal{F}^n(\vec{0})$ a sufficient number of times, n, one can achieve a reasonable approximation of the LFP value. This method is generally effective in practice.

However, this approach does not guarantee precision in convergence, as demonstrated in some work [86, 109], it is possible for convergence to a specified accuracy to take exponential time. Under this situation, some studies have introduced iteration through Newton's method, with further research indicating a positive outcome: for a system of equations that is strongly connected – a property that is easily attained – iteration via Newton's method achieves *linear* convergence. That is, the number of iterations needed increases linearly in terms of the bits of accuracy required.

Another approach involves employing bisection by iteratively using the quantitative decision procedure (i.e., Theorem 4.38). In this case, for any specified accuracy, the number of times the decision procedure is invoked remains constant when employing bisection.

Example 4.45. Consider the Example 4.1. We are now ready to analyse its termination probability, hence the probability of randomly generating a string of the form $w\#w$ with w an arbitrary string in $(A|B)^*$.

Observe that the relevant patterns for this examples for termination analysis are: $\mathit{Trips}^{\gamma_\perp}_{[q_f,q_d]}$ and $\mathit{Trips}^{\gamma}_{[q_f,q_d,q_s,q_d]}$. The case for the former is trivial. And by by applying $wt(-)$ on both sides on Equation (4.26), we may derive the following equation system for this example:

$$\begin{aligned} wt(\mathit{Trips}^{\gamma_\perp}_{[q_f,q_d]}) &= wt(\mathit{Trips}^{\gamma}_{[q_f,q_d,q_s,q_d]}) \\ wt(\mathit{Trips}^{\gamma}_{[q_f,q_d,q_s,q_d]}) &= p_T + (p_A^2 + p_B^2) \cdot wt(\mathit{Trips}^{\gamma}_{[q_f,q_d,q_s,q_d]}) \end{aligned}$$

As established in the above proof, for this decomposition equation system, we first substitute $wt(\mathit{Trips}^{\gamma_\perp}_{[q_f,q_d]})$ and $wt(\mathit{Trips}^{\gamma}_{[q_f,q_d,q_s,q_d]})$ with variables. Subsequently, we determine the least non-negative solution for the system with respect to these variables, serving as the correct values for the weights, as established. This solution also yields the result for the termination probability:

$$tp(\mathcal{A}_{COPY}) = wt(\mathit{Trips}^{\gamma_\perp}_{[q_f,q_d]}) = wt(\mathit{Trips}^{\gamma}_{[q_f,q_d,q_s,q_d]}) = \frac{p_T}{1-(p_A^2+p_B^2)}$$

Although the result has been verified by our method, one may further grasp the intuition behind the solution as follows: Consider two cases – first, for the empty string w, the probability of printing just # is p_T; second, for a non-empty string w, the probability of matching characters before and after # is p_A^2 for A and p_B^2 for B, thus the total probability of generating a corresponding matching pair is $p_A^2 + p_B^2$.

For any length of w, this forms a geometric sequence – For w of length 1, the probability is $p_T(p_A^2 + p_B^2)$; while for length 2, it is $p_T(p_A^2 + p_B^2)^2$; with $p_T(p_A^2 + p_B^2)^3$ for length 3; and so on.

Then, by standard technique, the sum of this sequence is:

$$\frac{p_T(p_A^2 + p_B^2)}{1 - (p_A^2 + p_B^2)}$$

Combining both cases, the total probability is:

$$\frac{p_T}{1 - (p_A^2 + p_B^2)}$$

This is exactly the result produced by our analysis procedure.

4.4 Analysis of Expected Termination Time

Subsequently, we are prepared to address the enquiries related to the expected termination time. However, instead of directly tackling this, we shift our focus to the more general attribute of the *expected total accumulated reward*.

More specifically, our attention is centred on the following theorem:

Theorem 4.46. *For any* simple *reward function f, the quantitative query regarding whether $etar_f(\mathcal{A}) \sim c$ for a specified constant c and a comparator $\sim$ is in EXPSPACE.*

As a corollary, we derive the outcome for the expected termination time:

Theorem 4.47. *The quantitative problem concerning the expected termination time of $\mathcal{A}$ is in EXPSPACE.*

Moreover, in conjunction with the decidable AST, we ascertain that:

Theorem 4.48 (Positive AST)**.** *The positive almost surely terminating problem of $\mathcal{A}$ is in EXPSPACE.*

In the ensuing discussion, we shall initially generate an equation system, this time for the expected total accumulated reward, and thereafter undertake analysis with the equation system. We hereby designate a *simple* reward function f for use in the remainder of this section.

4.4.1 The Equation System

As always, we hereby derive the decomposition equation system for the expected total accumulated rewards by first establishing the inductive decomposition equation, followed by the complete equation system consisting of the equations.

The Decomposition Equation

Adhering to the methodology in the analysis of the termination probability, we initially deduce the decomposition of the equation for the expected total accumulated reward by applying $etar_f$ to both sides of the decomposition of the pre-paths patterns. Recalling Equation (4.14), for a specified pattern $v := \mathit{Trips}_D^{\gamma_0}$, we have:

$$etar_f(v) = etar_f(\mathsf{path}(v)) = etar_f\Big(\biguplus_{\mathbb{D}\sim:v} S_{\mathbb{D}}\Big)$$

with $S_{\mathbb{D}}$, again, being the heavy notation:

$$S_{\mathbb{D}} := \left\{ \mathsf{Sync}\Big(\pi_s, \{\gamma \mapsto \vec{\phi}_\gamma \mid \gamma \in \Gamma\}\Big) \;\middle|\; \begin{matrix} \pi_s \in \mathit{SynPaths}^{\mathbb{D}}_{(\gamma_0,D)} \\ \forall \gamma \in \Gamma . \vec{\phi}_\gamma \in \mathsf{path}(\,\mathit{Trips}^{\gamma}_{\mathbb{D}(\gamma)}) \end{matrix} \right\}$$

As with the termination probability scenario, we subsequently examine the formula of $etar_f(\biguplus_{\mathbb{D}\sim:v} S_{\mathbb{D}})$. Adhering to identically the same logic as for the termination probability analysis, we deduce that:

$$etar_f\Big(\biguplus_{\mathbb{D}\sim:v} S_{\mathbb{D}}\Big) = \sum_{\mathbb{D}\sim:v} etar_f(S_{\mathbb{D}})$$

Further, should Γ be expressed as $\{\gamma_1, \ldots, \gamma_n\}$, then:

$$etar_f(S_{\mathbb{D}}) = \sum_{\pi_s \in \mathit{SynPaths}^{\mathbb{D}}_{(\gamma_0,D)}} \; \sum_{\vec{\phi}_{\gamma_1} \in \mathsf{path}(\,\mathit{Trips}^{\gamma_1}_{\mathbb{D}(\gamma_1)})} \cdots \sum_{\vec{\phi}_{\gamma_n} \in \mathsf{path}(\,\mathit{Trips}^{\gamma_n}_{\mathbb{D}(\gamma_n)})} X$$

with X being:

$$wt\Big(\mathsf{Sync}(\pi_s, \{\gamma_1 \mapsto \vec{\phi}_{\gamma_1}, \ldots, \gamma_n \mapsto \vec{\phi}_{\gamma_n}\})\Big) \cdot \mathsf{rwd}_f\Big(\mathsf{Sync}(\pi_s, \{\gamma_1 \mapsto \vec{\phi}_{\gamma_1}, \ldots, \gamma_n \mapsto \vec{\phi}_{\gamma_n}\})\Big)$$

Thus, the focus will be to define $\mathsf{rwd}_f(\mathsf{Sync}(\pi, f'))$. Owing to the described attribute of Sync in the analysis of the termination probability, which entails merely concatenating the rules in a specific sequence from both its arguments without omitting or replicating any rules, we deduce:

$$\mathsf{rwd}_f(\mathsf{Sync}(\pi, f')) = \mathsf{rwd}_f(\pi) + \sum_{\gamma \in \Gamma} \mathsf{rwd}_f(f'(\gamma))$$

Moreover, with the established Equation (4.40), it is evident that the aforementioned X component equals:

$$wt(\pi_s) \cdot \prod_{\gamma \in \Gamma} wt(\vec{\phi}_\gamma) \cdot \Big(\mathsf{rwd}_f(\pi_s) + \sum_{\gamma \in \Gamma} \mathsf{rwd}_f(\vec{\phi}_\gamma)\Big)$$

This can subsequently be expanded into the following sum:

$$\begin{aligned} &\Big(wt(\pi_s)\mathsf{rwd}_f(\pi_s) \cdot \prod_{\gamma \in \Gamma} wt(\vec{\phi}_\gamma)\Big) + \sum_{\gamma \in \Gamma} wt(\pi_s) \cdot \mathsf{rwd}_f(\vec{\phi}_\gamma) \cdot \prod_{\gamma \in \Gamma} wt(\vec{\phi}_\gamma) \\ = &\Big(wt(\pi_s)\mathsf{rwd}_f(\pi_s) \cdot \prod_{\gamma \in \Gamma} wt(\vec{\phi}_\gamma)\Big) + \sum_{\gamma \in \Gamma} wt(\pi_s) \cdot wt(\vec{\phi}_\gamma)\mathsf{rwd}_f(\vec{\phi}_\gamma) \cdot \prod_{\gamma' \in \Gamma \setminus \{\gamma\}} wt(\vec{\phi}_{\gamma'}) \end{aligned}$$

Here, we extract the corresponding $wt(\vec{\phi}_\gamma)$ from the summand for γ to facilitate the exposition below.

We then consider each summand separately in the context of $etar_f(S_\mathbb{D})$. Let the first one part above by Y_0, that is:

$$Y_0 := wt(\pi_s)\mathsf{rwd}_f(\pi_s) \cdot \prod_{\gamma \in \Gamma} wt(\vec{\phi}_\gamma)$$

And let Y_i for $i \in \{1, \ldots, n\}$, recall that $n = |\Gamma|$, be:

$$Y_i := wt(\pi_s) \cdot wt(\vec{\phi}_{\gamma_i})\mathsf{rwd}_f(\vec{\phi}_{\gamma_i}) \cdot \prod_{\gamma' \in \Gamma \setminus \{\gamma_i\}} wt(\vec{\phi}_{\gamma'})$$

Notice then that:

$$\begin{aligned} &\sum_{\pi_s \in SynPaths^{\mathbb{D}}_{(\gamma_0,D)}} \sum_{\vec{\phi}_{\gamma_1} \in \mathsf{path}(\, Trips^{\gamma_1}_{\mathbb{D}(\gamma_1)})} \cdots \sum_{\vec{\phi}_{\gamma_n} \in \mathsf{path}(\, Trips^{\gamma_n}_{\mathbb{D}(\gamma_n)})} Y_0 \\ &= etar_f(SynPaths^{\mathbb{D}}_{(\gamma_0,D)}) \cdot \prod_{\gamma \in \Gamma} wt(\, Trips^{\gamma}_{\mathbb{D}(\gamma)}) \end{aligned}$$

Further, for any i:

$$\begin{aligned} &\sum_{\pi_s \in SynPaths^{\mathbb{D}}_{(\gamma_0,D)}} \sum_{\vec{\phi}_{\gamma_1} \in \mathsf{path}(\, Trips^{\gamma_1}_{\mathbb{D}(\gamma_1)})} \cdots \sum_{\vec{\phi}_{\gamma_n} \in \mathsf{path}(\, Trips^{\gamma_n}_{\mathbb{D}(\gamma_n)})} Y_i \\ &= wt(SynPaths^{\mathbb{D}}_{(\gamma_0,D)}) \cdot etar_f(\, Trips^{\gamma_i}_{\mathbb{D}(\gamma_i)}) \cdot \prod_{\gamma' \in \Gamma \setminus \{\gamma_i\}} wt(\, Trips^{\gamma'}_{\mathbb{D}(\gamma')}) \end{aligned}$$

Combining the results, we reach:

$$\begin{aligned} etar_f(S_\mathbb{D}) = \; & etar_f(SynPaths^{\mathbb{D}}_{(\gamma_0,D)}) \cdot \prod_{\gamma \in \Gamma} wt(\, Trips^{\gamma}_{\mathbb{D}(\gamma)}) \\ & + \sum_{\gamma \in \Gamma} wt(SynPaths^{\mathbb{D}}_{(\gamma_0,D)}) \cdot etar_f(\, Trips^{\gamma}_{\mathbb{D}(\gamma)}) \cdot \prod_{\gamma' \in \Gamma \setminus \{\gamma\}} wt(\, Trips^{\gamma'}_{\mathbb{D}(\gamma')}) \end{aligned}$$

In this point, we have reached the decomposition equation of the expected accumulated reward:

$$etar_f(\, Trips^{\gamma}_{D}) = \sum_{\mathbb{D} \sim: v} \left(\begin{array}{ll} & etar_f(SynPaths^{\mathbb{D}}_{(\gamma,D)}) \cdot \prod_{\gamma' \in \Gamma} wt(\, Trips^{\gamma'}_{\mathbb{D}(\gamma')}) \\ + & \sum_{\gamma' \in \Gamma} \left(\begin{array}{ll} & wt(SynPaths^{\mathbb{D}}_{(\gamma_0,D)}) \\ \cdot & etar_f(\, Trips^{\gamma'}_{\mathbb{D}(\gamma')}) \\ \cdot & \prod_{\gamma'' \in \Gamma \setminus \{\gamma'\}} wt(\, Trips^{\gamma''}_{\mathbb{D}(\gamma'')}) \end{array} \right) \end{array} \right) \tag{4.49}$$

The Equation System

Then, to construct an equation system, this time, it is insufficient to rely solely on the decomposition equation of the pure expected total accumulated reward; we must also consider the decomposition of weights. Therefore, in this instance, to formulate an equation system, $etar_f(\, Trips^{\gamma}_{D})$ the vector of variables is no longer merely

indexed by *Trips*-patterns, but we should instead utilise the symbols $wt_{Trips_D^\gamma}$ and $etar_{Trips_D^\gamma}$ for indexing the vector, with the corresponding variable being substituted by these symbols. We can then define the corresponding function $\mathcal{F}_{etar_f(Trips)}$ as follows. For the symbol $wt_{Trips_D^\gamma}$: $\mathcal{F}_{etar_f(Trips)}(\vec{x})[wt_{Trips_D^\gamma}]$ represents the RHS of Equation (4.41) for the pattern $Trips_D^\gamma$, replacing each occurrence of $Trips_{D'}^{\gamma'}$ with $\vec{x}[wt_{Trips_{D'}^{\gamma'}}]$. Whereas for the symbol $etar_{Trips_D^\gamma}$: $\mathcal{F}_{etar_f(Trips)}(\vec{x})[etar_{Trips_D^\gamma}]$ is the RHS of Equation (4.49) with instances of $etar_f(Trips_{D'}^{\gamma'})$ substituted by $\vec{x}[etar_{Trips_{D'}^{\gamma'}}]$ for all D'-γ', and instances of $wt(Trips_{D'}^{\gamma'})$ replaced by $\vec{x}[wt_{Trips_{D'}^{\gamma'}}]$ for all pairs D'-γ'.

Consequently, the value vector $\vec{\Psi}_{etar_f(Trips)}$ is determined by setting its dimensions $wt_{Trips_D^\gamma}$ by the corresponding value $wt(Trips_D^\gamma)$ while the dimensions $etar_{Trips_D^\gamma}$ by $etar_f(Trips_D^\gamma)$. Once again, we default $\vec{\Psi}$ to $\vec{\Psi}_{etar_f(Trips)}$ and $\mathcal{F}$ to $\mathcal{F}_{etar_f(Trips)}$ in the remainder of this section.

4.4.2 Analysis via the Equation System

Following the same rationale as for termination probability, we could derive that the target value is again the LFP of the derived equation system:

Theorem 4.50. *$\vec{\Psi}$ is the LFP of $\mathcal{F}$, with the value in each dimension within the domain of $\mathbb{R}_\infty$.*

Proof. This follows the identical reasoning as in Theorem 4.43. It is noteworthy that the value in $\vec{\Psi}$ might now be ∞ rather than solely real numbers. □

Target Value Subsequently, with this result, we ought to consider how to represent the value of $etar_f(\mathcal{A})$, which, given that every step of a transition is significant, diverges from the straightforward case in the termination probability analysis where the new rules added for bottom from the assumption do not impact the result.

Firstly, let us define the notation for the special rules at $\perp$ derived from the assumption of $\mathcal{A}$. We denote $\delta_\perp$ as the rule $(q_{init}, m_{init}, \perp) \to (q_{init}, m_{init}, up_{\gamma_\perp})$. And let $\delta_\perp^{(q)}$ represent the terminating rule $(q, m_{init}, \perp) \to (q, m_{init}, down)$ for all q.

By definition, that is, Equation (3.14), we engage with the set of terminating runs $\mathbf{Run}_{ter}$. We partition this set according to their state prior to termination, denoted $\mathbf{Run}_{ter}^{(q)}$ for all $q \in \mathcal{Q}$. Thus, we have:

$$etar_f(\mathcal{A}) = \sum_{q \in \mathcal{Q}} etar_f(\mathbf{Run}_{ter}^{(q)})$$

Subsequently, a run $\mu \in \mathbf{Run}_{ter}^{(q)}$ must take the form $\mu = c_{init}\mu'(q, t, \varepsilon)\top$ with $[\mu'] \in Trips_{[q_{init}, q]}^{\gamma_\perp}$ and t being the same as the one in the last configuration in μ'. Note that from the view of path, $\mathsf{path}(\mu) = \delta_\perp \mathsf{path}(\mu')\delta_\perp^{(q)}$. Additionally, the value $wt(\mu) = wt(\mu') = wt([\mu'])$. Therefore, we ascertain:

$$wt(\mu) \cdot \mathsf{rwd}_f(\mu) = wt([\mu']) \cdot \mathsf{rwd}_f([\mu']) + wt([\mu']) \cdot (\mathsf{rwd}_f(\delta_\perp) + \mathsf{rwd}_f(\delta_\perp^{(q)}))$$

Conversely, given that every $[\mu] \in \mathit{Trips}^{\gamma_\perp}_{[q_{init},q]}$ must also result in $c_{init}\mu(q,t,\varepsilon)\top \in \mathbf{Run}^{(q)}_{ter}$ with t being the same with that in the last configuration of μ, we deduce:

$$\mathit{etar}_f(\mathbf{Run}^{(q)}_{ter}) = \mathit{etar}_f(\mathit{Trips}^{\gamma_\perp}_{[q_{init},q]}) + \mathit{wt}(\mathit{Trips}^{\gamma_\perp}_{[q_{init},q]}) \cdot (\mathsf{rwd}_f(\delta_\perp) + \mathsf{rwd}_f(\delta^{(q)}_\perp))$$

Hence, we arrive at the final target as:

$$\mathit{etar}_f(\mathcal{A}) = \sum_{q\in\mathcal{Q}} \mathit{etar}_f(\mathit{Trips}^{\gamma_\perp}_{[q_{init},q]}) + \mathit{wt}(\mathit{Trips}^{\gamma_\perp}_{[q_{init},q]}) \cdot (\mathsf{rwd}_f(\delta_\perp) + \mathsf{rwd}_f(\delta^{(q)}_\perp))$$

Quantitative Inquiry Subsequently, we are prepared to present the proof of Theorem 4.46.

Proof (Theorem 4.46). The proof broadly aligns with that of the termination probability, albeit with the following distinction.

As the EToR determines whether a query encompasses all real numbers, by enquiring $\vec{x} \geq \vec{0} \wedge \vec{x} = \mathcal{F}(\vec{x})$, we are effectively questioning if it holds that the LFP does not contain any ∞ value. Nonetheless, enquiring over the entirety of $\mathcal{F}$ might impede our goal – as our objective is merely to ascertain whether the value within the LFP of the dimensions $\mathit{etar}_{\mathit{Trips}^{\gamma_\perp}_{[q_{init},q]}}$ for all q is finite and if it is, further quantify the value with c and the comparator $\sim$. Consequently, we should consider solely the segment of $\mathcal{F}$ that impacts the dimension of $\mathit{etar}_{\mathit{Trips}^{\gamma_\perp}_{[q_{init},q]}}$ for all $q \in \mathcal{Q}$ and eliminate the dimensions $\mathit{etar}_{\mathit{Trips}^{\gamma}_{D}}$ that do not impact these dimensions.

More precisely, by firstly performing standard polynomial analysis, one could identify all the 0 value dimensions in the LFP, which are then replaced by 0 in the equation system and omitted from the disucssion. Then, a dimension v is deemed to impact another dimension v' if in $\mathcal{F}(\vec{x})[v']$, the variable $\vec{x}_v$ appears in a term with a non-negative coefficient, or there exists another dimension v'' that impacts v' and v influences v''. This, again, can be ascertained through conventional techniques of polynomial equation system analysis.

Thereafter, let the resulting sub-function of $\mathcal{F}$ be $\mathcal{F}'$, devoid of any $\mathit{etar}_{\mathit{Trips}^{\gamma}_{D}}$ dimension that are with 0 value in LFP or fails to impact any of the dimensions $\mathit{etar}_{\mathit{Trips}^{\gamma_\perp}_{[q_{init},q]}}$ for $q \in \mathcal{Q}$. We then inquire on $\mathcal{F}'$ rather than $\mathcal{F}$ when invoking EToR. Subsequently, if $\exists \vec{x}.\vec{x} \geq \vec{0} \wedge \vec{x} = \mathcal{F}'(\vec{x})$ does not hold, it signifies that the $\mathit{etar}_f(\mathcal{A})$ is ∞. If it holds, we can further quantify the result to perform the quantitative analysis with the constant number c and the comparator $\sim$ following exactly the same methods for that of the quantitative termination probability analysis above.

The complexity analysis similarly derives directly from the termination probability analysis, noting that the increased complexity of each term for a fixed $\mathbb{D}$ does not alter our conclusion. □

Example 4.51. Here, we address the problem of the expected length of the random copy language, i.e. Example 4.1, which is equivalent to the conditional expected termination time of $\mathcal{A}_{COPY}$ defined therein.

As in Example 4.45, we may derive the following system of equations for the patterns $v_\perp = \mathit{Trips}^{\gamma_\perp}_{[q_f,q_d]}$ and $v = \mathit{Trips}^{\gamma}_{[q_f,q_d,q_s,q_d]}$.

$$\begin{aligned} mht(v_\perp) &= mht(v) \\ mht(v) &= pT + 2(p_A^2 + p_B^2) \cdot wt(v) + (p_A^2 + p_B^2) \cdot mht(v) \\ wt(v) &= pT + (p_A^2 + p_B^2) \cdot wt(v) \end{aligned}$$

Hence, using the established algorithm, we compute the least non-negative solution for the equation system for the weights and mean hitting time, thereby obtaining the final result for the expected termination time:

$$ett(\mathcal{A}_{COPY}) = mht(v) = \frac{p_T \cdot (1 + p_A^2 + p_B^2)}{(1 - (p_A^2 + p_B^2))^2}$$

An intuitive explanation can also be presented as follows. We define $r = p_A^2 + p_B^2$ for simplicity. First, let S be the random variable of a generated string, and $S \in \text{COPY}$ denote the event that it is a desired string in the COPY language. Recall from Example 4.45 that the probability of a string $w\#w$ of length $2n+1$ (n denotes the length of w) is:

$$\mathbb{P}\left[S \in \text{COPY}, |S| = 2n+1\right] = p_T \cdot (p_A^2 + p_B^2)^n$$

The expected length E is:

$$\begin{aligned} & E \\ &= 1 \cdot \mathbb{P}\left[S \in \text{COPY}, |S| = 1\right] + 3 \cdot \mathbb{P}\left[S \in \text{COPY}, |S| = 3\right] + \ldots \\ &= p_T + 3p_T r + 5p_T r^2 + 7p_T r^3 + \ldots \\ &= p_T \cdot (1 + 3r + 5r^2 + 7r^3 + \ldots) \end{aligned}$$

After standard calculations, we arrive at:

$$E = p_T \cdot \frac{1 + r}{(1 - r)^2}$$

Substituting back $r = p_A^2 + p_B^2$, we get:

$$E = p_T \cdot \frac{1 + p_A^2 + p_B^2}{(1 - (p_A^2 + p_B^2))^2}$$

This is exactly the result derived by our expected termination time analysis.

Finally, conditioning on the prefixes that satisfy the target property, the expected length of such prefixes is then:

$$cett(\mathcal{A}_{COPY}) = \frac{ett(\mathcal{A}_{COPY})}{tp(\mathcal{A}_{COPY})} = \frac{1 + p_A^2 + p_B^2}{1 - (p_A^2 + p_B^2)}$$

The global and the local dance in intricate patterns, never fully merging yet always in dialogue. Like parallel lines, they maintain their distinct identities while shaping the landscape together.

— Eliza Chen, Patterns of Complexity

5 Linear-Time Model Checking

Contents

In this chapter, we tackle linear-time model checking problems, focusing on rPTSA against ω-regular and LTL properties.

Linear-time model checking is a fundamental aspect of formal verification, enabling us to verify both the short-term and long-term behaviour of a system. This technique is vital for ensuring the correctness and reliability of complex systems.

Our examination begins with local ω-regular model checking, where we establish its decidability and present an algorithm that further leverages the patterns and decomposition principles introduced in the previous chapter. This result also addresses the decidability of local quantitative LTL model checking for rPTSA. To clarify, it is a well-known fact that LTL properties are a subset of ω-regular properties, which allows us to extend our findings from ω-regular properties to LTL properties.

We then shift our focus to global model checking for rPTSA against LTL properties. This transition broadens our scope from considering only initial runs to examining all possible runs. We prove decidability for this problem as well. Global model checking aims to identify all configurations that almost surely satisfy

a given property, taking into account the entire state space for a comprehensive analysis of the system's behaviour.

Notably, this exploration of global model checking also serves as a crucial step towards branching-time model checking, which we will delve into in the next chapter.

Assumptions Regarding the rPTSA Throughout this chapter, we fix a particular rPTSA $\mathcal{A} = (\mathcal{Q}, \mathcal{M}, \Gamma, \Delta, q_{\text{init}}, m_{\text{init}})$. To simplify the discussion, we impose certain constraints on the rPTSA without compromising the overall generality of the approach, notably eliminating explicit termination.

More specifically, we stipulate that $\mathcal{A}$ contains no termination rules. That is, rules of the form $(-, -, \bot) \xrightarrow{p} (-, -, \mathit{down})$ are absent. This condition can be achieved for any rPTSA by introducing a unique stack symbol $\gamma_\top$, where termination is modelled through rules that indefinitely ascend towards $\gamma_\top$ starting from q_{init}. Formally, all rules of the form:

$$(q, m, \bot) \xrightarrow{p} (q', m', \mathit{down})$$

for all $q, q' \in \mathcal{Q}$ and $m, m' \in \mathcal{M}$, are substituted with:

$$(q, m, \bot) \xrightarrow{p} (q, m', \mathit{up}_{\gamma_\top})$$

Additionally, rules of the form:

$$(q, m, \gamma_\top) \to (q, m, \mathit{up}_{\gamma_\top})$$

are introduced for every q and m. Regarding the valuation or interpretation maps, configurations with a stack pointer ending with $\gamma_\top$ ought to be mapped to the same value as the termination configuration $\top$.

With this assumption, a desirable property is then achieved: the concept of the trajectory of $\mathscr{M}(\mathcal{A})$ and the run of $\mathcal{A}$ coincide exactly.

5.1 ω-Regular Model Checking

In this section, we address the problem of local ω-regular model checking. To be more precise, we focus on developing an algorithm to prove the following theorem:

Theorem 5.1. *Given an rPTSA $\mathcal{A}$ and a DRA $\mathcal{D}$ along with a* simple *interpretation map f, a constant c (ranging between 0 and 1) and a comparator $\sim \in \{\leq, <, =, \neq, >, \geq\}$, the problem of determining whether $\mathbb{P}[\mathcal{A} \models_f \mathcal{D}] \sim c$ is decidable.*

Our approach can be regarded as an adaptation of the method presented by Brazdil et al. [110] for probabilistic pushdown automata (pPDA), tailored specifically to the context of rPTSA. The adaptation is analogous to the relationship between the patterns of rPTSA and those for pPDA, as described in Section 4.1.2 — we employ *Up* patterns as counterparts to the $\langle qX \uparrow \rangle$ variables used in the original method. Both serve the critical purpose of investigating the long-term running

behaviour of their respective automata. However, this extension is non-trivial and relies heavily on the decomposition principle established in the previous chapter.

To be more specific, our introduction of the model checking procedure starts with the discussion on some key supportive structures. Specifically, we examine the weights of the *Up* patterns, including their inductive characterisation using equation systems and their actual values.

With this information, we can present the local model checking algorithm, which begins by constructing a supportive Markov chain. The states of this chain are the *Up* patterns, and the transition probabilities are the conditional transition values derived from the equation characterisation and the actual values obtained previously. The Markov chain is then analysed as a graph, where the *bottom strongly connected components* (BSCCs) fundamentally represent the long-term (infinite) behaviours. Consequently, the BSCCs are classified as either "good" or "bad", depending on whether they satisfy the acceptance condition of the given DRA. Finally, the target local model checking value is determined by the probability of reaching the "good" BSCCs.

Symbolic Conventions Throughout this section, we fix a specific DRA $\mathcal{D} = (\mathfrak{D}, \Sigma, \mathfrak{T}, \mathscr{d}_{init}, \mathscr{R})$ and a *simple* valuation f. Given the nature of a simple valuation, whereby the target symbol relies solely on the status of a given configuration, we denote $\mathscr{d} \xrightarrow{(\mathscr{q},\mathscr{m},\mathscr{g})} \mathscr{d}'$ to imply that $\mathfrak{T}(\mathscr{d}, f(\mathscr{c})) = \mathscr{d}'$ for some $\mathscr{c}$ with $\mathsf{status}(\mathscr{c}) = (\mathscr{q}, \mathscr{m}, \mathscr{g})$.

Moreover, to simplify the presentation, we impose a similar restriction as in the termination analysis – limiting transitions at the stack's bottom. Specifically, we stipulate again that a distinct stack symbol $\gamma_\bot$ is responsible for bottom transitions, with the only meaningful transition rule being $(\mathscr{q}_{init}, \mathscr{m}_{init}, \bot) \to (\mathscr{q}_{init}, \mathscr{m}_{init}, up_{\gamma_\bot})$ (complemented by $(\mathscr{q}, \mathscr{m}, \bot) \to (\mathscr{q}, \mathscr{m}, up_{\gamma_\bot})$ for all $\mathscr{q}$ and $\mathscr{m}$ to ensure the transition's totality).

This is easily achieved by introducing a distinct delegate bottom stack symbol $\gamma_\bot$, to which all behaviours for $\bot$ are transferred, leaving $\bot$ with only the aforementioned rules. Furthermore, in the $\mathcal{D}$, we have $\mathscr{d}_{init} \xrightarrow{(\mathscr{q},\mathscr{m},\bot)} \mathscr{d}_{init}$ for all $\mathscr{q}$ and $\mathscr{m}$, to preserve the initial state while ensuring totality.

5.1.1 Weights of *Up* Patterns

In the preceding chapter on termination analysis, we meticulously derived the decomposition equation for the weights of *Trips* patterns, culminating in the formulation of Equation (4.41) from the more general pre-path decomposition presented as Equation (4.21). Following a parallel line of reasoning, we now proceed to derive the corresponding decomposition for the weights of *Up* patterns.

$$wt(\mathit{Up}^{\gamma}_{D}) := \sum_{\mathbb{D} \sim: \mathit{Up}^{\gamma}_{D}} wt(\mathit{SynPaths}^{\mathbb{D}}_{(\gamma,D)}) \cdot \prod_{\gamma' \in \Gamma} wt(\mathit{TripsOrUp}^{\gamma'}_{\mathbb{D}(\gamma')}) \tag{5.2}$$

This derivation adopts the same methodological framework as that employed

in Section 4.3.1 for the analysis of *Trips* patterns, with only a minor divergence, which lies in the specific definition adopted for the *Up* patterns. We present here a succinct yet comprehensive proof to substantiate the validity of this derivation.

Proof. As mentioned, the overall strategy to proceed the proof would be to mirror that for the *Trips* patterns above in Section 4.3.1. However, given the more rigorously defined concept of the weights of *Up* patterns, we observe that the weights of their associated pre-path patterns remain undefined at this juncture. To resolve this ambiguity, we explicitly define the weights for a specific class of sets comprising infinite pre-paths.

Formally, let S be a set of infinite pre-paths. The concept of weights within this context is well-defined under the following two stringent conditions: (1) every pre-path $\vec{\phi}$ in the set has exactly one infinite component, specifically its last path; and, (2) for any given finite pre-path $\vec{\phi}$, there exists a configuration that facilitates all infinite paths ϕ such that the concatenated pre-path $\vec{\phi} +\!\!+ [\phi]$ belongs to the set S. For a set satisfying these criteria, the weight $wt(S)$ is determined by the summation of components across all finite pre-paths $\vec{\phi}$. Each component for a pre-path $\vec{\phi}$ is computed as the product of the weight of $\vec{\phi}$ and the probability $\mathbb{P}\left[\{\mathsf{host}(c, \phi) \mid \vec{\phi} +\!\!+ [\phi] \in S\}\right]$, where c represents one of the configurations enabling all paths ϕ such that $\vec{\phi} +\!\!+ [\phi]$ is an element of S.

It becomes evident that for any given Up_D^γ, the corresponding set of pre-paths $\mathsf{path}(Up_D^\gamma)$ satisfies these conditions, ensuring that its weight is well-defined and equivalent to $wt(Up_D^\gamma)$ as established in Equation (4.10) – that is, formally, $wt(\mathsf{path}(Up_D^\gamma)) = wt(Up_D^\gamma)$.

With this definition for the pre-paths in place, the remainder of the proof derivation straightforwardly mirrors that for the *Trips* patterns detailed in Section 4.3.1. □

Considering the derived equation for the *Up* patterns, alongside the analogous ones for *Trips* patterns, by substituting the patterns with their corresponding variables in a pattern-indexed vector of variables, similar to the approach in Section 4.3.1, one directly obtains a comprehensive system of equations, which we denote as $\mathcal{F}_{wt(\mathit{TripsOrUp})}$. In this system, the pattern-indexed vector $\vec{\Psi}_{wt(\mathit{TripsOrUp})}$, where $\vec{\Psi}_{wt(\mathit{TripsOrUp})}(\mathit{TripsOrUp}_D^\gamma) := wt(\mathit{TripsOrUp}_D^\gamma)$, is guaranteed to represent a fixed point.

However, despite the close resemblance between the decompositions and equations of *Up* and *Trips* patterns, significant challenges arise when attempting to analyse the equation system associated with *Up* patterns for determining their weights. One major obstacle is that, unlike their *Trips* counterparts, the weights for *Up* patterns cannot be straightforwardly expressed as the least fixed point (LFP) of the system. This fact could be simply seen by that the equation Equation (5.2) is intrinsically *homogeneous* with respect to *Up* patterns, which leads to that the LFP value must be 0.

To overcome this issue, we propose an alternative method to accurately derive the weights of *Up* patterns. This approach involves the utilisation of the following

equation for all $Up^{\gamma}_{D \Rightarrow [q]}$ patterns:

$$wt(Trips^{\gamma}_{D}) = wt(Up^{\gamma}_{D \Rightarrow [q]}) + \sum_{q' \in Q} wt(Trips^{\gamma}_{D \Rightarrow [q,q']}) \tag{5.3}$$

This equation stems from the insight that the pre-runs starting from q at the subject node after completing D could be divided into the following two distinct scenarios: those that after completing D, do not transition *down* again and those that do, transitioning *down* with some state q'. The former scenario, without further restrictions, directly equates to $wt(Trips^{\gamma}_{D})$.

To elucidate this reasoning, the following proof offers a formal account.

Proof. We first present some auxiliary symbols. Consider a pre-run $\vec{\mu} \in Trips^{\gamma}_{D}$. We define $c^{(\vec{\mu})}$ as the configuration from which the pattern $Up^{\gamma}_{D \Rightarrow [q]}$ begins its final infinite runs after the prefix $\vec{\mu}$. For the set of infinite trajectories stemming from a specific configuration c within $\mathcal{M}(\mathcal{A})$, its subset that eventually reach the bottom of the stack with state q' (at their first encounter) are denoted by $c \rightsquigarrow_{q'} \bot$, and those that never visit the bottom are denoted by $c \not\rightsquigarrow \bot$. It is observed that the complete set of trajectories, $\mathsf{InfTraj}(c)$, is the disjoint union of $c \not\rightsquigarrow \bot$ and all $c \rightsquigarrow_{q'} \bot$ for each $q' \in Q$.

With the above definitions and observations, we derive that:

$$\begin{aligned}
& wt(Trips^{\gamma}_{D}) \\
= & \sum_{\vec{\mu} \in Trips^{\gamma}_{D}} wt(\vec{\mu}) \\
= & \sum_{\vec{\mu} \in Trips^{\gamma}_{D}} wt(\vec{\mu}) \cdot \mathbb{P}\left[\mathsf{InfTraj}(c^{(\vec{\mu})})\right] \\
= & \sum_{\vec{\mu} \in Trips^{\gamma}_{D}} wt(\vec{\mu}) \cdot \left(\mathbb{P}\left[c^{(\vec{\mu})} \not\rightsquigarrow \bot\right] + \sum_{q' \in Q} \mathbb{P}\left[c^{(\vec{\mu})} \rightsquigarrow_{q'} \bot\right]\right) \\
= & \left(\sum_{\vec{\mu} \in Trips^{\gamma}_{D}} wt(\vec{\mu}) \cdot \mathbb{P}\left[c^{(\vec{\mu})} \not\rightsquigarrow \bot\right]\right) + \left(\sum_{q' \in Q} \sum_{\vec{\mu} \in Trips^{\gamma}_{D}} wt(\vec{\mu}) \cdot \mathbb{P}\left[c^{(\vec{\mu})} \rightsquigarrow_{q'} \bot\right]\right) \\
= & \; wt(Up^{\gamma}_{D \Rightarrow [q]}) + \sum_{q' \in Q} wt(Trips^{\gamma}_{D \Rightarrow [q,q']})
\end{aligned}$$

Additionally, note that in the final equality, the probability $\mathbb{P}\left[c^{(\vec{\mu})} \rightsquigarrow_{q'} \bot\right]$ equates to simply the weight of all finite runs until the first encounter with the stack bottom with state q', which then matches precisely the definition of the final segment of pre-runs in $Trips^{\gamma}_{D \Rightarrow [q,q']}$. □

5.1.2 The Algorithm

Following the elucidation of the equations for the decomposition of weights of *Up* patterns and the qualification of their actual values, we are poised to detail the local ω-regular model checking algorithm. This section outlines the algorithm step-by-step.

The Product rPTSA The first step involves constructing the product rPTSA, $\mathcal{A} \times \mathcal{D}$, which intuitively integrates the states of $\mathcal{D}$ into $\mathcal{A}$. This amalgamation does not alter the behaviours of $\mathcal{A}$, but simply allow $\mathcal{D}$ to operate alongside by observing the behavior (transitions between statuses) of $\mathcal{A}$.

Formally, the product rPTSA is $\Big(\mathcal{Q} \times \mathfrak{D}, \mathcal{M}, \Gamma, \Delta^{(\mathcal{A}\times\mathcal{D})}, (q_{init}, d_{init}), m_{init}\Big)$. The new transition function $\Delta^{(\mathcal{A}\times\mathcal{D})}$ includes the rule

$$\Big((q, d), m, g\Big) \xrightarrow{p} \Big((q', d'), m', up_\gamma\Big)$$

if and only if $(q, m, g) \xrightarrow{p} (q', m', up_\gamma)$ exists in $\mathcal{A}$ and $d \xrightarrow{(q,m,g)} d'$ occurs in $\mathcal{D}$. Similarly, the rule

$$\Big((q, d), m, g\Big) \xrightarrow{p} \Big((q', d'), m', down\Big)$$

is included in $\mathcal{A} \times \mathcal{D}$ if $(q, m, g) \xrightarrow{p} (q', m', down)$ is in $\mathcal{A}$ and $d \xrightarrow{(q,m,g)} d'$ in $\mathcal{D}$.

Furthermore, the concept of *accepting runs* within the product automaton stipulates that an infinite run μ of $\mathcal{A} \times \mathcal{D}$ is considered accepting if the infinite sequence of DRA states it comprises is accepted by $\mathcal{D}$. For a set S of infinite runs of $\mathcal{A} \times \mathcal{D}$, the subset containing only the accepting runs is denoted by $Acc(S)$. This notion readily extends to the set of infinite pre-runs as well.

This concept is intuitive, as suggested by its name — the sequence of DRA states corresponds exactly to what one obtains by reading the rPTSA run; hence, the rPTSA run is accepted by the DRA if its associated sequence in the product rPTSA is accepted by the DRA. Consequently, it follows that:

$$\mathbb{P}\left[\mathcal{A} \models \mathcal{D}\right] = \mathbb{P}\left[Acc(\mathbf{Run}_{inf}^{(\mathcal{A}\times\mathcal{D})})\right] = \mathbb{P}\left[Acc(\,Up^{\gamma_\perp}_{[(q_{init}, d_{init})]})\right]$$

Notably, as the product rPTSA continues to function as a normal rPTSA, the pattern $Up^{\gamma_\perp}_{[(q_{init}, d_{init})]}$ is naturally utilised.

The Auxiliary Markov Chain Following the construction of the product rPTSA, an auxiliary Markov chain, denoted as $\mathscr{M}^{(\mathcal{A}\times\mathcal{D})}$, is developed using the *Up* patterns of $\mathcal{A} \times \mathcal{D}$. Specifically, this Markov chain comprises states represented by *Up* patterns of $\mathcal{A} \times \mathcal{D}$ with weights ≥ 0. The transition probability from one pattern, Up^{γ}_{D}, to another, $Up^{\gamma'}_{D'}$, is determined as follows:

$$\frac{\sum_{\substack{\mathbb{D} \sim :\, Up^{\gamma}_{D} \\ \mathbb{D}(\gamma') = D'}} wt(SynPaths^{\mathbb{D}}_{(\gamma,D)}) \cdot \prod_{\gamma''} wt(\,TripsOrUp^{\gamma''}_{\mathbb{D}(\gamma'')})}{wt(\,Up^{\gamma}_{D})} \tag{5.4}$$

This probability intuitively represents the conditional probability that, when constrained to the pre-runs within Up^{γ}_{D}, it measures the mass of those pre-runs whose running interface one level above the subject node conforms to D' in direction γ'.

SCC Analysis Viewing the constructed auxiliary Markov chain as a graph, we can perform the standard *strongly connected component* (SCC) analysis. Consider each *bottom SCC* (BSCC) C, we are then interested in the $\mathcal{D}$ states within its transitions, denoted C_{inf}. These states intuitively represent the $\mathcal{D}$ states that are almost surely appearing infinitely for runs reaching this BSCC. Hence, when there exists a component $(E_i, F_i) \in \mathscr{R}$ such that $C_{inf} \cap E_i = \emptyset$ while $C_{inf} \cap F_i \neq \emptyset$, it means runs that reach this BSCC are almost surely accepting. Such components are then classified as "good" components, with those that do not satisfy the property being the "bad" ones. It is straightforward to observe that the probability $\mathbb{P}\left[Acc(Up^{\gamma_\perp}_{[(q_{init},\mathscr{d}_{init})]})\right]$ is equal to the reachability probability from the Markov chain state $Up^{\gamma_\perp}_{[(q_{init},\mathscr{d}_{init})]}$ to the "good" components.

The collection of C_{inf} begins by analysing each transition within the BSCC C, aiming to compute the associated $\mathcal{D}$ states – intuitively, those with a non-zero probability of being involved, referred to as the transition's "footprint". Initially, we identify summands in Equation (5.4) that have a non-zero value, indicating the presence of synchronisation paths with non-zero probabilities and further involving patterns of non-zero weights. Across the non-zero summands, we all collect the *Trips* patterns mentioned as the basis for the recursive collection below.

For each identified *Trips* pattern $Trips^{\gamma}_{D}$, we proceed to examine the patterns mentioned on the right-hand side (RHS) (in Equation (4.41)) of this pattern, with a focus solely on those appearing in non-zero summands. They are termed the "relevant patterns". This recursive examination is repeated for newly discovered *Trips* patterns, progressively expanding the collection to encompass all relevant *Trips* patterns. This recursive phase concludes once no further *Trips* pattern is added to the collection set.

Upon completing the recursive collection, the footprint of a transition is defined by all the $\mathcal{D}$ states mentioned across the interaction sequences of the collected *Trips* patterns, as well as those in Up^{γ}_{D} and $Up^{\gamma'}_{D'}$. Ultimately, C_{inf} is constituted by the union of footprints from all transitions within the BSCC.

5.1.3 Proof of Model Checking

Then, we proceed to prove the final result for the local ω-regular model checking with the above algorithm.

Solution by EToR To achieve the desired result, it suffices to first derive the symbolic representation of the reachability probability and then apply both Theorem 4.43 and Equation (5.3) to formulate a system of qualification equations. However, the employment of weights for the *Up* patterns as determined via Equation (5.3) is contingent upon the variables at the *Trips* dimensions assuming the least qualified outcome.

This necessitates a tricky approach to qualify the LFP of $\mathcal{F}_{wt(Trips)}$. In this context, a dependency graph of *Trips* patterns is constructed, wherein the vertices represent *Trips* patterns, and an edge from $Trips^{\gamma}_{D}$ to $Trips^{\gamma'}_{D'}$ exists if and only if $Trips^{\gamma'}_{D'}$ is present as a non-zero summand on the right-hand side of the decomposition

equation of $wt(\mathit{Trips}_D^\gamma)$ (i.e., Equation (4.41)). Subsequently, the SCC analysis is performed on this dependency graph once more. This time, rather than the bottom SCCs, we consider the top SCCs. For each of them, we identify a pattern. The identified patterns are termed the *anchor patterns*. For each anchor pattern, the objective is to identify the "eigenvalue", referred to as the *LFP upper bound*, which signifies that only the LFP value is less than or equal to it, with all other fixed points exceeding this value.

Concerning each anchor pattern v, the focus shifts to $\mathcal{F}_{v\Rightarrow}$, which denotes the segment of $\mathcal{F}_{wt(\mathit{Trips})}$ where the LHS is accessible from v within the dependency graph. The aim is to determine the LFP value of v. This entails pinpointing a constant $c \in [0,1]$, called an LFP upper bound, ensuring that the LFP value of v does not surpass c while all other FP surpass it.

To ascertain such a value, the following formulation is proposed, delineating two distinct fixed points of $\mathcal{F}_{v\Rightarrow}$:

$$\begin{aligned} & \vec{0} \leq \vec{x}^{(1)} \leq \vec{1} \\ \wedge\ & \vec{0} \leq \vec{x}^{(2)} \leq \vec{1} \\ \wedge\ & \vec{x}^{(1)} = \mathcal{F}_{v\Rightarrow}(\vec{x}^{(1)}) \\ \wedge\ & \vec{x}^{(2)} = \mathcal{F}_{v\Rightarrow}(\vec{x}^{(2)}) \\ \wedge\ & \vec{x}^{(1)}[v] < \vec{x}^{(2)}[v] \end{aligned}$$

Here, $\vec{x}^{(1)}$ and $\vec{x}^{(2)}$ symbolise two sets of existentially quantified variables, each corresponding to distinct fixed points of $\mathcal{F}_{v\Rightarrow}$. The notation $\vec{x}^{(i)}[v]$ references the value of v within the fixed point $\vec{x}^{(i)}$. The inequality $\vec{x}^{(1)}[v] < \vec{x}^{(2)}[v]$ confirms the distinctness of these fixed points. This formulation is referred to as P_v.

Bisection is then employed to identify c, such that $P_v \wedge x^{(1)}[v] \leq c$ is valid, whereas $P_v \wedge x^{(2)}[v] \leq c$ is not. This implies that within the range $[0,1]$, only the LFP value of v is $\leq c$.

In conclusion, by imposing the condition $v \leq c_v$ for *each* anchor *Trips* pattern v, together with its associated upper bound c_v, it is ensured that the entire function $\mathcal{F}_{wt(\mathit{Trips})}$ adheres to the LFP for the *Trips* patterns. This is because, if any set of variables take the value other than the LFP, the increasement will also increase the variables at the anchor patterns.

Utilising the aforementioned strategy, the problem can subsequently be addressed through an EToR enquiry. Yet, as one generally lacks knowledge regarding the proximity of the LFP to other fixed points (FPs), it is typically challenging to analyse complexity in this context.

5.2 Global LTL Model Checking

This chapter addresses the global model checking problem for LTL formulas. We focus on proving the following theorem for qualitative model checking:

Theorem 5.5 (Simple Global Qualitative LTL Model Checking)**.** *For an rPTSA $\mathcal{A}$, an LTL formula φ, and a* simple *valuation ν over $\mathcal{A}$, a TA $\mathcal{T}$ can be constructed such that:*

$$\mathcal{C}(\mathcal{T}) = \{c \in \mathcal{C} \mid \mathbb{P}\left[\{u \in \mathsf{InfTraj}(c) \mid u \models_\nu \varphi\}\right] = 1\}$$

Our proof for this theorem draws inspiration from the global qualitative LTL model checking algorithm for pPDA, as developed in [120]. However, given the complexity of rPTSA, our construction differs significantly from the original procedure.

We will begin by introducing additional concepts crucial to the model checking procedure, particularly those related to "LTL types". This approach, seen in LTL model checking for RMC/pPDA [111, 120], essentially examines the sub-formula satisfiability of the general LTL formula. We then present an overview of the model checking procedure, aiming to convey the intuition behind the technically intricate construction.

Following this overview, we proceed to present the model checking procedure with full technical details. The construction is highly non-trivial due to the inherent complexity of rPTSA. After the construction, we provide the proof in the subsequent Section 5.2.4.

Throughout this section, unless otherwise specified, we consider a fixed rPTSA $\mathcal{A} = (\mathcal{Q}, \mathcal{M}, \Gamma, \Delta, q_{init}, m_{init})$, which also adheres to the non-terminating assumption as stated at the beginning of the chapter. We further assume an LTL formula φ and a *simple* valuation ν. In addition, we impose a *global restriction* on this automaton: the k-restriction is enforced across all runs, rather than being confined to the initial runs as assumed in previous discussions.

5.2.1 Additional Definitions

To begin, we define some additional structures and notations. It is important to note that certain definitions, such as the main branch, side branches, and bottom *Up* patterns, will be used not only in this section but throughout the remainder of this work. We start by introducing some general concepts, followed by more specific categorised notions for each topic.

List Operations of Synchronisation Paths For convenience, we define also the list operations for the synchronisation paths — intuitively for us to conveniently treat such a path as a list. For two paths $\pi_1 = [\zeta_1, \ldots, \zeta_i]_{(m_0, g)}$ and $\pi_2 = [\zeta_{i+1}, \ldots, \zeta_n]_{(m_i, g)}$ such that m_i is the reacting local memory of ζ_i, we define the concatenation of them to be $\pi_1 +\!\!+ \pi_2 := [\zeta_1, \ldots, \zeta_i, \zeta_{i+1}, \ldots, \zeta_n]_{(m_0, g)}$. Then, a path π is called a prefix of another path π', iff there exists some π'' such that $\pi +\!\!+ \pi'' = \pi'$.

Moreover, we may regard a path to be a list of reaction units, and will say something like the ith reaction unit of the path. By default, the indexing is counting from 1.

Extended Directions and Filtrations We define the *extended directions* $\mathcal{K}^{\Uparrow} := \mathcal{K} \uplus \{\Uparrow\}$. This set will function as the *source direction*, indicating the origin of the stack pointer, where $\Uparrow$ intuitively represents the direction of "within the current

node". In particular, we introduce a new case for filtration $\restriction_{\Uparrow}$, which mirrors the standard filtration in Section 4.2.3, except that the initial state of the first reaction unit is not associated with any direction in $\mathcal{K}$. For simplicity, we continue to use κ to range over $\mathcal{K}^{\Uparrow}$, as these extended directions are used exclusively for source directions.

Sets of Synchronisation Paths For the purposes of clarity, further to the default synchronisation paths set $\mathit{SynPaths}^{\mathbb{D}}_{(q,D)}$, we introduce more variants to be used in this section. we denote the fully qualified set of all synchronisation paths that: begin at a given control state q and a specific local memory m with stack status g, featuring downward interaction D and upward interactions $\mathbb{D}$ with the source direction $\kappa \in \mathcal{K}^{\Uparrow}$, as $\mathit{SynPaths}^{(\kappa,\mathbb{D})}_{((q,m),g,D)}$. Formally:

$$\left\{ \pi \in \Pi \;\middle|\; \begin{array}{c} \pi = [(q,-,-,-),\ldots]_{(m,g)} \\ \pi \restriction_{\kappa} \Downarrow = D \\ \forall \gamma \in \Gamma . \pi \restriction_{\kappa} \gamma = \mathbb{D}(\gamma) \end{array} \right\} \tag{5.6}$$

Notably, when κ is not explicitly mentioned, we default it to $\Downarrow$ to maintain consistency with previous usage.

Additionally, we define $\mathit{SynPaths}^{\mathbb{D}}_{(m,g,D)}$ to relax the control state constraint, which modifies the first line of Equation (5.6) to require $\pi = [\ldots]_{(m,g)}$. Note that this is NOT simply the union of all $\mathit{SynPaths}^{\mathbb{D}}_{((q,m),g,D)}$ for all q, since, with q, the first reaction unit must be examined, and thus the original case (Equation (5.6)) cannot include the empty path ε. In contrast, as no reaction unit is examined in this case, it can include the empty path ε, as long as $D = \varepsilon$ and $\mathbb{D} = \lambda - .\varepsilon$. [1]

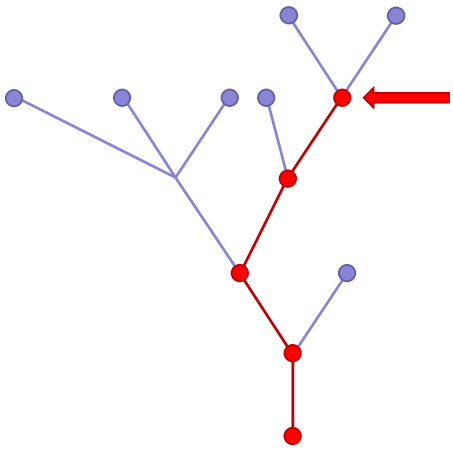

Figure 5.1: Side Branches and Main Branch

The Main Branch and Side Branches For any configuration, we call the path from where the stack pointer is pointing at to the bottom as the *main branch*, with all other nodes considered as at *side branches*. Refer to Figure 5.1 for an illustration. The red arrow indicates the stack pointer, with red nodes denoting the main-branch nodes and blue nodes representing the side-branches nodes.

Reading Tree Then, we introduce the concept of the *reading tree*, which intuitively attaches to each node of an accepting tree additional information about the specific tree-automaton state appearing at that node when accepting the tree.

Consider a tree automaton $\mathcal{T}$ and a $(\mathcal{I}, \Sigma)$-tree t that is accepted by $\mathcal{T}$. We define a corresponding tree t', sharing the same domain as t, to be a *reading tree* of $\mathcal{T}$ accepting t if: for every node x in $\mathrm{dom}(t)$, let $(\mathtt{a}, t) := t'(x)$, then: (1) $\mathtt{a} = t(x)$;

[1] Recall that the empty path ε can denoted any path of the form $[]_{(m,g)}$ for all m and g.

(2) t is an accepting state of the subtree originating from x in $\mathscr{t}$; and, (3) there exists a transition $r = (t, \mathtt{a}, f)$ in $\mathcal{T}$ such that for all $i \in \Sigma(\mathtt{a})$, $\mathsf{snd}(\mathscr{t}'(xi)) \in f(i)$. The element t is termed the *reading state* at the node at at route x. We straightforwardly refer to the rule r as the rule *applied* to this node at route x.

A reading tree for $\mathcal{T}$ accepting $\mathscr{t}$ provides a visual representation of how $\mathscr{t}$ is processed and accepted by $\mathcal{T}$, highlighting the accepting states at each node during the acceptance of $\mathscr{t}$.

It is evident from definitions that a reading tree for $\mathcal{T}$ accepting $\mathscr{t}$ exists if and only if $\mathscr{t}$ is indeed accepted by $\mathcal{T}$. This can be straightforwardly established by referring directly to the definitions. When dealing with general tree automata $\mathcal{T}$, it is possible to have multiple reading trees for a single accepting tree $\mathscr{t}$; however, distinguishing among these multiple possibilities is generally not crucial for our model-checking purpose [2]. Thus, we will denote one potential reading tree for $\mathcal{T}$ accepting $\mathscr{t}$ as $\mathsf{RdT}_{\mathscr{t}}^{(\mathcal{T})}$, omitting the superscript when the context makes it clear.

The Bottom *Up* Patterns

Given the aim of global model checking, where every configuration, not simply those reachable from the initial configuration, is considered, the behaviour of $\mathcal{A}$ at the stack bottom $\bot$ could no longer be assumed to be trivial. Hence, we should now introduce an additional kind of pattern to capture the transitions happening thereat.

The patterns are denoted $Up^{\bot}_{[\mathscr{q}]}$, for all $\mathscr{q} \in \mathcal{Q}$. It captures all the singleton pre-runs (which, can be effectively treated as just runs) that starts from the configuration $(\mathscr{q}, \{\varepsilon \mapsto \mathscr{m}_{init}\}, \varepsilon)$ while satisfying the k restriction.

Naturally, by applying a similar approach as in Section 4.2.4, it is straightforward to obtain the following decomposition equation for the bottom *Up pre-paths* patterns.

$$\mathsf{path}(\mathit{Up}^{\bot}_{[\mathscr{q}]}) = \biguplus_{\mathbb{D} \sim: \mathit{Up}^{\bot}_{[\mathscr{q}]}} \left\{ \mathsf{Sync}\big(\vec{\phi}_s, \{\gamma \mapsto \vec{\phi}_\gamma \mid \gamma \in \Gamma\}\big) \;\middle|\; \begin{array}{c} \vec{\phi}_s \in \mathit{SynPaths}^{\mathbb{D}}_{(\bot, D)} \\ \forall \gamma \in \Gamma. \vec{\phi}_\gamma \in \mathsf{path}(V^{\gamma}_{\mathbb{D}}) \end{array} \right\} \tag{5.7}$$

Likewise, their weights can be decomposed as follows, in accordance to the same rationale as that for the patterns introduced before.

$$\mathit{wt}(\mathit{Up}^{\bot}_{[\mathscr{q}]}) = \sum_{\mathbb{D} \sim: \mathit{Up}^{\bot}_{[\mathscr{q}]}} \mathit{wt}(\mathit{SynPaths}^{\mathbb{D}}_{(\gamma, D)}) \cdot \prod_{\gamma \in \Gamma} \mathit{wt}(\mathit{TripsOrUp}^{\gamma}_{\mathbb{D}(\gamma)}) \tag{5.8}$$

From here on, for exposition simplicity, we write simply $\mathit{TripsOrUp}^{\mathscr{g}}_{D}$ to range over all the patterns introduced including the bottom one. By definition then, if $\mathscr{g} = \bot$, it is implied that $|D| = 1$. Moreover, we may obtain the general weight decomposition equation as follows:

$$\mathit{wt}(\mathit{TripsOrUp}^{\mathscr{g}}_{D}) = \sum_{\mathbb{D} \sim: \mathit{TripsOrUp}^{\mathscr{g}}_{D}} \mathit{wt}(\mathit{SynPaths}^{\mathbb{D}}_{(\gamma, D)}) \cdot \prod_{\gamma \in \Gamma} \mathit{wt}(\mathit{TripsOrUp}^{\gamma}_{\mathbb{D}(\gamma)}) \tag{5.9}$$

[2] In fact, the reading tree is unique in our model-checking construction. Nonetheless, since this property will be confirmed only by the proof of our final target, we will assume multiplicity in our discussion below until the whole proof is established.

LTL Types and Observations

Then, we introduce the "LTL types", a concept drawn from the LTL model checking methodology for pPDA and RMC. The concept intuitively provides a way to uniquely describe the property of a run wrt. the given LTL formula to check, by deciding precisely the set of satisfying sub-formulas of the LTL formula of the run.

Types of Runs To begin with, we define the *sub-formula*s of φ inductively as follows:

$$\begin{aligned}
\mathsf{SubFml}(\mathsf{p}) &:= \{\mathsf{p}\},\\
\mathsf{SubFml}(\neg\varphi) &:= \{\neg\varphi\} \cup \mathsf{SubFml}(\varphi),\\
\mathsf{SubFml}(\varphi_1 \wedge \varphi_2) &:= \{\varphi_1 \wedge \varphi_2\} \cup \mathsf{SubFml}(\varphi_1) \cup \mathsf{SubFml}(\varphi_2),\\
\mathsf{SubFml}(\mathcal{X}\varphi) &:= \{\mathcal{X}\varphi\} \cup \mathsf{SubFml}(\varphi),\\
\mathsf{SubFml}(\varphi_1 \,\mathcal{U}\, \varphi_2) &:= \{\varphi_1 \,\mathcal{U}\, \varphi_2\} \cup \mathsf{SubFml}(\varphi_1) \cup \mathsf{SubFml}(\varphi_2).
\end{aligned}$$

Each *subset* of $\mathsf{SubFml}(\varphi)$ is called a ***type***. Intuively, for an infinite run $\mu \in \mathsf{InfTraj}_{\mathscr{M}(\mathcal{A})}$, the exact set of satisfying sub-formulas of φ is termed *the type of* μ. Formally, τ is called the type of μ iff: for every $\varphi' \in \mathsf{SubFml}(\varphi)$, $\varphi' \in \tau$ if and only if $\varphi' \models_\nu \mu$. It is not hard to observe that every run admits a unique type. We denote the set of types as $\mathbb{T}(\varphi)$ (or simply $\mathbb{T}$), i.e., $\mathbb{T}(\varphi) := 2^{\mathsf{SubFml}(\varphi)}$.

Back-Inference of Types Given a configuration $\mathscr{c}$ and a run μ with its type τ', we can uniquely determine (which can be proved straightforwardly) the type τ of the combined entity $\mathscr{c}\mu$ as follows. A sub-formula $\varphi' \in \tau$ iff:

- For $\varphi' = \mathsf{p}$, it holds that $\mathscr{c} \in \nu(\mathsf{p})$.
- For $\varphi' = \neg\varphi''$, φ'' must not be in τ.
- For $\varphi' = \varphi_1 \wedge \varphi_2$, both φ_1 and φ_2 are required to be in τ.
- For $\varphi' = \mathcal{X}\varphi''$, it is necessary that φ'' is in τ'.
- For $\varphi' = \varphi_1 \,\mathcal{U}\, \varphi_2$, φ_2 must be in τ, or both φ_1 must be in τ and φ_2 in τ'.

We define τ as the type *back-inferred* (or just inferred) from the *next type* τ' and the configuration $\mathscr{c}$.

The back-inference process can be applied to a sequence of configurations $\mathscr{c}_1, \mathscr{c}_2, \ldots, \mathscr{c}_n$ through sequential inference. Starting with τ' as the next type, we iteratively back-infer the type for each configuration, beginning with $\mathscr{c}_n$ and setting $\tau_n := \tau'$. At each step $\mathscr{c}_i$, we determine an intermediate type τ_i based on the back-tracking criteria, which then serves as the next type for $\mathscr{c}_{i-1}$. The process concludes with the type τ_0, which we refer to as the inferred type τ. We denote the above process by: $\tau \xleftarrow{\mathscr{c}_1,\ldots,\mathscr{c}_n} \tau'$.

A type τ is considered a *consistent type* of $\mathscr{c}$ if there is a next type τ' from which τ can be back-inferred.

φ-Observed Run Let $\mu = \mathscr{c}_1\mathscr{c}_2\ldots$ be an infinite run of configurations. The *φ-observed run* (or simply *observed run*) of μ is defined as the unique sequence:

$$(\mathscr{c}_1, \tau_1)(\mathscr{c}_2, \tau_2)\ldots$$

where for each index $i \geq 1$, τ_i is the LTL type of the suffix $\mathscr{c}_i\mathscr{c}_{i+1}\ldots$ of μ.

φ-Observed Interaction Sequence A *φ-observed interaction sequence* (or just observed interactions) $D^\star$ is a list of pairs of the form $(\mathscr{q}, \tau)$, where $\mathscr{q} \in \mathcal{Q}$ and τ is a type. We use the function $\mathsf{RmTy}(-)$ to remove all the types in an observed seqeuence to obtain the pure interaction seqeuence. The set of all observed interactions is denoted $\mathscr{D}^\star$.

Like in the patterns introduced above, the observed interaction sequences are also used to describe the "interface" of the interactions between a node and its parent node, this time additionally attached with the type at the corresponding location in the unique observed run.

Specifically, given a configuration $\mathscr{c}$ and a run μ starting from it, the *observed interactions at a node* referred to as the subject node, at the path x on $\mathsf{tree}(\mathscr{c})$ are denoted as $D^\star = (\mathscr{q}_1, \tau_1)\ldots(\mathscr{q}_n, \tau_n)$, where there are exactly n pairs of contiguous elements $((-,-,\rho),-)((\mathscr{q}_i,-,\rho'),\tau_i)$ in $\mu^\star$, the observed run of μ, from left to right such that either $\rho = x$ and $\rho' \sqsubseteq \rho$, denoting a *down* from the subject node to its parent node, or $\rho' = x$ and $\rho \sqsubseteq \rho'$, denoting an *up* visit to the subject node from its parent node.

Typing Sorts and Typing Skeletons

Given the above definition, it is noteworthy that a unique type is only definable for *infinite* runs. To adequately describe *finite* runs using type information, we employ the concept of *typing sort* (abbreviated as sort). Specifically, an *infinite* run admits a unique type while a *finite* run admits a unique sort.

A sort can be understood as a function that deduces starting types from possible ending types via the backward inference of a finite run. Specifically, a *finite* run μ possesses a sort $\mathscr{s}$, if and only if for any types τ and τ', $\mathscr{s}(\tau) = \tau'$ is true if and only if the backward inference $\tau' \overset{\mu}{\Longleftarrow} \tau$ is valid. Straightforward from this definition, we name τ to be the possible *next* type or just *input* type, and τ' as *source* type or simply, from the sense of a function, *return* type. Due to the uniqueness of backward inference, the sort associated with μ is uniquely determined and denoted by $\mathsf{sort}(\mu)$. Due to the simple valuation, given a status $(\mathscr{q}, \mathscr{m}, \mathscr{g})$, its sort can be uniquely identified, allowing the notation $\mathsf{sort}(\mathscr{q}, \mathscr{m}, \mathscr{g})$ to be used to denote $\mathsf{sort}([\mathscr{c}])$ where $\mathscr{c}$ satisfies $\mathsf{status}(\mathscr{c}) = (\mathscr{q}, \mathscr{m}, \mathscr{g})$.

Naturally from the definition, we may consider the *composition* of sorts — for two runs μ_1 and μ_2 such that there is a transition from the last configuration of μ_1 to the first configuration of μ_2, it is easily verifiable that the sort for the whole composed run $\mathsf{sort}(\mu_1 +\!\!+ \mu_2) = \mathsf{sort}(\mu_1) \circ \mathsf{sort}(\mu_2)$, where the operator $\circ$ is the usual function composition.

Following this, we introduce the concept of *typing skeletons* for categorising observed interaction sequences. Typing skeletons classify pre-runs into distinct

patterns based on their sorts and, potentially, their final type. These are referred to as skeletons because they provide a structural framework that can be filled with specific interaction sequences.

To delineate this concept, we first define a *sorted up-down interaction* as $q \overset{\mathfrak{s}}{\frown} q'$, which integrates sort information into the typical up-down interactions in sequences. A *typing skeleton* (abbreviated as skeleton) $D^{\triangle}$ is a sequence of up to k elements, divided into three segments: an optional *partial heading* of the form $\overset{\mathfrak{s}_0}{\frown} q_0$, a series of *middle interactions* comprising sorted up-down interactions, and an optional *final observed up* of the form (q_{n+1}, τ_{n+1}), where τ_{n+1} is termed the *final up type*. Thus, the structure of a skeleton is represented as:

$$[\overset{\mathfrak{s}_0}{\frown} q_0, q_1 \overset{\mathfrak{s}_1}{\frown} q_1', \ldots, q_n \overset{\mathfrak{s}_n}{\frown} q_n', (q_{n+1}, \tau_{n+1})] \tag{5.10}$$

where both the initial and final elements are optional.

The middle interactions, namely, the sub-list from $q_1 \overset{\mathfrak{s}_1}{\frown} q_1'$ to $q_n \overset{\mathfrak{s}_n}{\frown} q_n'$ in Equation (5.10), denote sorted trips occurring at or above the subject node, with the inclusion of sort information for each interaction. The partial heading indicates that the stack pointer is initially at or above the subject node, suggesting the first interaction is a *down* from the subject node with state q_0, and this initial transition until the first *down* possesses sort $\mathfrak{s}_0$. The final observed *up*, denoting infinite runs that do not descend from the subject node, contains no sort information, only the type.

Skeletons with a partial heading are termed *partial* skeletons, while others are referred to as *full* skeletons. Those with a final observed *up* are classified as *Up* skeletons; others are known as *Trips* skeletons. The set of all skeletons is denoted $\mathscr{D}^{\triangle}$.

Base Interaction Sequence The *base interaction sequence* is derived from a typing skeleton by excluding all sort information and any potentially existing final *up* type. This sequence is obtained using the function $\mathsf{base}(-)$, which extracts the base interaction sequence from the skeleton. Additionally, the function $\mathsf{RmTy}(-)$ is employed to strip the final *up* type from skeletons, while leaving the sort information intact.

Typing Instance The relationship between observed interaction sequences and typing skeletons is encapsulated through the concept of a *typing instance*. An observed interaction sequence $D^{\star}$ is said to be a typing instance of a skeleton $D^{\triangle}$, iff: $D^{\star}$ can be segmented into three sub-lists such that $D^{\star} = D_1^{\star} +\!\!+ D_2^{\star} +\!\!+ D_3^{\star}$, where: if the optional partial heading $\overset{\mathfrak{s}_0}{\frown} q_0$ exists in $D^{\triangle}$, then $D_1^{\star} = [(q_0, \tau_0)]$ comprises the first element of $D^{\star}$, with τ_0 being unrestricted; otherwise, $D_1^{\star}$ is ε. If the optional final observed *up* (q_{n+1}, τ_{n+1}) exists, then $D_3^{\star}$ strictly matches

$[(q_{n+1}, \tau_{n+1})]$; otherwise, $D_3^\star$ is ε. The middle interactions of $D^\triangle$ are delineated as:

$$[q_1 \overbrace{\quad}^{\vartheta_1} q'_1, \ldots, q_n \overbrace{\quad}^{\vartheta_n} q'_n]$$

and consequently, $D_2^\star$ takes the form:

$$[(q_1, \tau_1), (q'_1, \tau'_1), \ldots, (q_n, \tau_n), (q'_n, \tau'_n)]$$

where for each i, $\vartheta_i(\tau'_i) = \tau_i$. Especially, we define the instance of an empty skeleton ε to be the empty sequence ε.

Interaction Skeleton Given an infinite run μ starting from a configuration c, the *interaction skeleton* of a node, referred to as the subject node, on the tree $\mathsf{tree}(c)$ is the skeleton $D^\triangle$ such that the behaviours of μ occurring within and above this node conform to $D^\triangle$.

Specifically, let $D^\triangle$ be denoted as Equation (5.10) with an optional partial heading and an optional final *up*. If it includes the optional partial heading, then the run μ starts from a stack pointer within or above the subject node. The first finite run μ_0, as a prefix of μ, ends with the first visit to the parent node of the subject node. This run must occur with control state q_0, and the sort of μ_0 must be ϑ_0. Subsequently, there are exactly n up-down visits in μ, where the ith visit starts from the parent node of the subject node with control state q_i and ends with the first *down* back to the parent node again with control state q'_i, with the entire visit having sort ϑ_i. Finally, if the final *up* exists in $D^\triangle$, there must also be an infinite postfix of μ that starts with an *up* visit to the subject node with control state q_{n+1}, with the entire postfix being of type τ_{n+1}.

Skeleton Pattern With the skeletons introduced, we then extend the concept of patterns discussed so far by incorporating LTL information, thus introducing *skeleton patterns*. As the name suggests, a skeleton pattern substitutes the usual interaction sequence D with a *full* skeleton $D^\triangle$ — specifically, of the form $\mathit{TripsOrUp}^{q}_{D^\triangle}$. This concept intuitively serves as a *categorisation* of the pre-runs within the original pattern $\mathit{TripsOrUp}^{q}_{\mathsf{base}(D^\triangle)}$, categorising the pre-runs according to the sorts and types information.

Specifically, let $D_1^\triangle$ be expressed as in Equation (5.10) without the optional heading. A pre-run $\vec{\mu} = \mu_1 \ldots \mu_n \mu_{n+1}$ in $\mathit{Up}^{q}_{\mathsf{base}(D_1^\triangle)}$ is also in $\mathit{Up}^{q}_{D_1^\triangle}$ if, for all $1 \leq i \leq n$, μ_i has sort ϑ_i, and the final infinite run $\vec{\mu}_{n+1}$ has type τ_{n+1}. Similarly, let $D_2^\triangle$ be expressed as in Equation (5.10) without both the optional heading and the final *up*. Then, a pre-run $\vec{\mu} = \mu_1 \ldots \mu_n$ in $\mathit{Trips}^{\gamma}_{\mathsf{base}(D_2^\triangle)}$ is in $\mathit{Trips}^{\gamma}_{D_2^\triangle}$ if, for all i, μ_i has sort ϑ_i.

Upward Skeletons For a synchronisation path π, a given *extended* direction $\kappa \in \mathcal{K}^{\Uparrow}$, called the *source direction*, a function $f : \Gamma \to \mathscr{D}^\triangle$ is considered an *upward skeleton assigning function* (abbr. upward skeletons) that is *consistent* with the pair (π, κ) if: (1) the values of f are all *full* skeletons, with the sole exception when $\kappa \in \Gamma$, in which case $f(\kappa)$ is partial or empty sequence; (2) for all $\gamma \in \Gamma$, $\pi \restriction_\kappa \gamma = \mathsf{base}(f(\gamma))$.

Computing Downward Skeleton For a given synchronisation path and source direction pair (π, κ) and an upward skeletons function f consistent with it, the actual skeleton for the downward interaction at the subject node can be computed using the upward sort information provided by f through backward inference, denoted by $\mathsf{DownSkel}_\kappa(\pi, f)$.

In the definition, this function computes and composes the sorts and possibly the final type in a backward manner based on the information provided by f. Specifically, let $\pi = [\zeta_1 \dots \zeta_n]_{(\boldsymbol{m}_0, \mathscr{G})}$. We first *furnish* π with the information from f by inserting the corresponding sort and type information, such that:

1. At the beginning, if the source direction $\kappa \in \Gamma$, let the first sort in $f(\kappa)$ be $\mathscr{s}_0$. We then insert $\overset{\mathscr{s}_0}{\frown}$ before ζ_1.

2. For every contiguous pair of reaction units $\zeta_i\zeta_{i+1}$ in π, if the reaction direction of ζ_i is in Γ, let it be γ. We then insert the mth sort $\mathscr{s}$ in $f(\gamma)$ so that $\zeta_i \overset{\mathscr{s}}{\frown} \zeta_{i+1}$, where m is determined by the number of times the direction γ appears in the prefix up to ζ_i (when $\gamma = \kappa$, m should be 1 more than this number).

3. Finally, at the end ζ_n, if the reaction direction of ζ_n is not $\Downarrow$, let it be γ. We additionally attach the final *up* type of $f(\gamma)$, denoted by τ, so that it results in (ζ_n, τ).

Generally, this process yields a *furnished synchronisation path* of the following form, with optional $\mathscr{s}_0$ and τ:

$$\Big[\overset{\mathscr{s}_0}{\frown}\ \zeta_1\ \overset{\mathscr{s}_1}{\frown}\ \cdots\ \overset{\mathscr{s}_{i_1-1}}{\frown}\ \zeta_{i_1}, \zeta_{i_1+1}\ \overset{\zeta_{i_1+1}}{\frown}\ \cdots\ \overset{\mathscr{s}_{i_2-1}}{\frown}\ \zeta_{i_2}, \dots, \zeta_{i_m+1}\ \overset{\zeta_{i_m+1}}{\frown}\ \cdots\ \overset{\zeta_{n-1}}{\frown}\ (\zeta_n, \tau)\Big]_{(\boldsymbol{m}_0, \mathscr{G})} \tag{5.11}$$

where each i_j denotes the index of the jth occurrence of the *down* operation. By construction, there is no sort arc to attach between it and the next reaction unit. There might be no $\mathscr{s}_0$ if the source direction $\kappa = \Downarrow$; and the final element will simply be ζ_n if its reaction direction is $\Downarrow$.

Equipped with this furnished path, we may then present the definition of $\mathsf{DownSkel}_\kappa(\pi, f)$ by considering each comma-separated segment of Equation (5.11) and collecting the content produced in the same order as the furnished path. Let the sort of the ith reaction unit in π, denoted as $\mathsf{sort}(\pi[i])$, be defined as $\mathsf{sort}(\boldsymbol{q}_i, \boldsymbol{m}_{i-1}, \mathscr{G})$, where $\boldsymbol{q}_i$ is the source state of ζ_i and $\boldsymbol{m}_{i-1}$ is either the reaction memory of ζ_{i-1} if $i > 1$, or $\boldsymbol{m}_0$ if $i = 1$. Then, we have the following inspection for the segments:

1. If $\mathscr{s}_0$ does not exist, this segment produces $\boldsymbol{q} \overset{\mathscr{s}}{\frown} \boldsymbol{q}'$, where $\boldsymbol{q}$ is the source state of ζ_1, $\boldsymbol{q}'$ is the reaction state of ζ_{i_1}, and $\mathscr{s}$ is given by the straightforward

sort composition:

$$\mathsf{sort}(\pi[1]) \circ \mathscr{s}_1 \circ \cdots \circ \mathscr{s}_{i_1-1} \circ \mathsf{sort}(\pi[i_1])$$

Otherwise, when $\mathscr{s}_0$ exist, we also compute the $\mathscr{s}$ and $\mathscr{q}'$ as in the former case, and produce the result for this case as $\overbrace{\qquad}^{\mathscr{s}_0 \circ \mathscr{s}}\ \mathscr{q}'$.

2. For each middle segment from ζ_{i_j+1} to $\zeta_{i_{j+1}}$ of the furnished path, it produces $\mathscr{q}\ \overbrace{\qquad}^{\mathscr{s}}\ \mathscr{q}'$, where $\mathscr{q}$ is the source state of ζ_{i_j+1}, $\mathscr{q}'$ is the reaction state of $\zeta_{i_{j+1}}$, and the sort $\mathscr{s}$ is the straightforward composition of:

$$\mathsf{sort}(\pi[i_j + 1]) \circ \mathscr{s}_{i_j+1} \circ \cdots \circ \mathscr{s}_{i_{j+1}-1} \circ \mathsf{sort}(\pi[i_{j+1}])$$

3. For the final segment from ζ_{i_m+1} to ζ_n, when there is no final τ attached to ζ_n, like in the previous case, it produces $\mathscr{q}\ \overbrace{\qquad}^{\mathscr{s}}\ \mathscr{q}'$, where $\mathscr{q}$ is the source state of ζ_{i_m+1}, $\mathscr{q}'$ is the reaction state of ζ_n, and the sort $\mathscr{s}$ is the straightforward composition of:

$$\mathsf{sort}(\pi[i_m + 1]) \circ \mathscr{s}_{i_j+1} \circ \cdots \circ \mathscr{s}_{n-1} \circ \mathsf{sort}(\pi[n])$$

Otherwise, when the final τ exists, we still compute $\mathscr{q}$ and $\mathscr{s}$ as in the former case when τ is absent, but in this case, it produces $(\mathscr{q}, \mathscr{s}(\tau))$.

4. As a special case, if there is only one segment, and both $\mathscr{s}_0$ and τ exist, it indicates that there is no interaction with the lower level, hence the overall result of $\mathsf{DownSkel}_\kappa(\pi, f)$ is ε. Otherwise, if $\mathscr{s}_0$ is missing, handle the unique segment as in the third case above. Alternatively, if τ is missing, address it as in the first case above.

Computing Upward Typing Instances For a given synchronisation path and source direction pair (π, κ), an upward skeletons assignment f consistent with it, and a downward observed interactions sequence $D^\star$ that is an instance of $\mathsf{DownSkel}_\kappa(\pi, f)$, it is evident that for any upward direction $\gamma \in \Gamma$, the backward inference allows the deduction of the observed interactions $D^\star_\gamma$ as instances of $f(\gamma)$ according to π. This inference is formalised by $\mathsf{Instance}_\kappa(\pi, f, D^\star, \gamma)$.

The intuition behind this definition is straightforward — one may compute the types at each reaction unit in a backward manner. Specifically, let the *furnished synchronisation path* of π derived by incorporating the sorts and type information in f be depicted also in the form of Equation (5.11). We then further incorporate the information of $D^\star$ into this furnished path and compute backwardly the type before and after each reaction unit, ultimately forming the following path, referred

to as the *observed furnished path*:

$$\left[\begin{array}{l} \overbrace{\qquad}^{\mathscr{Y}_0}(\tau_1,\zeta_1,\tau_1')\overbrace{\qquad}^{\mathscr{Y}_1}\cdots\overbrace{\qquad}^{\mathscr{Y}_{i_1-1}}(\tau_{i_1},\zeta_{i_1},\tau_{i_1}'), \\ (\tau_{i_1+1},\zeta_{i_1+1},\tau_{i_1+1}')\overbrace{\qquad}^{\zeta_{i_1+1}}\cdots\overbrace{\qquad}^{\mathscr{Y}_{i_2-1}}(\tau_{i_2},\zeta_{i_2},\tau_{i_2}'),\ldots, \\ (\tau_{i_m+1},\zeta_{i_m+1},\tau_{i_m+1}')\overbrace{\qquad}^{\zeta_{i_m+1}}\cdots\overbrace{\qquad}^{\zeta_{n-1}}(\tau_n,\zeta_n,\tau_n') \end{array}\right]_{(m_0,\mathscr{g})} \tag{5.12}$$

where, for each segment, the typing procedure involves first consulting the corresponding type in $D^\star$ or the final *up* type in the skeleton of the source direction of f, and then backwardly typing each location by backward inference via the sorts. Specifically, the inference is given by:

1. Consider the first segment in the path. If $\mathscr{Y}_0$ exists, then let τ_{i_1}' be the first type in $D^\star$, and then each type to attach is derived by a consequential backward inference from right to left: $\tau_{i_1} = \mathsf{sort}(\zeta_{i_1})(\tau_{i_1}')$, $\tau_{i_1-1}' = \mathscr{Y}_{i_1-1}(\tau_{i_1})$ …etc., until $\tau_1' = \mathscr{Y}_1(\tau_2)$ and $\tau_1 = \mathsf{sort}(\zeta_1)(\tau_1')$.

   Otherwise, if $\mathscr{Y}_0$ does not exist, then let τ_{i_1}' be the second type in $D^\star$, and perform the above inference. Note that, in this case, τ_1 must match the first type in $D^\star$, as guaranteed by $D^\star$ being an instance of $\mathsf{DownSkel}_\kappa(\pi, f)$.

2. Then, for all middle segments, if $\mathscr{Y}_0$ exists, let τ_{i_j}' be the $(2j-1)$th type in $D^\star$; otherwise, let it be the $2j$th type in $D^\star$. The consequential backward inference follows as described above.

3. Finally, if the direction of ζ_n is $\Downarrow$, simply let τ_n' be the last type in $D^\star$; otherwise, let it be the τ as in the furnished path. The consequential backward inference remains the same as described above.

In this way, obtaining $\mathsf{Instance}_\kappa(\pi, f, D^\star, \gamma)$ involves traversing from left to right in the observed furnished path Equation (5.12) as follows:

- Each time we encounter a reaction unit ζ_i for $i < n$ with reaction direction γ, consider the reaction state $\mathscr{q}$, type $\tau := \tau_i'$, and state $\mathscr{q}'$ as the source state of ζ_{i+1} with $\tau' := \tau_{i+1}$. We collect the two pairs $(\mathscr{q}, \tau)$ and $(\mathscr{q}', \tau')$.

  Ultimately, after the traversal, it forms a list of pairs that constitute a proper observed interaction sequence $D_\gamma^\star$.

- If either $\mathscr{Y}_0$ does not exist or the source direction of f is not γ, the final result is simply the collected observed interaction sequence $D_\gamma^\star$. Otherwise, we additionally prepend the sequence with the pair $(\mathscr{q}, \tau)$, where $\mathscr{q}$ is the source state of ζ_1 and $\tau := \tau_1$ in Equation (5.12).

Computing Head Types In addition to upward typing instances, the starting type of π can also be computed using the information from (π, κ), f, and $D^\star$, and is denoted by $\mathsf{HdType}_\kappa(\pi, f, D^\star)$. The definition for this case is straightforward: by first forming an observed furnished path for π with the help of f and $D^\star$, as described in Equation (5.12), the final result is simply the type τ_1 therein.

Remark 5.13 (Partial Functions and Set Extension)**.** It is essential to note that the functions described are *partial*. When the parameters are not qualified, the result is assumed to be a special value of $\mathsf{Undefined}$. Extending these definitions to sets, we naturally disregard any $\mathsf{Undefined}$ results.

For instance, we may write $\mathsf{Instance}_\kappa(\Pi, F, D^\star, \gamma)$, where Π is a set of paths and F is a set of functions of type $\Gamma \to \mathscr{D}^\triangle$, to denote the set of elements resulting from $\mathsf{Instance}_\kappa(\pi, f, D^\star, \gamma)$ for each $\pi \in \Pi$ and $f \in F$, where the function $\mathsf{Instance}$ is defined (as opposed to being undefined) for these parameters.

5.2.2 Model Checking Overview

With these definitions in place, let us proceed to introduce the intuition behind the model checking procedure. We will first present the principles and intuition underlying the construction, followed by a motivating example that illustrates the idea with more concrete details.

Key Principles of the Model Checking Approach

Our proof proceeds by constructing the *configuration tree automaton* that accepts precisely the set of configuration trees, $\mathsf{tree}(\mathscr{c})$, which satisfy the given LTL property φ almost surely. The central idea is to *track the potential future interaction skeletons* at every node of the configuration tree. To clarify this concept, we make use of the notion of reading trees.

More specifically, for a given configuration $\mathscr{c}$, the goal is to construct a reading tree of $\mathsf{tree}(\mathscr{c})$ such that, at each node, the reading state captures information about *all the* ***probable*** *future interaction skeletons at that node.* The term *probable* is used, rather than *possible*, to indicate that the set of infinite runs conforming to the given skeleton at the node has a non-zero probability. This differs from the term "possible", which only suggests the existence of at least one run passing through the skeleton.

It is important to highlight that this tracking involves only *observation* without any *qualification.* If we were to rely solely on this information, every configuration tree would be accepted, as the information would merely be recorded. Our objective, however, is to accept only those configurations that almost surely satisfy the given LTL formula, while rejecting all those not.

To achieve this, in addition to tracking all interaction skeletons, we also need to gather the set of probable observed interaction sequences at each node. These sequences allow us to maintain the typing information of the actual runs. A probable observed interaction sequence of a node refers to the those sequences $D^\star$ for which the set of runs having $D^\star$ as the observed interactions of the node

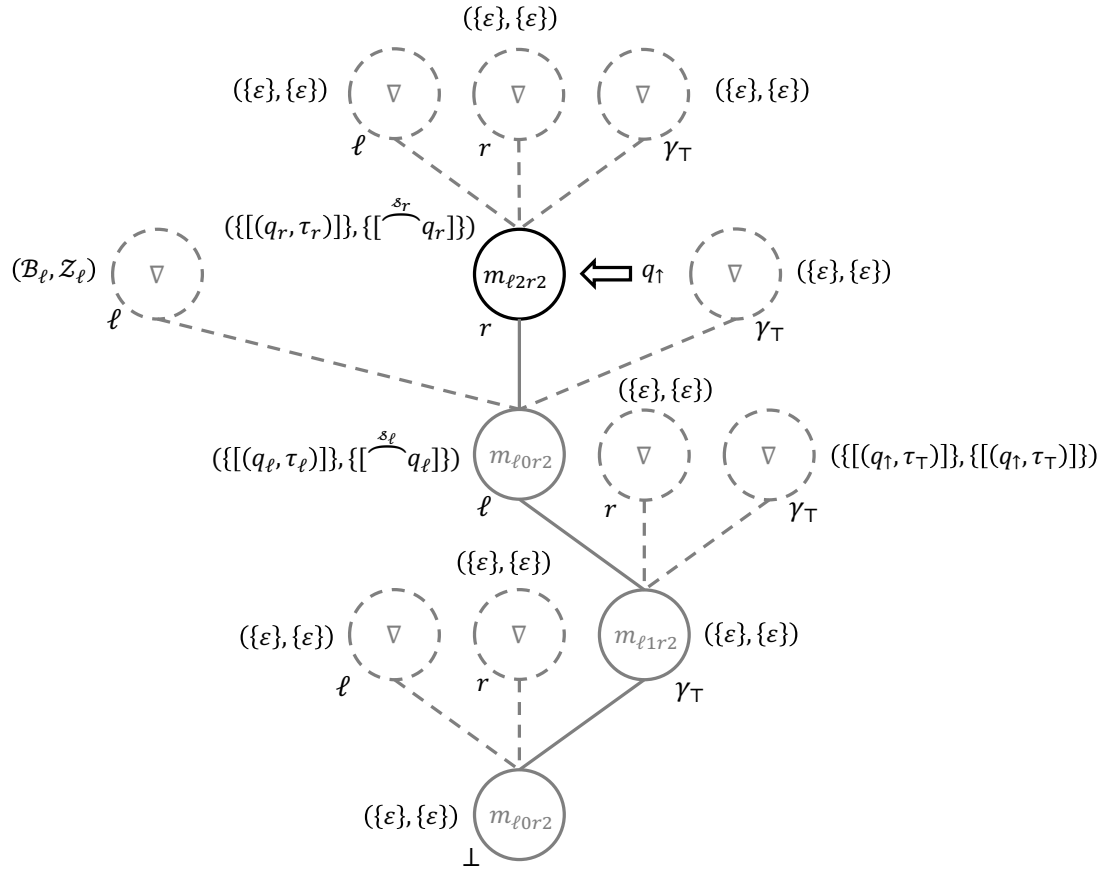

Figure 5.2: $\mathcal{C}_b$ and Its $(\mathcal{B}, \mathcal{Z})$ Pairs

has a probability greater than zero. Notably, each such sequence corresponds to a typing instance of a probable interaction skeleton.

With complete type information for a node and its child nodes, we can locally determine whether the element (q, m) can be accepted at the node by evaluating all probable *head types* across synchronisation paths. Intuitively, acceptance of the configuration occurs if and only if all types of probable runs satisfy φ.

In summary, to fulfil these requirements, the states of the configuration tree automaton must consist of two components: a set $\mathcal{Z}$ representing all probable interaction skeletons of a node, and a set $\mathcal{B}$ representing all probable observed interaction sequences as typing instances of $\mathcal{Z}$. The actual states of the automaton will either be of the form $\Downarrow^{\gamma}_{(\mathcal{B},\mathcal{Z})}$ for side-branch states, or $\Uparrow^{q}_{(\mathcal{B},\mathcal{Z})}$ for main-branch states. Besides the properties $\mathcal{B}$ and $\mathcal{Z}$, these states also encode the stack status of the node being read, while indicating whether the node is part of the main branch or side branches. We range over both types by $\Updownarrow^{q}_{(\mathcal{B},\mathcal{Z})}$.

Motivating Example

Let us now consider a concrete example to illustrate the principles discussed.

$$
\begin{array}{rcl}
(q_\uparrow, m_{\ell NrM}, \bot) & \to & (q_\uparrow, m_{\ell NrM}, up_\ell) \\
(q_X, m_{\ell NrM}, \bot) & \to & (q_\uparrow, m_{\ell NrM}, up_{\gamma_\top}) \\
(q_Y, m_{\ell NrM}, \gamma_\top) & \to & (q_\uparrow, m_{\ell NrM}, up_{\gamma_\top})
\end{array}
$$

$$
(q_Y, m_{\ell NrM}, X) \xrightarrow{p_{NM}} \begin{cases} (q_\uparrow, m_{\ell(N+1)rM}, up_\ell) & N < 2 \\ (q_\uparrow, m_{\ell Nr(M+1)}, up_r) & M < 2 \\ (q_X, m_{\ell NrM}, down) & \end{cases}
$$

$$
\begin{aligned}
* &: X \text{ stands for either } \ell \text{ or } r. \\
&\; Y \text{ ranges over } \ell,\ r, \text{ or } \uparrow. \\
&\; N \text{ and } M \text{ are all } \geq 0 \text{ and } \leq 2. \\
&\; p_{NM} := \frac{1}{\mathbf{1}[N < 2] + \mathbf{1}[M < 2] + 1}
\end{aligned}
$$

Figure 5.3: Rules of $\mathcal{A}_b$

Example 5.14. Consider a 2-PTSA $\mathcal{A}_b$ that randomly constructs and traverses a binary tree, with rules depicted in Figure 5.3. Let the initial control state q_{init} be $q_\uparrow$, and the initial memory state m_{init} be $m_{\ell 0 r 0}$. Note that the rules are total.

Intuitively, the control state tracks whether control originates from below ($\uparrow$) or from a direction (ℓ or r) above. The local memory, represented as $m_{\ell NrM}$ (with N and M ranging from 0 to 2), records the number of visits to each branch. Specifically, N indicates the number of visits to the left branch (ℓ), while M tracks visits to the right branch (r). During traversal, probabilistic outcomes include visiting a branch that has not been visited twice or transitioning *down* to the parent.

The traversal concludes after all branches are visited, but since no termination rule is allowed, a special stack symbol, $\gamma_\top$, is introduced to represent termination with infinite *up* transitions.

It is essential to emphasise that the rules are total, a property particularly important in global model checking. In global model checking, unlike local checking that focuses on rules reachable from initial runs, configurations considered may not be reachable from the initial configuration. This means that the local memories might not accurately reflect visitation information. For example, a node with memory $m_{\ell 2 r 2}$ could be a descendant of a node with m_{init}, and may have generated nodes in the upward direction, terminating in $\gamma_\top$.

Now consider the property $\varphi_b = \neg \mathcal{X} m_{\ell 2 r 2} \mathcal{U} (q_r, m_{\ell 0 r 2})$. For simplicity, we directly represent the atomic proposition using the control state and local memory, providing clear valuation information. This makes the meaning of the property straightforward and self-explanatory.

Assume that we have constructed the configuration tree automaton $\mathcal{T}_b$ for the purpose of global model checking $\mathcal{A}_b$ against φ_b, following the established principles. We now investigate whether a particular configuration c of $\mathcal{A}_b$ is accepted by $\mathcal{T}_b$.

The configuration, including skeletons and observed interactions for each node, is depicted in Figure 5.2. It is important to note that this configuration exemplifies configurations that are not reachable from the initial configuration — its local memories do not accurately reflect visitation data for each node.

For convenience, we break down φ_b into a parsing tree, as shown in Figure 5.4. We represent the type by a five-element list of 0s and 1s, where each element indicates whether a subformula is included in the type. For example, $[1, 0, 0, 1, 0]$ represents $\{\boldsymbol{m}_{\ell 2r2}, \neg\mathcal{X}\boldsymbol{m}_{\ell 2r2}\}$. This notation follows the approach in [111].

Next, we explain the reasoning behind the reading states, specifically why these $(\mathcal{B}, \mathcal{Z})$ are correct as per the principle.

We begin by considering the possible types of visits to $\gamma_\top\ell\ell$, which represent the most challenging part of the construction, referred to here as the "subject node". Examining the whole configuration, after getting back to the node at $\gamma_\top\ell$, there might be at most two visits to this node. Hence, the possible visit sequences are $[q_\uparrow, q_\ell]$, $[q_\uparrow, q_\ell, q_\uparrow, q_\ell]$, and the empty sequence ε. This set corresponds to $\mathsf{base}(\mathcal{Z}_\ell)$.

For the first visit, we examine two cases: an *up* move with $q_\uparrow$ and a *down* move with q_ℓ. The sorts for this visit could be:

- $\mathcal{s}_1$: returns $[0, 0, x, \overline{x}, y]$, where, for an input type $[x', -, -, -, y']$, $x = x'$, $\overline{x}$ is the Boolean negation of x, and y is 0 if $x' = 1$, otherwise y'. This sort represents the run that immediately returns *down* to the parent node at $\gamma_\top\ell$, thus the operator directly relates to the input type.

- $\mathcal{s}_2$: always returns $[0, 0, 0, 1, 0]$, representing runs that reach local memory $\boldsymbol{m}_{\ell 2r2}$ before returning to the parent node. Note that the third position is always 0, as the node has just been created, and thus any upward visit will not involve $\boldsymbol{m}_{\ell 2r2}$ — here we focus solely on upward visits for this case and subsequent cases, since the immediate *down* has already been addressed in the first case.

- $\mathcal{s}_3$: returns $[0, 0, 0, 1, 1]$, indicating runs that reach $(q_r, \boldsymbol{m}_{\ell 0r2})$ before the next position reaches $\boldsymbol{m}_{\ell 2r2}$.

- $\mathcal{s}_4$: returns $[0, 0, 0, 1, x]$, where, given the input type $[y, -, -, -, z]$, x is 0 if $y = 1$, otherwise x is z. This implies that the runs neither reach $(q_r, \boldsymbol{m}_{\ell 0r2})$ nor $\boldsymbol{m}_{\ell 2r2}$, and they do not receive an immediate *down* at the start.

We now examine the second visit, which involves an additional *up* move with $q_\uparrow$ and a *down* move with q_ℓ. The possible outcomes for this visit include the following: Let the input type be denoted as $[x', -, -, -, y']$.

- $\mathcal{s}_1'$: returns $[1, 0, x, \overline{x}, y]$, where $x = x'$ and $y = 0$ if $x' = 1$; otherwise, $y = y'$. This indicates that the local memory at the node is $\boldsymbol{m}_{\ell 2r2}$, and hence an immediate *down* move must follow.

- $\mathcal{I}'_2$: similar to $\mathcal{I}'_1$, it signifies an instant *down*, except the local memory at the node is not $\mathcal{m}_{\ell 2r2}$. It returns $[0, 0, x, \overline{x}, y]$, where $x = x'$ and $y = \neg x' \wedge y'$.

  Note that in this case, and for the cases below, the node never contains local memory $\mathcal{m}_{\ell 2r2}$, as such cases would necessitate an instant *down*, which is already handled by $\mathcal{I}'_1$.

  On the other hand, the focus of the following cases is only on beginning with an upward visit, since instant *down* moves have already been addressed in $\mathcal{I}'_1$ and this case.

- $\mathcal{I}'_3$: always returns $[0, 0, 1, 0, 0]$, indicating that an upward visit will instantly involve $\mathcal{m}_{\ell 2r2}$. Notably, the second position is always 0, as the control state must be $q_\uparrow$.

- $\mathcal{I}'_4$: always returns $[0, 0, 0, 1, 0]$, and $\mathcal{I}'_5$ returns $[0, 0, 0, 1, 1]$, both mirroring $\mathcal{I}_2$ and $\mathcal{I}_3$ from the first visit.

- $\mathcal{I}'_6$: returns $[0, 0, 0, 1, x]$, where $x = \neg x' \wedge y'$, similarly to $\mathcal{I}_4$ from the first visit.

We now examine the combinations of these cases to form possible skeletons. Almost all skeletons of the form $[q_\uparrow \overbrace{\quad}^{\mathcal{I}_i} q_\ell, q_\uparrow \overbrace{\quad}^{\mathcal{I}'_j} q_\ell]$ are valid for any combination of $\mathcal{I}_i$ and $\mathcal{I}'_j$, except that, certain combinations are impossible:

- $\mathcal{I}_1$ and $\mathcal{I}'_1$ — an instant *down* in the first visit prevents local memory $\mathcal{m}_{\ell 2r2}$ from being involved in the second;

- $\mathcal{I}_1$ and $\mathcal{I}'_3$ — the second visit cannot reach $\mathcal{m}_{\ell 2r2}$ if the first visit has an instant *down*;

- $\mathcal{I}_4$ and $\mathcal{I}'_1$ — $\mathcal{I}_4$ disallows reaching $\mathcal{m}_{\ell 2r2}$, conflicting with $\mathcal{I}'_1$;

- $\mathcal{I}_4$ and $\mathcal{I}'_3$ — $\mathcal{I}_4$ assumes no $\mathcal{m}_{\ell 2r2}$ on the whole sub-tree above this branche.

Notably, $\mathcal{I}_3$ and $\mathcal{I}'_1$ can be combined, as the run may first reach $(q_r, \mathcal{m}_{\ell 0r2})$ before the subject node holds $\mathcal{m}_{\ell 2r2}$.

Thus, the complex set $\mathcal{Z}_\ell$ comprises the following skeletons: (1) $[q_\uparrow \overbrace{\quad}^{\mathcal{I}_i} q_\ell]$ for all $i \in \{1, \ldots, 4\}$; (2) $[q_\uparrow \overbrace{\quad}^{\mathcal{I}_i} q_\ell, q_\uparrow \overbrace{\quad}^{\mathcal{I}'_j} q_\ell]$ for combinations of $i \in \{1, \ldots, 4\}$ and $j \in \{1, \ldots, 6\}$, excluding combinations where $i = 1$ or 4 and $j = 1$ or 3; and, additionally, (3) the empty skeleton ε.

Next, let us determine the appropriate form of $\mathcal{B}_\ell$. To perform backward inference, we first need to establish that $\tau_\ell = [0, 0, 0, 1, 0]$. This is because, the run from the node $[\gamma_\top]$ will no longer reach either $\mathcal{m}_{\ell 2r2}$ or $(q_r, \mathcal{m}_{\ell 0r2})$, as it simply consists of an infinite sequence of *up* transitions towards $\gamma_\top$.

In addition to examining the final type, we must also consider the synchronisation paths to carry out the inference, essentially following the approach introduced in the Instance computation above. It is evident that the available paths at the node $\gamma_\top \ell$ consists of:

$$[(q_r, q_\ell, m_{\ell 0 r 2}, down)]_{(m_{\ell 0 r 2}, \ell)} \tag{5.15}$$

$$[(q_r, q_\uparrow, m_{\ell 1 r 2}, up_\ell), (q_\ell, q_\ell, m_{\ell 1 r 2}, down)]_{(m_{\ell 0 r 2}, \ell)} \tag{5.16}$$

$$[(q_r, q_\uparrow, m_{\ell 1 r 2}, up_\ell), (q_r, q_\uparrow, m_{\ell 2 r 2}, up_\ell), (q_\ell, q_\ell, m_{\ell 2 r 2}, down)]_{(m_{\ell 0 r 2}, \ell)} \tag{5.17}$$

These paths describe the presence of no up_ℓ, one up_ℓ, and two up_ℓ respectively.

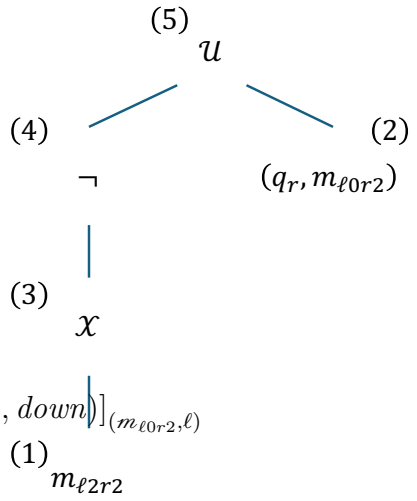

Figure 5.4: Parsing Tree of φ_b

We now deduce $\mathcal{B}_\ell$ from these paths, making use of $\mathcal{Z}_\ell$ and the final type $\tau_\ell = [0, 0, 0, 1, 0]$. First, let us consider the scenario where there is only one visit to ℓ, represented by the path in Equation (5.16). In this case, using backward inference from τ_ℓ with the sort of the second reaction unit in this path, we find that the type after returning from ℓ remains $[0, 0, 0, 1, 0]$. When this type is incorporated into the skeletons and applied to the sorts $\mathcal{s}_i$, we can deduce that the possible source types before executing the single up are either $[0, 0, 0, 1, 0]$ or $[0, 0, 0, 1, 1]$.

Then, we consider the case when there are two visits to ℓ, represented by the path in Equation (5.17). By the backward inference from τ_ℓ, the type after the two visits to ℓ should be $[1, 0, 0, 1, 0]$. Putting this type into $\mathcal{s}'_j$, the resulting source types at the beginning of the second visit are: $[1, 0, 1, 0, 0]$ (from $\mathcal{s}'_1$), $[0, 0, 1, 0, 0]$ (from $\mathcal{s}'_2$ and $\mathcal{s}'_3$), $[0, 0, 0, 1, 0]$ (from $\mathcal{s}'_4$ and $\mathcal{s}'_6$), and $[0, 0, 0, 1, 1]$ (from $\mathcal{s}'_5$). By backward inference from these types with the sort of the first reaction unit, namely $\mathsf{sort}(q_r, m_{\ell 1 r 2}, \ell)$, which, given the input type $[x, -, -, -, y]$ returns $[0, 0, x, \overline{x}, \neg x \wedge y]$. We then obtain the types after the first visit to be: $[0, 0, 1, 0, 0]$ (from $\mathcal{s}'_1$), $[0, 0, 0, 1, 0]$ (from $\mathcal{s}'_2$, $\mathcal{s}'_3$, $\mathcal{s}'_4$ and $\mathcal{s}'_6$) or $[0, 0, 0, 1, 1]$ (from $\mathcal{s}'_5$). Hence, by putting these types in $\mathcal{s}_i$ for $i \in \{1, \ldots, 4\}$, we obtain the source types at the beginning of the first visiting to be: $[0, 0, 0, 1, 0]$ and $[0, 0, 0, 1, 1]$. Either one can work in combination with all input types.

In summary, $\mathcal{B}_\ell$ contains the following configurations (with X ranging over $\{0, 1\}$):

$$[] \tag{5.18}$$

$$[(q_\uparrow, [0, 0, 0, 1, X]), (q_\ell, [0, 0, 0, 1, 0])] \tag{5.19}$$

$$[(q_\uparrow, [0, 0, 0, 1, X]), (q_\ell, [0, 0, 1, 0, 0]), (q_\uparrow, [1, 0, 1, 0, 0]), (q_\ell, [1, 0, 0, 1, 0])] \tag{5.20}$$

$$[(q_\uparrow, [0, 0, 0, 1, X]), (q_\ell, [0, 0, 0, 1, 0]), (q_\uparrow, [0, 0, 1, 0, 0]), (q_\ell, [1, 0, 0, 1, 0])] \tag{5.21}$$

$$[(q_\uparrow, [0, 0, 0, 1, X]), (q_\ell, [0, 0, 0, 1, 0]), (q_\uparrow, [0, 0, 0, 1, 0]), (q_\ell, [1, 0, 0, 1, 0])] \tag{5.22}$$

$$[(q_\uparrow, [0, 0, 0, 1, X]), (q_\ell, [0, 0, 0, 1, 1]), (q_\uparrow, [0, 0, 0, 1, 1]), (q_\ell, [1, 0, 0, 1, 0])] \tag{5.23}$$

The empty sequence at the beginning corresponds to the empty skeleton. Equation (5.19) corresponds to the case where there is only a single *up* to ℓ, while Equation (5.20) represents the second visit via $\mathfrak{s}'_1$, and Equation (5.21) corresponds to $\mathfrak{s}'_3$. The remaining expressions handle the other possible scenarios.

Having dealt with the most complex case for $\mathcal{B}_\ell$ and $\mathcal{Z}_\ell$, we now examine the remaining skeletons and types in the configurations. The type τ_r is clearly $[0,1,0,1,1]$, as, in this configuration, the return to the node must occur with $(q_\uparrow, \boldsymbol{m}_{\ell 0 r 2})$, and the subsequent visit will never be $\boldsymbol{m}_{\ell 2 r 2}$. The function $\mathfrak{s}_r$ returns $[1,0,x,\overline{x},y]$, where, for a given input type $[x',-,-,-,y']$, $x = x'$ and $y = \neg x' \wedge y'$, which is precisely the sort of the current status.

On the other hand, $\mathfrak{s}_\ell$ always returns $[1,0,0,1,1]$. Recall that this output reflects the sequence of operations from the current status until the node at $[\gamma_\top]$ is revisited. Thus, the first and second positions are determined by the current status, since the next operation must be a *down*. Given the local memory of the node at $\gamma_\top \ell$, the third position is similarly fixed. As for the fourth position, starting from the current status, the next operation is a *down* transition to the node at $\gamma_\top \ell$ with control state q_r, immediately satisfying $(q_r, \boldsymbol{m}_{\ell 0 r 2})$, which guarantees the fifth position holds true. Consequently, regardless of the next type, the sort function $\mathfrak{s}_\ell$ always returns the source type $[1,0,0,1,1]$.

In conclusion, we determine that this configuration almost surely satisfies φ_b. Based on the local $\mathcal{B}$ information, which confirms that the next type must be τ_r, by applying this to the sort function $\mathfrak{s}_r$ immediately yields $[1,0,0,1,1]$, implying that all probable run types are $[1,0,0,1,1]$. Thus, φ_b is satisfied almost surely.

Additionally, note that the rules in $\mathcal{T}_b$ that is applied to $\mathcal{A}_b$ can be straightforwardly recovered from the figure. For instance, the rule applied to the node at $\tau_\top \ell$ is:

$$\Uparrow^{\ell}_{(\{[(q_\ell,\tau_\ell)]\},\{[\overbrace{\quad}^{\mathfrak{s}_\ell} q_\ell]\})} \xrightarrow{\boldsymbol{m}_{\ell 0 r 2}} \left\{ \begin{array}{rcl} \ell & \mapsto & \Downarrow^{\ell}_{(\mathcal{B}_\ell,\mathcal{Z}_\ell)}, \\ r & \mapsto & \Uparrow^{r}_{(\{[(q_r,\tau_r)]\},[\overbrace{\quad}^{\mathfrak{s}_r} q_r])}, \\ \gamma_\top & \mapsto & \Downarrow^{\gamma_\top}_{(\{\varepsilon\},\{\varepsilon\})} \end{array} \right\} \tag{5.24}$$

5.2.3 Model Checking Procedure

We are now poised to present the model-checking procedure. To begin, we introduce a fundamental concept that is crucial for both the construction of the configuration tree automaton and the proof: the notion of *local totality*. While this concept may be both conceptually and technically intricate, understanding it is vital for comprehending the overall model checking procedure.

After introducing this concept, we will outline the complete construction of the configuration tree automaton. This will be followed by a detailed example that demonstrates the essential ideas of the formal construction through specific, illustrative details.

Introducing Local Totality: A Key Concept

By the motivating example provided in the overview, the overall principle of the construction should now be clearer. However, the technical construction might

still be more complex than one might have considered.

One key technical challenge has been to "connect" or "chain" the TA states within a rule. To formalise this sense of connection, we define the set of *chaining synchronisation paths* (abbr. the chaining set) for a given TA-rule-like triple r:

$$\left(\Updownarrow^{q}_{(\mathcal{B},\mathcal{Z})}, x, \{\gamma \mapsto \Updownarrow^{\gamma}_{(\mathcal{B}_\gamma,\mathcal{Z}_\gamma)} | \ \gamma \in \Gamma\} \right) \tag{5.25}$$

where x is either a pair (q, m) in $\mathcal{Q} \times \mathcal{M}$ or simply $m \in \mathcal{M}$. Suppose this is a rule to be applied to a node just like the rules of $\mathcal{T}_b$ applying to c_b, this notion of a chaining set of it corresponds straightforwardly to what one would expect — the set simply captures all the possible synchronisation paths that exist between the given interaction skeletons in $\mathcal{Z}$ of this node and the upper skeletons in $\mathcal{Z}_\gamma$ for each γ.

The chaining set Chains_r is defined as the union of all $\mathit{SynPaths}^{(\kappa,\mathbb{D})}_{(x,q,D)}$ across all $D \in \mathsf{base}(\mathcal{Z})$ and all $\mathbb{D}$ such that $\mathbb{D}(\gamma) \in \mathsf{base}(\mathcal{Z}_\gamma)$. The value of κ is determined by the following conditions:

- **Side-branch state**: If the LHS state $\Updownarrow^{q}_{(\mathcal{B},\mathcal{Z})}$ is a side-branch state $\Downarrow^{q}_{(\mathcal{B},\mathcal{Z})}$, it indicates that the source of the first control state arriving at this node must be from below. Thus, set $\kappa = \Downarrow$.
- **Main-branch state with** $x \in \mathcal{M}$: If the LHS state is a main-branch state and $x \in \mathcal{M}$, then there must be a direction γ_0 such that the state $\Updownarrow^{\gamma_0}_{(\mathcal{B}_{\gamma_0},\mathcal{Z}_{\gamma_0})}$ is also a main-branch state. This scenario indicates that the stack pointer is above this node, and γ_0 corresponds to this branch. Therefore, the source direction, from which the control state originates, must be γ_0. Hence, setting $\kappa = \gamma_0$.
- **Main-branch state with** $x \in \mathcal{Q} \times \mathcal{M}$: If the LHS state is a main-branch state $\Uparrow^{q}_{(\mathcal{B},\mathcal{Z})}$ and $x \in \mathcal{Q} \times \mathcal{M}$, this indicates that the stack pointer is at the current node, so it should not be coming from any direction. Specifically, we set $\kappa = \Uparrow$. Recall that in the definition of filtration, $\restriction_{\Uparrow}$ precisely refers to this situation.

Note that these three cases cover all possible scenarios when a rule is to apply to an internal node on the configuration tree. We also call this κ the *source direction* of this triple r. Addtionally, we call a function f to be an *upward skeleton context* of this triple, if $f(\gamma) \in \mathcal{Z}_\gamma$ for every γ with the set of all these functions denoted $\mathsf{UpSkels}_r$

This definition of chaining paths, as the name suggests, encompasses the set of synchronisation paths that connect the lower and upper interfaces specified by $\mathsf{base}(\mathcal{Z})$ and $\mathsf{base}(\mathcal{Z}_\gamma)$ for all γ.

With this definition in place, recall that our goal is to construct, for every configuration c, a reading tree for $\mathsf{tree}(c)$, where each node is associated with a corresponding reading state that accurately captures the likely observed interactions and probable interaction skeletons. From this intuition, it is evident that, for a rule r, expressed as Equation (5.25), to apply to a node of $\mathsf{tree}(c)$, a *necessary* condition is that the set of chaining paths must precisely match *all the possible synchronisation*

paths that could occur across all runs in $\mathbf{Run}_{inf}(\mathscr{c})$, referred to as the *possible future paths*. Recall that $\mathbf{Run}_{inf}(\mathscr{c})$ denotes all the infinite runs starting from $\mathscr{c}$.

Intuitively, the notion of a chaining set denotes all the possible synchronisation paths *syntactically* from the presented triple, disregarding the actual *semantics*, where $\mathcal{Z}$ and those $\mathcal{Z}_\gamma$ are intended to describe *all* the interaction skeletons of a node and its child nodes. In simpler terms, the concept of a chaining set first describes what the set of paths *is*, while the notion of all possible future paths, on the other hand, describes what it *should be*. Hence, the condition above essentially states that the actual syntactically determined path set should match what it *principally* should be, and the judgement is therefore *not* a tautology. Specifically, a triple such as in Equation (5.25) will *never* be a proper rule if its chaining set does not satisfy the condition.

This condition is straightforward to verify, given the configuration $\mathscr{c}$ itself. Consider the following two examples.

Example 5.26. Referring back to Example 5.14. From Figure 5.2, one may recover the rule applied to the node at $[\gamma_\top]$ straightforwardly be:

$$\Uparrow^{\gamma_\top}_{(\{\varepsilon\},\{\varepsilon\})} \xrightarrow{m_{\ell 1 r 2}} \left\{ \begin{array}{rcl} \ell & \mapsto & \Uparrow^{\ell}_{(\{[(q_\ell,\tau_\ell)]\},\{[\overbrace{\qquad}^{\vartheta_\ell} q_\ell]\})}, \\ r & \mapsto & \Downarrow^{r}_{(\{\varepsilon\},\{\varepsilon\})}, \\ \gamma_\top & \mapsto & \Downarrow^{\gamma_\top}_{\left(\{[(q_\uparrow,\tau_\top)]\},\{[(q_\uparrow,\tau_\top)]\}\right)} \end{array} \right\} \tag{5.27}$$

The set of all possible future paths for this node in the configuration and the chaining synchronisation paths set of this rule could be trivially seen to coincide as a singleton set comprising:

$$\{[(q_\ell, q_\uparrow, m_{\ell 1 r 2}, up_{\gamma_\top})]_{(m_{\ell 1 r 2}, \gamma_\top)}\} \tag{5.28}$$

Example 5.29. Consider a more intricate example. From the motivating example, recall that the rule applied at the node $\gamma_\top \ell$ is the one given in Equation (5.24). Now, by examining the three cases of interactions in $\mathsf{base}(\mathcal{Z}_\ell)$, we define three upward functions: $\mathbb{D}_1(\ell) = \varepsilon$, $\mathbb{D}_2(\ell) = [q_\uparrow, q_\ell]$, and $\mathbb{D}_3(\ell) = [q_\uparrow, q_\ell, q_\uparrow, q_\ell]$. In each case, the value of r is $[q_r]$, while for $\gamma_\top$, the value is ε.

Consequently, the chaining set, according to the above construction, is the union of: $\uplus_{i=1}^{3} \mathit{SynPaths}^{(r,\mathbb{D}_i)}_{(m_{\ell 0 r 2}, \ell, [q_\ell])}$. This yields exactly the three possible future paths, as described in Equations (5.15) to (5.17). Specifically, the path in Equation (5.15), which bypasses the ℓ branch, is generated by utilising $\mathbb{D}_1$. Similarly, the single visit path in Equation (5.16) is produced from $\mathbb{D}_2$, while the double visit path in Equation (5.17) is from $\mathbb{D}_3$.

This condition is, technically, one of the essential conditions that must be examined when constructing the configuration tree automaton. However, confirming the chaining set being exactly all possible future paths during the construction process is challenging, as we cannot obtain information about the overall configuration when talking about a single rule.

Therefore, we require explicitly testable criteria to either accept or reject a given set of synchronisation paths, checking whether it constitutes a complete set of all

possible future paths. This check must utilise only the *local* information of the lower and upper skeletons, e.g., the $\mathcal{Z}$ and $\mathcal{Z}_\gamma$ for all γ as in Equation (5.25). To achieve this, we define the concept of *local totality*, which is technically framed as a series of *necessary* conditions for a set to exactly all the possible future paths of the node this rule is applying to. The sufficiency of these conditions in formalising the possible future paths, however, is not immediately clear and will follow as a corollary of the correctness of the overall construction, as will be demonstrated in Lemma 5.47.

Formally, consider the triple r as in Equation (5.25), let its source direction be κ, and let $\mathcal{Z}_\Downarrow := \mathcal{Z}$ for convenience, we say the chaining set $\Pi = \mathsf{Chains}_r$ of r is *locally total*, if the following criteria hold:

Rule Totality For any synchronisation path π in Π and any paths $\boxed{\pi_1}$ and π_2 such that $\pi_1 +\!\!+ \pi_2 = \pi$, with π_2 being non-empty and assuming $\pi_2 = [\zeta, \dots]_{(m,\mathscr{g})}$, let the starting control state of ζ be denoted $\mathscr{q}$, then for any rule $(\mathscr{q}, m, \mathscr{g}) \xrightarrow{p} op$, there must be at least one path $\pi_1 +\!\!+ \pi_2'$ in Π for some π_2', where π_2' starts with an observed reaction unit ζ', having the initial control state $\mathscr{q}$ and operation $op(\zeta') = op$.

This property captures the intuition that every decision made within this node must be represented in the set of paths Π. If this property is violated, it indicates that there is at least one reachable operation that is not included, meaning Π does not capture *all* possible paths in this node.

Mention Totality For all directions κ', and for all elements $D^\triangle \in \mathcal{Z}_{\kappa'}$, there exists a synchronisation path $\pi \in \Pi$ such that $\pi \restriction_\kappa \kappa' = \mathsf{base}(D^\triangle)$.

This property ensures that every element in the set of possible skeletons for each direction κ' is visited by some path in Π. If this property is not satisfied, it would mean that some skeletons are never visited, violating the intuition that *all possible skeletons* should be explored.

Continuation Totality For all synchronisation paths π (not restricted in Π) such that for all $\kappa' \in \mathcal{K}$, $\pi \restriction_\kappa \kappa'$ is a prefix of some element in $\mathsf{base}(\mathcal{Z}_{\kappa'})$, then, let the final reaction direction of π be κ', for all $D \in \mathsf{base}(\mathcal{Z}_{\kappa'})$, such that $\pi \restriction_\kappa \kappa' +\!\!+ [\mathscr{q}] \subseteq D$ for some $\mathscr{q}$, then there is a synchronisation path $\pi' \in \Pi$ such that π' has a prefix of the form $\pi +\!\!+ [(\mathscr{q}, -, -, -)]_{(-,-)}$.

This property ensures that any possible *down* resulting from an *up* to a direction must be accounted for. For instance, suppose there is an interaction with direction γ hinted by some paths that when going *up* to γ with control state $\mathscr{q}_1$, the possible *down* states are $\mathscr{q}_2$ and $\mathscr{q}_3$. Then, every other *up* visits to γ with $\mathscr{q}_1$ must handle both $\mathscr{q}_2$ and $\mathscr{q}_3$ as possible *down* to continue the computation. This property is crucial for capturing all possible synchronisation paths in Π.

Unlike rule totality, which requires capturing every rule decision at the current node, continuation totality requires handling every possible *down* from any direction.

Prefix Totality For any path π in Π and any non-empty prefix π_1 of π, let the reaction direction of the final reaction of π_1 be κ_1, if $\pi_1 \restriction_\kappa \kappa_1 \in \mathsf{base}(\mathcal{Z}_{\kappa_1})$, then π_1 must be included in Π.

This property aligns with the intuition that if there is a scenario where a visit to a direction (κ_1) is never returned, then there must be a path in Π that reflects this abrupt end.

Especially, the assumption of prefix totality is an additional requirement that is not readily existing in the following construction. This condition essentially requires that the function F to also contain sufficient prefix information.

Skeleton Totality The set $\mathcal{Z}_{\Downarrow}$ is exactly $\mathsf{DownSkel}_\kappa(\Pi, F)$, where F contains all functions f such that for every γ, $f(\gamma) \in \mathcal{Z}_\gamma$.

This property ensures that every skeleton for this node ($\mathcal{Z}$) can be inferred from Π and the upward skeletons, $\mathcal{Z}_\gamma$ for all γ.

Intuitively, this definition consists of a series of necessary conditions that must be fulfilled for the set to accurately represent the possible future paths of a node, assuming that F correctly encapsulates all interaction skeletons that are going to occur within and above the node.

Constructing Configuration Tree Automaton

With the definitions established earlier, we are now ready to present the complete construction of the configuration tree automaton, following the intuition previously outlined.

Specifically, we hereby construct a Non-deterministic Tree Automaton (NTA) $\mathcal{T} = (T, \Gamma, \Sigma^{\mathcal{A}}, \mathscr{T}, T_{init})$, where T consists of two types of states: $\Downarrow^\gamma_{(\mathcal{B},\mathcal{Z})}$, representing side branches, and $\Uparrow^{\mathcal{G}}_{(\mathcal{B},\mathcal{Z})}$, for the main branch.

As previously discussed, $\mathcal{B}$ captures all possible future interactions observed in a downward direction from the node, while $\mathcal{Z}$ describes all potential interactions with sort information. Thus, the following conditions must hold:

- Every interaction sequence in $\mathcal{B}$ must be a typing instance of at least one skeleton in $\mathcal{Z}$. Conversely, every skeleton in $\mathcal{Z}$ must have at least one typing instance in $\mathcal{B}$.
- For the state $\Downarrow^\gamma_{(\mathcal{B},\mathcal{Z})}$, all skeletons in $\mathcal{Z}$ must be full, whereas for the state $\Uparrow^\gamma_{(\mathcal{B},\mathcal{Z})}$, the skeletons set $\mathcal{Z}$ is required to contain either partial skeletons or the empty sequence ε.
- Both $\mathcal{B}$ and $\mathcal{Z}$ must be *non-empty*.

Both kinds of states are ranged over by $\Updownarrow^{\mathcal{G}}_{(\mathcal{B},\mathcal{Z})}$.

The initial state T_{init} is defined as the singleton $\{\Uparrow^\perp_{(\{\varepsilon\},\{\varepsilon\})}\}$, representing the assumption that the bottom of the stack has no interactions below it.

Finally, the transition relation $\mathscr{T}$ is designed to encompass rules for four distinct cases, with the first three pertaining to internal nodes and the last to leaf nodes:

1. We first address the rules that depict internal side branches, for which, let us consider all side-branch tree states $\Downarrow^{\gamma}_{(\mathcal{B},\mathcal{Z})}$.

   Specifically, we consider all possible triples r of the following form:

   $$\left(\Downarrow^{\gamma}_{(\mathcal{B},\mathcal{Z})}, \mathscr{m}, \left\{\gamma' \mapsto \Downarrow^{\gamma'}_{\left(\mathcal{B}_{\gamma'},\mathcal{Z}_{\gamma'}\right)} \;\middle|\; \gamma' \in \Gamma\right\}\right)$$

   across all $\mathscr{m} \in \mathcal{M}$ and all $\Downarrow^{\gamma'}_{\left(\mathcal{B}_{\gamma'},\mathcal{Z}_{\gamma'}\right)}$ for each γ', respectively. The rule is included in $\mathscr{T}$ if and only if the following two conditions are satisfied:

   - The chaining synchronisation paths set of r, Chains_r, is *locally total*.
   - For each γ', its observed interaction set $\mathcal{B}_{\gamma'}$ must exactly be the inferred instances $\mathsf{Instance}_{\Downarrow}(\mathsf{Chains}_r, \mathsf{UpSkels}_r, \mathcal{B}, \gamma')$. [3]

   Notably, when $\mathcal{Z}$ and all $\mathcal{Z}_\gamma$ for each γ are $\{\varepsilon\}$, it follows that $\mathcal{B}$ and all $\mathcal{B}_\gamma$ must also be $\{\varepsilon\}$. In this case, for any $\mathscr{m}$, the above rule must be included in $\mathscr{T}$, as the chaining set of the triple contains only the empty synchronisation path ε [4], for which the local totality condition trivially holds.

2. Consider rules for the main branch with the stack pointer *above* the current node, focusing on states $\Uparrow^{\mathscr{q}}_{(\mathcal{B},\mathcal{Z})}$.

   Specifically, we further consider each $\gamma_0 \in \Gamma$, intuitively representing the direction in which the stack pointer is positioned. Similar to Case 1, we examine each triple r of the form:

   $$\left(\Uparrow^{\mathscr{q}}_{(\mathcal{B},\mathcal{Z})}, \mathscr{m}, \left\{\gamma \mapsto \Downarrow^{\gamma}_{(\mathcal{B}_\gamma,\mathcal{Z}_\gamma)} \;\middle|\; \gamma \in \Gamma \setminus \{\gamma_0\}\right\} \uplus \left\{\gamma_0 \mapsto \Uparrow^{\gamma_0}_{\left(\mathcal{B}_{\gamma_0},\mathcal{Z}_{\gamma_0}\right)}\right\}\right)$$

   across all $\mathscr{m}$, all side-branch states $\Downarrow^{\gamma}_{(\mathcal{B}_\gamma,\mathcal{Z}_\gamma)}$ for each γ, and all main-branch states $\Uparrow^{\gamma_0}_{\left(\mathcal{B}_{\gamma_0},\mathcal{Z}_{\gamma_0}\right)}$ for the fixed γ_0.

   This rule is included in $\mathscr{T}$ under conditions similar to those in Case 1. Specifically, the chaining set of r, denoted Chains_r, is locally total, and for all γ, $\mathcal{B}_\gamma = \mathsf{Instance}_{\gamma_0}(\mathsf{Chains}_r, \mathsf{UpSkels}_r, \mathcal{B}, \gamma)$.

   Also note that this includes a special case. When both $\mathcal{B}$, $\mathcal{Z}$, and all $\mathcal{B}_\gamma$, $\mathcal{Z}_\gamma$ (including γ_0) are $\{\varepsilon\}$, then for all $\mathscr{m}$, the rule must be included in $\mathscr{T}$, for the same reason as in the previous case — the chaining path set contains only ε, which trivially satisfies local totality.

3. Define rules for nodes indicated by the stack pointer, considering $\Uparrow^{\mathscr{q}}_{(\mathcal{B},\mathcal{Z})}$ for each triple r:

   $$\left(\Uparrow^{\mathscr{q}}_{(\mathcal{B},\mathcal{Z})}, (\mathscr{q}, \mathscr{m}), \left\{\gamma \mapsto \Downarrow^{\gamma}_{(\mathcal{B}_\gamma,\mathcal{Z}_\gamma)} \;\middle|\; \gamma \in \Gamma\right\}\right)$$

[3] Recall that we are now using the set extension of functions to compute all combinations of elements from the sets Chains_r, $\mathsf{UpSkels}_r$, and $\mathcal{B}$, as per Remark 5.13.

[4] Recall that $\varepsilon \restriction - = \varepsilon$ by definition.

across all $q \in \mathcal{Q}$, $m \in \mathcal{M}$ and all side-breanches states $\Downarrow^{\gamma}_{(\mathcal{B}_\gamma, \mathcal{Z}_\gamma)}$ for all γ.

As before, one must first ensure conditions similar to the previous two cases. Specifically, the chaining set of r, Chains_r, is locally total, and for all γ, $\mathcal{B}_\gamma = \mathsf{Instance}_{\Uparrow}(\mathsf{Chains}_r, \mathsf{UpSkels}_r, \mathcal{B}, \gamma)$. To include the rule in $\mathscr{T}$, beyond these two conditions, it must also be ensured that φ exists in all types of $\mathsf{HdType}_{\Uparrow}(\mathsf{Chains}_r, \mathsf{UpSkels}_r, \mathcal{B})$ — indicating that all probable run types from this configuration are satisfying φ.

4. Finally, consider the leaf node that accepts the special tree-top ∇. In this case, the goal is to evaluate whether the given $\mathcal{Z}$ aligns with the intuition of "how an empty node could react to each *up* visit".

   Hence, technically, include the rule $(\Downarrow^{\gamma}_{(\mathcal{B},\mathcal{Z})}, \nabla, \emptyset)$, if the following conditions are met:

   - The content in $\mathcal{Z}$ must be probable, i.e., $wt(\mathit{TripsOrUp}^{\gamma}_{D^\triangle}) > 0$ for all $D^\triangle \in \mathcal{Z}$.
   - For every interaction $D \in \mathsf{base}(\mathcal{Z})$, let $\mathcal{Z}^{(D)}$ denote all skeletons in $\mathcal{Z}$ with base D. Then, intuitively, every such $\mathcal{Z}^{(D)}$ should cover exactly those probable for any given D, which is technically symbolised by: $\sum_{D^\triangle \in \mathcal{Z}^{(D)}} wt(\mathit{TripsOrUp}^{\gamma}_{D^\triangle}) = wt(\mathit{TripsOrUp}^{\gamma}_{D})$.
   - Additionally, one should ensure that the set $\mathsf{base}(\mathcal{Z})$ already handles "all possible reactions for each *up* visit". To this end, let P_O be the set of all odd-length prefixes of sequences from $\mathsf{base}(\mathcal{Z})$, and let P_E contain the even-length ones. Intuitively, each prefix in P_O, written as $D +\!\!+ [q]$, includes each "visit (by q) with history (D)." It is essential to ensure that each probable reaction is captured.

     Technically, we require the following conditions to hold:

     – For each *up* visit to the subject node, each probable non-returning reaction should be accounted for — namely:

     $$wt(\mathit{Up}^{\gamma}_{D_O}) > 0 \iff D_O \in \mathsf{base}(\mathcal{Z})$$

     for all odd-length prefixes $D_O \in P_O$;

     – Besides the non-returning case, for each *up* visit, each probable *down* should be accounted for:

     $$wt(\mathit{Trips}^{\gamma}_{D_O +\!\!+ [q]}) > 0 \iff D_O +\!\!+ [q] \in P_E$$

     for all odd-length prefixes $D_O \in P_O$ and all $q \in \mathcal{Q}$.
     Note that it is not required for every such reaction to be in the base set; it is only necessary for it to be a prefix of some full interactions. This is because every interaction within the base set signifies "the end of visiting", but this is often not the case that the reaction might be merely part of a longer visit — namely, after a *down* there is a

possibility that the automaton goes *up* again to the subject node with probability 1, meaning that this *down* is not an end but merely a prefix of further interactions.

Notably, in the case when $\mathcal{Z} = \mathcal{B} = \{\varepsilon\}$, then the above conditions are naturally satisfied and hence must be included.

Concerning the Case 4, it is pertinent to note that the first two condition of this case effectively involves the identification of all skeleton patterns $\mathit{TripsOrUp}^{\gamma}_{D^{\triangle}}$ with a weight > 0, termed as *probable skeleton patterns*. Once all such skeleton patterns are determined, the first two conditions equate to verifying that for every $D \in \mathsf{base}(\mathcal{Z})$, the set $\mathcal{Z}^{(D)}$ comprises exactly all the probable skeleton patterns $\mathit{TripsOrUp}^{\gamma}_{D^{\triangle}}$ for which $D = \mathsf{base}(D^{\triangle})$.

To substantiate this process, we propose the following lemma, whose proof is provided below in the last part of Section 5.2.4:

Lemma 5.30. *It is decidable whether a given skeleton pattern $\mathit{TripsOrUp}^{\gamma}_{D^{\triangle}}$ is probable.*

Illustrative Example

Let us then refer back to Example 5.14 to verify that the rules applied to each node on Figure 5.2 can be constructed by the above algorithm.

Firstly, we handle the trivial cases: the leaves at ℓ, r, $\gamma_\top r$, $\gamma_\top \ell \gamma_\top$, $\gamma_\top \ell r \ell$, $\gamma_\top \ell r r$, and $\gamma_\top \ell r \gamma_\top$. These all have the state of the form $\Downarrow^{-}_{(\{\varepsilon\},\{\varepsilon\})}$, which by Case 4 accepts ∇ as not-yet-created nodes. For the rest, we examine the configuration tree from the root.

At the root, the following rule exists in the construction as the special case of Case 2:

$$\Uparrow^{\bot}_{(\{\varepsilon\},\{\varepsilon\})} \xrightarrow{m_{\ell 0 r 2}} \left\{ \begin{array}{rcl} \ell & \mapsto & \Downarrow^{\ell}_{(\{\varepsilon\},\{\varepsilon\})} \\ r & \mapsto & \Downarrow^{r}_{(\{\varepsilon\},\{\varepsilon\})} \\ \gamma_\top & \mapsto & \Uparrow^{\gamma_\top}_{(\{\varepsilon\},\{\varepsilon\})} \end{array} \right\}$$

We then examine the node at $\gamma_\top \gamma_\top$. This node applies the rule at the leaf as described in Case 4. Once entering this node, only an infinite transitions of *up* moves towards $\gamma_\top$ exists. Therefore, the only possible type is $\tau_\top$, leading to the singleton skeleton set $\{[(q_\uparrow, \tau_\top)]\}$, easily verifiable to conform to the conditions of the leaf rule.

Next, we proceed to the examination of the node at $[\gamma_\top]$, whose rule and associated chaining set have been illustrated in Example 5.26 earlier. The rule governing this node falls under Case 2. Here, the skeleton context H is defined as $H(\ell) = \{[\overbrace{\quad}^{\vartheta_\ell} q_\ell]\}$, $H(r) = \{\varepsilon\}$, and $H(\gamma_\top) = \{[(q_\uparrow, \tau_\top)]\}$. It is straightforward to verify that the chaining set is locally total with respect to $H[\Downarrow \mapsto \varepsilon]$, thus confirming the structure.

Moreover, the observed interactions for its child nodes, namely $\mathcal{B}_\ell = \{[(q_\ell, \tau_\ell)]\}$, $\mathcal{B}_r = \{\varepsilon\}$, and $\mathcal{B}_{\gamma_\top} = \{[(q_\uparrow, \tau_\top)]\}$, are uniquely determined through an intuitive

computation of the Instance. In particular, the case of $\mathcal{B}_r$ is trivial, as no visits occur there; the case of $\mathcal{B}_{\gamma_\top}$ is similarly straightforward, as it is directly governed by $\mathcal{Z}_{\gamma_\top}$, which immediately establishes the run type as $\tau_\top$; finally, for the case of $\mathcal{B}_\ell$, the unique path allows one to infer the type τ_ℓ, given that the type for $\gamma_\top$ is already known to be $\tau_\top$. By inferring backwards through the unique chaining synchronisation path, namely Equation (5.28), the type τ_ℓ follows from that $\tau_\ell \xleftarrow{(q_\ell, m_{\ell 1 r 2}, \gamma_\top)} \tau_r$.

We now turn our attention to the node located at $\gamma_\top \ell$, whose application rule and chaining set are discussed in Example 5.29. The rule is derived from Case 2. It can be readily verified that the mentioned set of chaining synchronisation paths is locally total.

Furthermore, validation of the observed interactions for direction r as $\{[q_r, \tau_r]\}$ and for $\gamma_\top$ as $\{\varepsilon\}$ is similarly straightforward. As for $\mathcal{B}_\ell$, let Π represent the set of chaining paths. The fact that Π is equivalent to $\mathsf{Instance}(\Pi, F^{(H)}, \mathcal{B}, \ell)$ has already been effectively explained in the motivating example. It is particularly noteworthy that the explanation of $\mathcal{B}_\ell$ in that example serves as an ideal illustration of how the actual computation of Instance proceeds, particularly how the types are inferred in a backward manner through the synchronisation paths and the sorts in $\mathcal{Z}_\ell$.

Additionally, the preceding discussion offers ample evidence that $\mathcal{Z}_\ell$ contains all probable skeletons. As a result, for the node at $\gamma_\top \ell \ell$, the leaf node rule from Case 4 is applicable. Notably, it can be observed that the presence or absence of the empty skeleton ε makes no difference in the satisfaction of all the conditions for a leaf node.

Lastly, when examining the node with the stack pointer, we apply the rule from Case 3. The local totality of the set of synchronisation paths, which contains just a single path $[(q_\uparrow, q_r, m_{\ell 2 r 2}, \Downarrow)]_{(m_{\ell 2 r 2}, r)}$, is easily confirmed. Based on the discussion above in the motivating example, it is evident that φ_b resides within the only head type, thereby ensuring the rule's existence.

5.2.4 Proving the Model Checking Procedure

We proceed to validate the soundness and completeness of the construction. For clarity in exposition, we fix a specific tree automaton $\mathcal{T} = (T, \Gamma, \Sigma^{\mathcal{A}}, \mathscr{T}, T_{init} = \left\{\Uparrow_\varepsilon^\perp\right\})$ constructed for the automaton $\mathcal{A}$ against the LTL specification φ. In this part, we first present an auxiliary yet crucial proof that the pair $(\mathcal{B}, \mathcal{Z})$ in each reading state indeed holds the desired information as conceived in our intuitive description. Subsequently, the proof of the configuration tree automaton construction follows from this correctness. Finally, we address the claim made in Lemma 5.30 for deciding the probableness of a skeleton pattern.

Validating the Core Components: $(\mathcal{B}, \mathcal{Z})$ Pairs

We fix a particular configuration $c_0 = (\boxed{q_c}, \boxed{t_c}, \boxed{\rho_c})$, such that $t_0 := \mathsf{tree}(c_0)$ is accepted by $\mathcal{T}$.

To precisely define the functionalities of these pairs as per their intuition, we introduce the concept of a *probably deviating (PD) configuration* and the associated *probably deviating types*. Specifically, a configuration c' is considered

a PD configuration of $\mathscr{c}_0$ if: (1) there is a finite run from $\mathscr{c}_0$ to $\mathscr{c}'$; (2) the stack pointer of $\mathscr{c}'$ can be decomposed into $\rho\gamma$, and satisfies $\rho \in \mathrm{dom}(t_c)$ but $\rho\gamma \notin \mathrm{dom}(t_c)$; (3) the likelihood of runs from $\mathscr{c}'$ that do not revisit ρ' is non-zero. A type τ is labelled as a *probably deviating type* for $\mathscr{c}'$ if, with the stack pointer of $\mathscr{c}'$ being $\rho\gamma$, the probability of runs from $\mathscr{c}'$ not revisiting configurations with stack pointer ρ and type τ exceeds zero.

The term "deviating" signifies a configuration that departs from the current tree-stack t_c of $\mathscr{c}_0$ for the first time without returning, ensuring no revisits to any nodes of t_c. We then define that an infinite run from $\mathscr{c}_0$ is termed a *probably deviating run* if it can be divided into a finite segment reaching a PD configuration $\mathscr{c}'$, followed by an infinite segment that does not descend below the stack pointer of $\mathscr{c}'$. An *ending probably deviating type* of the run refers to a PD type of the PD configuration $\mathscr{c}'$. By definition, the probability of the set of all PD runs is 1.

Given the above definition, we can equivalently present another definition of the *interaction skeleton* as follows. By utilising the concept of PD configurations and the associated PD types, we can simplify our discussion of interaction skeletons to only *finite* runs. Given a PD run μ from $\mathscr{c}_0$ to a PD configuration $\mathscr{c}'$ and a PD type τ of $\mathscr{c}'$, we define $D^{\triangle}$ as the *interaction skeleton* at node ρ if the pre-run within and above ρ during μ aligns with the structure specified by $D^{\triangle}$. Specifically, recall that the concept of behaviours "within-and-above" has been formalised as $\mathsf{Abv}'(\mu, \rho)$ in Section 4.2.2 for decomposing a single pattern. Hence, the above sense could be formalised as, writing $D^{\triangle}$ as Equation (5.10), the interface of $\vec{\mu} = \mathsf{Abv}'(\mu, \rho)$ matches $\mathsf{base}(D^{\triangle})$, and for the ith segment of $\vec{\mu}$, when $i < n+1$, the sort of $\vec{\mu}$ is $\mathscr{s}_i$, otherwise, when $i = n+1$, the last segment of $\vec{\mu}$ could be backwardly inferred from τ to be aligning with τ_{n+1}.

In a similar manner, we can offer an alternative definition of the *observed interactions* at ρ during μ, this time focusing also on PD runs backwardly inferred from an ending PD type. Likewise, we also define the *possible future synchronisation paths* of a node on the tree-stack of $\mathscr{c}_0$ to be all the possible synchronisation paths that happen in the node across all possible finite paths from $\mathscr{c}_0$ to one of its PD configuration.

With this foundation, we are prepared to rigorously establish the meaning of the $(\mathcal{B}, \mathcal{Z})$ pairs through the following pivotal lemma:

Lemma 5.31. *At every node on the reading tree* RdT_{t_0}*, the reading state* $\Updownarrow^{\mathscr{g}}_{(\mathcal{B},\mathcal{Z})}$ *ensures that* $\mathcal{Z}$ *precisely captures all possible interaction skeletons of this node across all possible runs from* $\mathscr{c}_0$ *to any PD configurations with their respective PD types, and* $\mathcal{B}$ *accurately reflects all observed interactions throughout these runs.*

The proof of this lemma involves a relatively complex process, which is outlined in Figure 5.5. This process entails progressively breaking down the problem into smaller theorems in a sequential manner. For simplicity, we directly use the symbols $\mathcal{Z}$ and $\mathcal{B}$ generally denote the skeletons set and the observed interactions set at the reading state of the nodes, respectively. Besides, the conditions specified in Lemma 5.31 are briefly referred to as the *correctness* conditions for $\mathcal{Z}$ and $\mathcal{B}$, respectively.

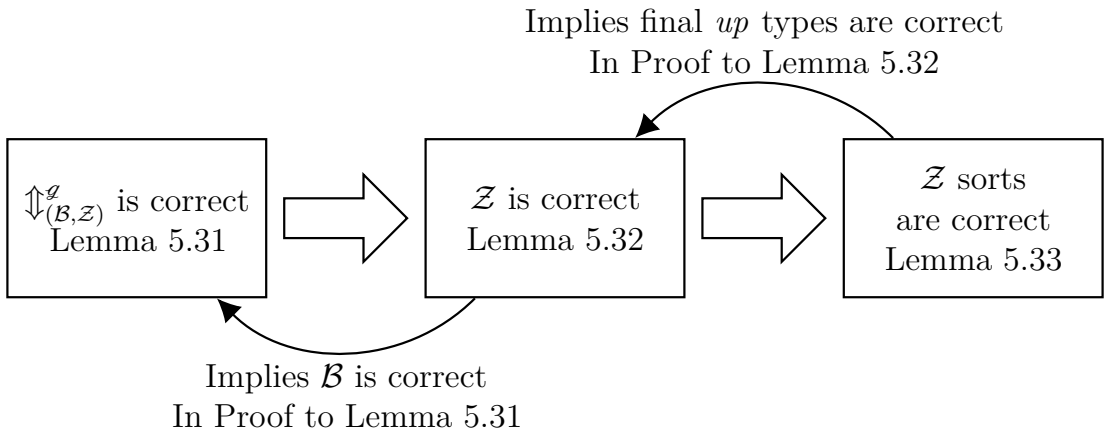

Figure 5.5: Proof Decomposition Overview

We start by focusing on the correctness of $\mathcal{Z}$, from which the correctness of $\mathcal{B}$, and additionally the local totality follows. To demonstrate the correctness of $\mathcal{Z}$, we further divide the problem into verifying the correctness of both the sorts and the final *up* types within the skeletons. In these two, ensuring the correctness of the sorts is crucial, as it underpins the correctness of the final *up* types.

Correctness of $\mathcal{Z}$ Let us then focus on the correctness of $\mathcal{Z}$, more specifically:

Lemma 5.32. *At every node on the reading tree* RdT_{t_0}*, the reading state* $\Updownarrow^{\mathcal{G}}_{(\mathcal{B},\mathcal{Z})}$ *has that* $\mathcal{Z}$ *is correct.*

To address this, we begin by considering the validity of the state and sort information, excluding the final *up* type, as demonstrated in the following lemma:

Lemma 5.33. *For the reading state* $\Updownarrow^{\mathcal{G}}_{(\mathcal{B},\mathcal{Z})}$ *at every node on the reading tree* RdT_{t_0}*, a skeleton,* regardless of its final *up* type, $\mathsf{RmTy}(D^{\triangle})$*, is included in* $\mathsf{RmTy}(\mathcal{Z})$*, if and only if there is a run from* c *to a PD configuration in which the interactions between this node and its parent adhere to the state and sort information of* $D^{\triangle}$*.*

Now, let us begin the description by discussing both soundness and completeness of the claim.

Soundness of Lemma 5.33 To establish soundness, we fix a specific node at the route $\boxed{\rho_0} \in \mathrm{dom}(t)$ and a specific skeleton $\boxed{D^{\triangle}_0}$ within the $\mathcal{Z}$ of this node.

We now work with skeletons where the final type is removed. We refer to this sequence as a *no-type skeleton*, denoted $D^{\overline{\triangle}}$, which takes the form:

$$[q_1 \overbrace{\quad}^{\mathfrak{s}_1} q'_1, \ldots, q_n \overbrace{\quad}^{\mathfrak{s}_n} q'_n, q_{n+1}] \tag{5.34}$$

where q_{n+1} is optional. The function base applies as it does for regular skeletons, yielding $[q_1, q'_1, \ldots, q_n, q'_n, q_{n+1}]$, with q_{n+1} being optional.

A *cohesive* pre-run $\vec{\mu}$ *conforms* to a $D^{\overline{\triangle}}$ if $\mathsf{Interface}(\vec{\mu}) = \mathsf{base}(D^{\overline{\triangle}})$ with each segment $\vec{\mu}[i]$ with $i < n+1$ having the sort $\mathfrak{s}_i$.

Hence, our objective for this soundness proof can be summarised as constructing a PD run μ such that $\mathsf{Abv}'(\mu, \rho_0)$ conforms to $\mathsf{RmTy}(D_0^{\triangle})$. Note that the case for the root is trivial, as any PD run satisfies the requirement. Now, we consider the case when $\rho_0 \neq \varepsilon$.

To support this, we introduce an auxiliary structure to "plan" the future transitions until a PD configuration is reached. The structure is denoted as $\varkappa$, a Γ-indexed tree, with the same domain as RdT_{t_0} and a pointer in $\mathrm{dom}(\mathsf{RdT}_{t_0})$. Each node deterministically stores the next operation to perform, ensuring that, by traversing the tree, one reaches a PD configuration. $\varkappa$ is treated as a tree, and its associated pointer is accessed as $\mathsf{Ptr}(\varkappa)$.

Formally, internal nodes in $\varkappa$ are labelled with pairs (π, f), where π denotes a synchronisation path with stack status $\mathscr{g}(\rho)$, with ρ being the route of this node at $\varkappa$, and f is a function mapping each $\gamma \in \Gamma$ to a no-type skeleton. This function must satisfy $\pi \restriction_\kappa \gamma = \mathsf{base}(f(\gamma))$ for all γ. If the node is at route ρ, then κ is determined as follows: $\Uparrow$ if $\rho = \mathsf{Ptr}(\varkappa)$, some $\gamma \in \Gamma$ if $\rho\gamma \sqsubseteq \mathsf{Ptr}(\varkappa)$, or $\Downarrow$ otherwise. This direction κ is referred to as the *source direction* of the node. Intuitively, π indicates the remaining path to execute in the current node, while f specifies the remaining sorts and interactions. Additionally, the root must be an internal node with no *down* operation in its synchronisation path. Also, when $\mathsf{Ptr}(\varkappa)$ refers to an internal node, the synchronisation path at this node must not be ε.

Leaves in $\varkappa$ are labelled by a *cohesive* pre-run $\vec{\mu}$, which intuitively indicates the remainder of the execution within and above this node. Here, a pre-run rather than a pre-path is adopted for some subtle technical reasons. Likewise, when $\mathsf{Ptr}(\varkappa)$ refers to a leaf node, the pre-run label must not be ε.

The structure $\varkappa$ is defined as *consistent* if, for every internal node at route ρ, with label $(-, f)$, the following holds for each $\gamma \in \Gamma$:

- If $\rho\gamma$ refers to an internal node, let $\varkappa(\rho\gamma) = (\pi_\gamma, f_\gamma)$. Then, $f(\gamma) = \mathsf{DownSkel}_\kappa^{(\mathscr{T})}(\pi_\gamma, f_\gamma)$, where κ is the source direction of the node at $\rho\gamma$.

  The function $\mathsf{DownSkel}_{\_}^{(\mathscr{T})}$, defined for no-type skeletons, is similar to its original version, except that in Equation (5.11), there is no τ, and no type inference is required.

- If $\rho\gamma$ refers to a leaf node, let $\varkappa(\rho\gamma) = (\vec{\mu}, -)$. In this case, $\vec{\mu}$ must *conform* to $f(\gamma)$.

Next, we define the *underlying configuration* of a consistent $\varkappa$ as $(\mathscr{q}, \mathscr{t}, \rho)$, where: $\rho_\varkappa = \mathsf{Ptr}(\varkappa)$, and $\rho = \mathsf{Ptr}(\varkappa)$, with $\mathscr{q}$ determined as follows:

- $\mathscr{q}$ is given as follows. If $\rho_\varkappa$ refers to a leaf node, let its label be $\vec{\mu}$, then $\mathscr{q}$ is the first control state in the first run; otherwise, when $\rho_\varkappa$ refers to an internal node, let the label be $(\pi, -)$, then $\mathscr{q}$ is the first source state in π.

- The tree-stack $\mathscr{t}$ is defined such that its domain is a superset of $\varkappa$'s domain. For each $\rho' \in \mathrm{dom}(\varkappa)$:

  - If ρ' refers to an internal node labelled $(\pi, -)$, and $\pi = [\dots]_{(m,-)}$ [5], then $t(\rho') = m$.
  - If ρ' refers to a leaf node labelled $\mu :: \vec{\mu}$, let the first configuration of μ be c, then the sub-tree of t rooted at ρ' is the suprapointer sub-tree of c.

- The stack pointer ρ is precisely ρ_x.

We say that a structure is *terminated* if: $\mathsf{Ptr}(x)$ points at a leaf node with a pre-run $[\mu]$, where μ is a soar.

By the definition of consistency, we observe:

Lemma 5.35. *If x is both consistent and terminated, then for every path ρ in its domain, $x(\rho)$ is of the form $(\varepsilon, -)$, except when $\rho = \mathsf{Ptr}(x)$.*

Proof. We first show, by induction from $\mathsf{Ptr}(x)$ to the root, that this holds for $\rho \sqsubseteq \mathsf{Ptr}(x)$. Then, we extend the result to all ρ by induction on the length of the path. In the first step, notice that $\pi \restriction_\gamma \gamma = \varepsilon$ iff $\pi = \varepsilon$ for all γ. In the second step, notice that the filtration of an empty synchronisation path is always empty. □

We define the *execution* and thus the *production* of a consistent but *not terminated* x as $\mathsf{Exec}(x)$, which produces a pair (ϕ, x'), where: Let $\rho = \mathsf{Ptr}(x)$:

- If ρ refers to an internal node labelled (π, f), let $\pi = [\zeta_1, \zeta_2, \dots, \zeta_n]_{(m_0,g)}$. Let ${\delta_1}^{(\pi)}$ be the first rule of the reaction unit, m_1 be the reacting memory of ζ_1, and κ be the reacting direction. Set $\pi' = [\zeta_2, \dots, \zeta_n]_{(m_1,g)}$; and define f' as f if $\kappa = \Downarrow$, or $f[\kappa \mapsto \mathsf{tail}(f(\kappa))]$ when $\kappa \in \Gamma$, where tail removes the first segment from the list. More specifically, for a no-type skeleton $D^{\overline{\triangle}}$ written as Equation (5.34), $\mathsf{tail}(D^{\overline{\triangle}})$ removes either an up-down interaction or the final *up* if $n = 0$.

  Then, $\phi = [{\delta_1}^{(\pi)}]$, and $x' = x[\rho \mapsto (\pi', f')]$. If $\kappa = \Downarrow$, $\mathsf{Ptr}(x')$ refers to the parent node of ρ, which must exist (the root cannot have $\Downarrow$). Otherwise, $\mathsf{Ptr}(x') = \rho\kappa$.

- If ρ refers to a leaf node, let its label be $\mu :: \vec{\mu}$, then $\phi = \mathsf{path}(\mu)$ and $x' = x[\rho \mapsto \vec{\mu}]$ with its pointer $\mathsf{Ptr}(x')$ pointing to the parent node of the node at ρ.

From this definition, it is evident that applying Exec preserves consistency.

Lemma 5.36. *If x is consistent and $\mathsf{Exec}(x) = (-, x')$, then x' is also consistent.*

Proof. This follows directly from the definitions. □

Lemma 5.37. *If x is consistent and $\mathsf{Exec}(x) = (\phi, x')$, then ϕ is enabled by the underlying configuration c of x, where the ending configuration of $\mathsf{host}(c, \phi)$ is exactly the underlying configuration of x'.*

[5] Recall that an empty synchronisation path is essentially of the form $[]_{(m,g)}$.

Proof. This follows directly from the definitions. □

If $\mathsf{Exec}(\varkappa) = (\phi, \varkappa')$, we say there is an execution from $\varkappa$ to $\varkappa'$ that produces ϕ. Thus, we call a sequence $\varkappa_1 \dots \varkappa_n$ an *execution* from $\varkappa_1$ if for every consecutive pair of elements $\varkappa_i \varkappa_{i+1}$, it holds that $\mathsf{Exec}(\varkappa_i) = (\phi_i, \varkappa_{i+1})$. We call $\phi_1 +\!\!+ \dots +\!\!+ \phi_{n-1}$ the *execution path* produced from the execution.

Further, we extend Sync to handle an initial direction $\kappa_s \in \mathcal{K}^{\Uparrow}$, defining $\mathsf{Sync}_{\kappa_s}(\pi, f)$ as follows:

- If $\kappa_s \notin \Gamma$, the result is simply $\mathsf{Sync}(\pi, f)$.
- If $\kappa_s \in \Gamma$, $f(\kappa_s)$ must be non-empty and decomposable as $\phi :: \vec{\phi}$. Let $f' = f[\kappa_s \mapsto \vec{\phi}]$ and $\vec{\phi}_r = \mathsf{Sync}(\pi, f')$. Then: If $\vec{\phi}_r = \varepsilon$, the result is $[\phi]$. If $\vec{\phi}_r \neq \varepsilon$, let $\vec{\phi}_r = \phi'_r :: \vec{\phi}'_r$. The result is $(\phi +\!\!+ \phi'_r) :: \vec{\phi}'_r$.

We define $\mathsf{PrePathAt}(\varkappa, \rho)$, where $\varkappa$ is consistent and ρ is within its domain, inductively as follows:

- If ρ refers to a leaf node, let $\varkappa(\rho) = \vec{\mu}$, then the result is $\mathsf{path}(\vec{\mu})$.
- If ρ refers to an internal node, $\varkappa(\rho) = (\pi, -)$, let $\mathsf{PrePathAt}(\varkappa, \rho\gamma)$ be $\vec{\phi}_\gamma$ for each $\gamma \in \Gamma$, then $\mathsf{PrePathAt}(\varkappa, \rho) = \mathsf{Sync}_\kappa(\pi, \rho\gamma)$, where κ is the source direction of the node at ρ.

It is straightforward to verify that the function calls to Sync_κ are well-defined, given the consistency of $\varkappa$. Thus, the following result holds:

Lemma 5.38. *For a consistent $\varkappa$ and a route $\rho\gamma$ within its domain, let $\varkappa(\rho) = (-, f)$. Then, $\mathsf{PrePathAt}(\varkappa, \rho\gamma)$ must conform to $f(\gamma)$.*

Proof. The proof proceeds by induction, starting from the leaves and moving down towards the root. The base case for leaves follows directly from the definition. For internal nodes, it follows from an inspection of $\mathsf{DownSkel}_\kappa^{(\mathcal{T})}$ from the definition of consistency, where κ is the source direction of the node, completing the induction. □

With all the preparations above, we can now state the following key lemma for introducing the structure:

Lemma 5.39. *If $\varkappa$ is consistent, there exists a finite execution from $\varkappa$ to a unique terminated $\varkappa'$, such that the execution path ϕ is enabled by the underlying configuration $\mathscr{C}$ of $\varkappa$. Furthermore, let the run $\mu = \mathsf{host}(\mathscr{C}, \phi)$. The ending configuration of μ is the underlying configuration of $\varkappa'$. Also, for every node $\rho\gamma$ in the domain of $\varkappa$, let $\varkappa(\rho) = (-, f)$, then the pre-run $\mathsf{Abv}'(\mu, \rho)$ must conform to $f(\gamma)$.*

Proof. The finiteness of the execution is demonstrated by defining a ranking function based on the lengths of the first elements across $\varkappa$. Each time Exec is applied, the value of this function decreases by one. The execution terminates at a leaf when the ranking function reaches one, as per the definition and Lemma 5.35. Also, such execution must terminate in a configuration as Exec maintains consistency, by Lemma 5.36.

The proof for the result in $\varkappa'$ then proceeds by induction on the length of the execution. The base case, where $\varkappa$ is already terminated, is trivial. The inductive case follows from Lemmas 5.36 and 5.37.

Finally, for conformance, notice that the transitions in μ corresponding to $\mathsf{path}(\mathsf{Abv}'(\mu, \rho\gamma))$ are precisely $\mathsf{PrePathAt}(\varkappa, \rho\gamma)$. Thus, the result follows directly from Lemma 5.38. □

With the above definitions and properties, we can now uniquely define a consistent structure $\varkappa_0$ for RdT_{ℓ_0} where $\mathsf{PrePathAt}(\varkappa_0, \rho_0)$ conforms to $\mathsf{RmTy}(D_0^{\triangle})$. Then, by applying the above lemma, we obtain a run μ, with $\mathsf{Abv}'(\mu, \rho_0)$ conforming to $\mathsf{RmTy}(D_0^{\triangle})$, thus demonstrating soundness. Finally, we will demonstrate that the underlying path of μ leads to a proper PD run from $\mathscr{c}_0$.

Notably, for ease of exposition, in the following construction, the internal nodes are labelled by (π, f), with f assigning each node a proper skeleton (not a no-type version). After construction, we modify each internal node to $(\pi, \mathsf{RmTy} \circ f)$ to form $\varkappa_0$.

The construction occurs in two stages: (1) we build the nodes on the prefix of ρ_0, inductively, top-down to the root; (2) we build those non-prefixing nodes inductively, bottom-up. We begin with the top node of the first stage — the node at ρ_0.

- The label at node ρ_0 is determined by the rule applied at that node:
  - For rules from Case 4, suggesting ρ_0 is a leaf node, we select a pre-run $\vec{\mu}$ from $\mathit{TripsOrUp}_{D_0^{\triangle}}^{\mathscr{g}(\rho_0)}$, which must exist since $\mathit{TripsOrUp}_{D_0^{\triangle}}^{\mathscr{g}(\rho_0)} > 0$. The label is then simply $\vec{\mu}$.
  - For rules from Case 1, 2, or 3, the node is internal. Given the *skeleton totality* of the local totality of Π, there exists a synchronisation path π and an upward skeletons context f such that $\mathsf{DownSkel}_{\kappa}^{(\mathscr{T})}(\pi, f) = \mathsf{RmTy}(D_0^{\triangle})$. Thus, the label is (π, f).
- We then progressively construct each $\rho \sqsubseteq \rho_0$ in descending order, starting from ρ_0. When constructing a node at ρ, we keep the original skeleton used in the upward node at $\rho\gamma$ to form the no-type skeleton, starting from $D_0^{\triangle}$. We refer to this original skeleton as $D^{\triangle}$.

  Revisiting the rule applied at node ρ (an internal node), let κ be its source direction and the reading state $(-, \mathcal{Z})$. By *mention totality*, we select a chaining path π such that $\pi \restriction_{\kappa} \gamma = \mathsf{base}(D^{\triangle})$. By *skeleton totality*, there exists an upward skeletons context f such that $f(\gamma) = D^{\triangle}$ with $D_{\Downarrow}^{\triangle} = \mathsf{DownSkel}_{\kappa}(\pi, f) \in \mathcal{Z}$. The node is then labelled (π, f), and the skeleton for the lower node construction is updated to $D_{\Downarrow}^{\triangle}$.

- For nodes on routes not prefixing ρ_0, $\varkappa$ is constructed inductively, bottom-up, starting from already constructed nodes. Each node must be accounted for, since the root has been constructed.

  If an internal node at ρ is constructed but its child node at $\rho\gamma$ is not, let the label at ρ be $(-, f)$. The construction follows the same process as for ρ_0, replacing $D_0^{\triangle}$ with $f(\gamma)$ and ρ_0 with $\rho\gamma$. By the TA rules, the $\mathcal{Z}$ at node $\rho\gamma$ must contain $f(\gamma)$.

Finally, for each internal node, we change the label from (π, f) to $(\pi, \mathsf{RmTy} \circ f)$ so that the RHS functions map to no-type skeletons.

The construction can be easily verified to be consistent. By applying Lemma 5.39, we obtain a run μ from the underlying configuration of $\varkappa_0$, where $\mathsf{Abv}'(\mu, \rho_0)$ conforms to $\mathsf{RmTy}(D_0^{\triangle})$. Note that Lemma 5.39 does not account for the root node, yet this case is trivial and has been handled in the beginning of our proof.

Let us express μ as $\mathscr{c} \xrightarrow{\phi} \mathscr{c}'$, where $\mathscr{c} = (-, t, -)$ and $\mathscr{c}' = (-, t', \rho')$.

Note that $\mathscr{c}$ is *tree-stack bisimilar* to the target configuration $\mathscr{c}_0$, as the domain of t matches that of RdT_t, and any additional domain elements, compared to $t_{\mathscr{c}}$, map to m_{init}.

Thus, by the properties of tree-stack bisimilarity, $\mathscr{c}_0$ also enables ϕ, and the resulting configuration $\mathscr{c}_0'$ is tree-stack bisimilar to $\mathscr{c}'$.

Upon inspecting $\mathscr{c}'$, we observe that the node at ρ' is not in $\mathrm{dom}(t_{\mathscr{c}})$, but its parent node is. From the construction of $\varkappa_0$, as the pre-runs in the leaves are taken from probable skeleton patterns, the probability of a soar from $\mathscr{c}'$ is thus implied to be greater than zero. Hence, $\mathscr{c}'$, with ρ' and a non-zero probability of soars, is a valid PD configuration.

Therefore, the run μ_0 of $\mathscr{c}_0 \xrightarrow{\phi} \mathscr{c}_0'$ is a valid PD run, with $\mathsf{Abv}'(\mu_0, \rho_0)$ conforming to $\mathsf{RmTy}(D_0^{\triangle})$, demonstrating the soundness of Lemma 5.33.

Completeness of Lemma 5.33 To demonstrate completeness, we fix a specific run μ from $\mathscr{c}_0$ to some PD configuration $\mathscr{c}_0'$ and proceed by contradiction. Suppose there exists a node ρ where the interaction skeleton of μ is not contained in the set $\mathcal{Z}$ of ρ; we will derive contradictions. Let $D_0^{\triangle}$ be the interaction skeleton of μ at ρ. We classify potential violations into two types:

- Prefix violation: $D_0^{\triangle}$ is not a prefix of any skeleton in $\mathcal{Z}$.
- Prefix cut: $D_0^{\triangle}$ is a prefix but not a complete element in $\mathcal{Z}$.

It is clear that if neither a prefix violation nor a prefix cut occurs, completeness is guaranteed — $D_0^{\triangle}$ must be within $\mathcal{Z}$. In this context, we first show that a prefix violation is not possible through the following theorem:

Lemma 5.40. *For any prefix of μ, its interaction skeleton at* each node *must align with the prefix of certain elements in $\mathcal{Z}$ associated with the node. Also, its path within this node, if it is internal, must be the prefix of some elements in the chaining path set of the rule applied at the node.*

Proof. We establish this theorem by induction on the length of the prefix of μ. Note that the prefix ε must belong to the set of prefixes of $\mathcal{Z}$, since $\mathcal{Z}$ is defined to be non-empty.

For the base case, we consider the first transition of μ. We argue that this transition must appear as the first element of some chaining path. By the definition of the chaining paths set, every chaining path must begin with a rule $(q, m, g) \xrightarrow{-} op$, where the LHS (q, m, g) matches the status of the first configuration in μ. Thus, by *rule totality*, there exists a rule whose op matches the first transition of μ. Consequently, by the definition of the chaining set, the first move must yield a prefix in $\mathsf{base}(\mathcal{Z}_\kappa)$, with κ corresponding to the direction of the first transition.

Regarding sort information, if the move is an *up* move, no sort condition applies, as sorting cannot be judged locally. For a *down* move, the sort condition is satisfied by construction, due to *skeleton totality*.

Now, consider the inductive case. Assume a prefix μ' (with at least one transition) satisfies the condition. Let the last transition in μ' visiting the current node with state q.

By the induction hypothesis, μ' has an interaction skeleton matching some prefixes in $\mathcal{Z}$, and for its child node at each direction γ', matching some prefixes in $\mathcal{Z}_{\gamma'}$. This implies that earlier transitions in the newly reached node result in filtration prefixes in $\mathsf{base}(\mathcal{Z})$, and for the child nodes at γ', in $\mathsf{base}(\mathcal{Z}_{\gamma'})$ for each γ'. This satisfies the requirements of *continuation totality*. Therefore, a synchronisation path π is guaranteed to exist in the chaining set, whose prefix π_1 matches the earlier transitions in this node by μ', with the source state of the next reaction unit after π_1 in π exactly matching the the new state q.

By *rule totality*, there exists a synchronisation path π' with prefix $\pi_1 +\!\!+ [\delta]$, where the rule δ begins with state q and aligns with the new transition. The sort condition is also preserved, based on the induction hypothesis for upward transitions and the identified synchronisation path, due to *skeleton totality*. □

We now shift focus to establishing that a prefix cut cannot occur. We distinguish between two types of prefix cuts: *up* cut and *down* cut. As the names suggest, an *up* cut occurs when the prefix is truncated by an *up* transition, whereas a *down* cut follows a *down* transition. Let us first inspect the properties of prefix cuts.

Lemma 5.41. *If a prefix cut occurs, then there must also be an up cut.*

Proof. Assume for contradiction that only *down* cuts occur. First, identify the lowest node where the prefix cut occurs, and consider its parent node as the current node (which must exist since the root node cannot experience a prefix cut). Let the skeletons set in the reading state of the current node be $\mathcal{Z}$, and for the child node in direction γ, the set be $\mathcal{Z}_\gamma$ for all γ. By Lemma 5.40, there exists a synchronisation path π in the chaining set Π at the current node, such that the transitions in μ at this node, denoted π', form a prefix of π.

Now, consider the final reacting direction of π'. If it ends with a *down* transition, then, since no prefix cut occurs at the current node, we have $\pi' \restriction_\kappa \Downarrow = \mathsf{base}(\mathcal{Z})$, where κ is the source direction of the current node. This implies that π' is in

the chaining set Π, due to the *prefix totality* of local totality. By the definition of the chaining set, $\pi' \restriction_\kappa \gamma \in \mathsf{base}(\mathcal{Z}_\gamma)$ holds, indicating that no prefix cut occurs, contradicting the initial assumption.

If π' ends with an *up* transition, and since no *up* cut occurs, let the final *up* direction be γ_u. Then, $\pi' \restriction_\kappa \gamma_u \in \mathsf{base}(\mathcal{Z}_{\gamma_u})$. By prefix totality, π' must also be in the chaining set, which, following the same reasoning as above, indicates that no prefix cut occurs, again contradicting the assumption. □

Lemma 5.42. *If a node experiences an up cut, then it is either a leaf node, or there is also an up cut in its child node.*

Proof. The case for a leaf node is trivial as there is no further consideration necessary.

In the second scenario, if a run has an *up* cut at an internal node, given Lemma 5.40, let the run at this node be π', then we know there must be some synchronisation path π in the chaining set Π at this node with π' as a prefix. Let the last reacting direction of π', given that *up* cut happens, must be a $\gamma \in \Gamma$, and let the skeletons set in the reading state of the child node at γ be $\mathcal{Z}_\gamma$.

Assume for contradiction that there is no *up* cut. As the segment towards γ by assumption is uncut and is present in $\mathsf{base}(\mathcal{Z}_\gamma)$. By prefix totality of local totality, π' is also in the chaining set Π. Therefore, by the definition of chaining set, there should be no prefix cut at the current node, contradicting the condition that it is an *up* cut. □

By the lemmas, we establish that if a prefix cut occurs, it must culminate in an *up* cut at a leaf — which contradicts the construction stipulated in Case 4 for leaf nodes, asserting that it must conclude in a non-PD configuration. Consequently, a prefix cut is not possible.

In conclusion, completeness is thereby demonstrated, which, in conjunction with the previous soundness result, also proves the target Lemma 5.33.

Correctness of Local Totality With Lemma 5.33 established, we are now ready to demonstrate the correctness of our local totality claim that the chaining set contains exactly the possible future paths. This is of essential significance to both the overall correctness of $\mathcal{Z}$ and $\mathcal{B}$.

Let us first introduce some auxiliary definitions and their properties.

We will introduce a direct yet essential technical lemma. Intuitively, this lemma conveys that replacing the behaviour of a pre-path within and above a certain node with another pre-path sharing the same interface results in another valid run.

We define *path-based upward substitution*, referred to simply as substitution. Consider a path ϕ enabled by a configuration $\mathscr{c}$, along with a specific route x that ends with a stack symbol γ. Let $\mu = \mathsf{host}(\mathscr{c}, \phi)$. Recall that the function $\mathsf{Abv}'(\mu, x)$ yields a sequence of consecutive segments within μ, allowing us to express ϕ as:

$$\phi_1\phi_1''\phi_2\phi_2'' \dots \phi_n\phi_n''\phi_{n+1}$$

Here, ϕ_i'' represents the segment corresponding to the ith part extracted by $\mathsf{Abv}'(\mu, x)$.

Now, consider a *cohesive* pre-path $\vec{\phi'} = [\phi'_1, \ldots, \phi'_n]$ with the same interface as $\mathsf{Abv}'(\mu, x)$. Substituting the paths of μ from x with the pre-path $\vec{\phi'}$ results in a sequence:

$$\phi_1 \phi''_1 \phi_2 \phi''_2 \ldots \phi_n \phi''_n \phi_{n+1} \tag{5.43}$$

We denote this entire substitution process as $\mathsf{Subst}(\mu, x, \vec{\phi'})$.

We now assert the following claim regarding the validity of the substitution:

Lemma 5.44. *Any defined substitution* $\mathsf{Subst}(\mu, x, \vec{\phi})$ *yields a valid path enabled by a configuration* c *as follows: For any configuration* c' *enabling* $\vec{\phi}$*, the configuration* c *is obtained by substituting the sub-tree at* x *in the tree-stack of* $\mathsf{hd}(\mu)$ *with the suprapointer sub-tree from* c'*.*

Next, observe the following fact about runs, which intuitively states that if a sub-tree is not visited during the run, it must remain unchanged.

Fact 5.45. For any finite run μ from c to c', and for any route ρ on the tree-stack of c, if there is no configuration with tree-stack having a prefix ρ within in μ, except c', then the sub-tree from ρ in c is identical to that in c'.

Then, the proof of Lemma 5.44 could be simply given by:

Proof (Lemma 5.44). The proof proceeds by straightforward induction on the length of $\vec{\phi}$, using an alternating application of Fact 5.45 (for ϕ_i in Equation (5.43)) and Lemma 4.32 (for ϕ''_i). □

Next, recall that the basic assumption for the rPTSA now differs from that in the termination analysis chapter. Specifically, while the bottom transitions are now non-terminating, they remain non-trivial, as in the termination analysis chapter.

Thus, in addition to the Sync property developed for non-root nodes (see Lemma 4.35), we now extend this property to the root node. It is important to note that the definition of Sync and synchronisation paths does not exclude the root case.

Lemma 5.46. *Under the given assumption, let* $\pi = [\ldots]_{(m,\gamma)}$ *be a synchronisation path for the root node, and let* f *be a function mapping each stack symbol to cohesive pre-paths such that* Sync *is defined. Then* $\mathsf{Sync}(\pi, f)$ *yields a singleton pre-path enabled by any configuration* $c = (-, t, \bot)$ *such that: (1)* $t(\varepsilon) = m$*; and (2) for each* $\gamma \in \Gamma$*,* $(q, t, [\gamma])$ *enables* $f(\gamma)$*, where* q *is the first source state of* $f(\gamma)$ *if it is non-empty, or* q_{init} *otherwise.*

Proof. The proof follows the same procedure as in Lemma 4.35, constructing a valid run μ starting from the given configuration, with $[\mathsf{path}(\mu)] = \mathsf{Sync}(\pi, f)$. Note that this case is simpler than the original proof, as we only need to consider the scenario where the rules in π involve *up* operations. □

Further, let us provide more detail about the the correctness of $\mathcal{Z}$ by introducing the concept of *Possible Over-Node Pre-paths* (PoNP) that as intuitively the behaviors within-and-above a given node in the configuration tree under inspection. Specifically, we say the PoNP of a node at route ρ of the configuration under inspection $\mathsf{tree}(\mathscr{c})$, conforming to an interaction skeleton $D^{\triangle}$, denoted as $PoNP(\rho, D^{\triangle})$, contains all pre-paths $\mathsf{path}(\mathsf{Abv}'(\mu, \rho))$ across all PD runs μ.

Further, for a specific internal node on the reading tree, we define that a upward skeletons context of the node to be that of the applied rule.

Now, we are ready to verify our sufficiency claim of local totality as follows:

Lemma 5.47. *For any internal node on the reading tree RdT_{t_0} of a configuration $\mathscr{c}$, the chaining set of the rule applied at this node corresponds exactly to the set of possible future paths that may occur at this node.*

Specifically, let the node be at route ρ, the source direction be κ, for any chaining path π and any upward skeletons context f, for any function h such that $h(\gamma) \in PoNP(\rho\gamma, f(\gamma))$, $\mathsf{Sync}_{\kappa}(\pi, h)$, if defined, must be in $PoNP(\rho, D^{\triangle})$ where $D^{\triangle} = \mathsf{DownSkel}_{\kappa}(\pi, f)$. Conversely, every possible future path is in the chaining set.

Proof. We begin by assuming that $\mathsf{RmTy}(\mathcal{Z})$ is correct at each node, as established by Lemma 5.33. This assumption implies that the interactions between each node and its parent node must lie within $\mathsf{base}(\mathcal{Z})$, where $\mathcal{Z}$ is in its reading state. Consequently, it is evident that, by the definition of a chaining set, any possible future path must lie within the chaining set. This shows the reverse direction.

For the forward direction, consider a specific combination of π, f, and h, such that $\mathsf{Sync}_{\kappa}(\pi, h)$ is defined. The objective is to construct a pre-run such that its behaviour within-and-above ρ follows the pre-path $\mathsf{Sync}_{\kappa}(\pi, h)$. The case for an empty synchronisation path is trivial, so we focus on the non-empty case.

By definition, for each γ, we may select a PD run μ_{γ} such that the pre-path within-and-above the child node at γ, $\mathsf{path}\Big(\mathsf{Abv}'(\mu_{\gamma}, \rho\gamma)\Big)$, is equivalent to $h(\gamma)$. Further, since $\mathsf{RmTy}(\mathcal{Z})$ is correct for all nodes, we may choose a run $\mu_{\Downarrow}$ such that the interface of $\mathsf{Abv}'(\mu_{\Downarrow}, \rho)$ is $\pi \restriction_{\kappa} \Downarrow$.

Next, let μ_p be the pre-fix of μ_{κ} (inclusively) up to the first occurrence of the current node if $\kappa \in \mathcal{K}$. Otherwise, we let $\mu_p = [\mathscr{c}]$, denoting an empty transition run. [6] Denote the last configuration in μ_p as $\mathscr{c}_c = (-, \mathscr{t}_p, -)$. We then redefine μ_{κ} as the original run with the pre-fix μ_p removed.

If ρ is at the root, we apply Lemma 5.46; otherwise, we apply Lemma 4.35. In either case, we deduce that $\mathscr{c}_c$ enables the pre-path formed by $\vec{\phi}_r = \mathsf{Sync}(\pi, f')$, where $f'(\gamma)$ is defined as $\mathsf{path}(\mathsf{Abv}'(\mu_{\gamma}, \rho\gamma))$ for each γ.

We further adjust $\mu_{\Downarrow}$ by removing the prefix of the original run (exclusively) up to the first occurrence of a configuration with stack pointer ρ. Let this configuration be $\mathscr{c}_{\Downarrow}$. By applying the substitution principle from Lemma 5.44, we deduce that the pre-path $\vec{\phi}_c = \mathsf{Subst}(\mu_{\Downarrow}, \rho, \vec{\phi}_r)$ is also enabled by $\mathscr{c}_c$. Note that if $\kappa \in \Gamma$, then $\mathscr{c}_{\Downarrow}$ and $\mathscr{c}_c$ share the same tree-stack, except for the sub-tree at ρ; otherwise, $\mathscr{c}_{\Downarrow}$ is

[6] Recall that, by definition, a run must never be empty, as it inherits the trajectories of Markov chains.

identical to $\mathscr{c}_c$. Hence, by the substitution principle, in either case, the substituted pre-path is enabled by $\mathscr{c}_c$, as the substitution of the sub-tree at ρ in the tree-stack of $\mathscr{c}_{\Downarrow}$ with that of $\mathscr{c}_c$ yields exactly $\mathscr{c}_c$.

For the enabled run μ_c from $\mathscr{c}_c$ with $\vec{\phi}_c$, it must eventually reach a PD configuration, regardless of the final reacting direction in π. If the final direction in π is *down*, the transitions above ρ do not affect the tree structure below ρ, and the final PD configuration reached by the original run $\mu_{\Downarrow}$ is suprapointer bisimilar to the configuration reached by μ_c. More specifically, the two configurations differ only in the sub-tree originating from route ρ. Conversely, if the final direction is *up* towards a direction γ, it can be observed that μ terminates in a configuration suprapointer bisimilar to that of μ_γ. More precisely, the two configurations share the same sub-tree from path $\rho\gamma$.

By concatenating μ_p and μ_c, we obtain a PD run from the current configuration accepted by $\mathcal{T}$. Note that $\mathsf{path}(\mu_p)$, when $\kappa \in \Gamma$, corresponds to the first part in $f(\kappa)$, which has been removed when calling $\mathsf{Sync}_\kappa(\pi, f)$. Hence, by combining it with $\vec{\phi}_r$ as the result of Sync, $\mathsf{Sync}_\kappa(\pi, f)$ describes precisely the pre-path within-and-above ρ for μ_p. Also, by the computation of $\mathsf{DownSkel}(\pi, f)$, it naturally holds that $\mathsf{Sync}_\kappa(\pi, f)$ conforms to $D^{\triangle}$. This concludes the forward case. Together with the reverse case at the beginning, the entire lemma is established. □

Proving Correctness of $\mathcal{Z}$ Then, we are ready to present the proof for the correctness of $\mathcal{Z}$.

Proof of Lemma 5.32. We apply induction in an upside-down manner, starting from the leaves of the reading tree RdT_{t_0}.

In the base case, rules are applied as specified by Case 4. Consider a node with stack status γ. By Lemma 5.33, an *Up* skeleton $D^{\triangle} = [\ldots, (\mathscr{q}, \tau)]$ belongs to $\mathcal{Z}$ if and only if there is a run that visits this node in accordance with $\mathsf{RmTy}(D^{\triangle})$. Given the TA rule construction condition in Case 4 that $wt(Up^{\gamma}_{D^{\triangle}}) > 0$, every such run reaches a PD configuration with PD type τ.

Subsequently, we consider the rules applied to the internal nodes in the inductive case. By the correctness of local totality, i.e., Lemma 5.47, further with the local totality's *skeleton totality* condition and the uniqueness of backward inference, the correctness of $\mathcal{Z}$ is straightforwardly established. □

Correctness of $\mathcal{B}$ Further, we present the following full statement regarding the soundness of $\mathcal{B}$, from which we may verify Lemma 5.31.

Lemma 5.48. *For every node on the reading tree RdT_{t_0}, let the node be at route ρ, and let its reading state be $(\mathcal{B}, \mathcal{Z})$. For every $D^{\triangle} \in \mathcal{Z}$ and any instance $D^{\star}$ of $D^{\triangle}$ in $\mathcal{B}$, it holds that for every $\vec{\phi} \in PoNP(\rho, D^{\triangle})$, there exists a PD run μ with $\mathsf{path}(\mathsf{Abv}'(\mu, \rho))$ that has observed interactions as $D^{\star}$ in ρ.*

Proof. We proceed by induction on the length of ρ.

In the base case, when $\rho = \varepsilon$, the result is trivial, as both $\mathcal{B}$ and $\mathcal{Z}$ have only the empty sequence ε. Hence, any PD run satisfies the condition.

For the inductive case, assume the statement holds at a node at ρ. If it holds at the leaf node, no further nodes above it need to be considered by induction. Hence, we here consider the case when the node is an internal node. Let the reading state of the node at ρ be $(\mathcal{B}, \mathcal{Z})$, and the reading state of the node at $\rho\gamma$ be $(\mathcal{B}_\gamma, \mathcal{Z}_\gamma)$ for each γ. Our objective is to show that the condition holds for every child node above this node. Consider a specific child node in direction γ_0.

To confirm the condition, consider an observed interaction sequence $D^\star_{\gamma_0}$ within $\mathcal{B}_{\gamma_0}$. By rule construction, we consider all possible chaining paths π at ρ, any upward skeletons context f, and any observed sequence $D^\star \in \mathcal{B}$, such that the equation $D^\star = \mathsf{Instance}(\pi, f, D^\star, \gamma_0)$ holds. By the correctness of local totality established in Lemma 5.47, we know that for any h with $h(\gamma) \in \mathit{PoNP}(\rho\gamma, f(\gamma))$ for all γ, we have $\vec{\phi} = \mathsf{Sync}_\kappa(\pi, h)$ in $\mathit{PoNP}(\rho, D^\triangle)$, where $D^\triangle = \mathsf{DownSkel}(\pi, f)$ must have an instance $D^\star$ and κ is the source direction of the rule applied at ρ.

Then, by the induction hypothesis, there must be such a PD run traversing via $\vec{\phi}$ with observed interactions as $D^\star$ at the node at ρ. Hence, by the definition of $\mathsf{Instance}$, this run must also exhibit the observed interaction $D^\star_{\gamma_0}$ at $\rho\gamma_0$ by backward inference, thereby confirming the establishment of the entire lemma. □

Then, we can verify Lemma 5.31.

Proof of Lemma 5.31. We begin by assuming the correctness of all $\mathcal{Z}$ at each node, as guaranteed by Lemma 5.32. We may then focus on the correctness of $\mathcal{B}$. The soundness of $\mathcal{B}$ follows directly from Lemma 5.48 by recalling the definition that every $D^\star$ must be an instance of at least one skeleton in $\mathcal{Z}$.

For completeness, consider that, for every node on RdT_{t_0}, let it be at ρ, the observed interactions from any PD run at ρ must be captured. We proceed by induction on the length of ρ, similar to the proof for soundness.

The base case for the root is trivial, as is the inductive case for leaf nodes. We now consider the case for an internal node.

Again, let the reading state of the node at ρ be $(\mathcal{B}, \mathcal{Z})$, and the reading state of the node at $\rho\gamma$ be $(\mathcal{B}_\gamma, \mathcal{Z}_\gamma)$ for each γ.

Consider a direction γ_0. There exists a PD run μ with final PD type τ, whose observed interactions are $D^\star_{\gamma_0}$. By the induction hypothesis, there must be an $D^\star \in \mathcal{B}$ that corresponds to the observed interactions of μ with τ at ρ.

By the correctness of $\mathcal{Z}$ and $\mathcal{Z}_\gamma$ for all γ, there must exist a corresponding skeleton context f such that $f(\gamma)$ is the skeleton context performed by μ with τ at $\rho\gamma$ for every γ. Let the synchronisation path of μ at ρ be π. Thus, by backward inference, $\mathsf{Instance}(\pi, f, D^\star, \gamma_0) = D^\star_{\gamma_0}$. Hence, $D^\star_{\gamma_0}$ is contained within $\mathcal{B}_{\gamma_0}$.

This concludes the completeness of $\mathcal{B}$. Combining this result with the soundness established earlier, the overall correctness of $\mathcal{B}$ is confirmed. □

Proof of the Model Checking Procedure

We are now prepared to provide a proof for the algorithmic construction.

Soundness For the soundness proof, assume we have a configuration $\mathscr{c}$ such that $\mathsf{tree}(\mathscr{c})$ is accepted by the constructed automaton $\mathcal{T}$. We demonstrate that the configuration almost surely satisfies φ.

Initially, we construct the reading tree of $\mathsf{tree}(\mathscr{c})$ for $\mathcal{T}$, denoted $\mathsf{RdT}^{(\mathcal{T})}_{\mathsf{tree}(\mathscr{c})}$. By Lemma 5.31, we know that at the node indicated by the stack pointer of $\mathscr{c}$ and its immediate child nodes, the $\mathcal{B}$ and $\mathcal{Z}$ within them encompass all possible transitions towards all probable deviation configurations with probable deviation types. Hence, by the rule applied from Case 3 of the construction, we infer backward that φ must be present in all types from the current run, indicating that all runs towards probable deviation configurations with probable deviation types satisfy φ. Given that the probability of all these runs is 1, we conclude that the configuration satisfies φ with probability 1.

Completeness For the completeness of the construction, such that all configurations $\mathscr{c}$ satisfying φ with probability 1 are accepted by $\mathcal{T}$, we need only construct a tree $\mathscr{t}$ that at each node of $\mathsf{tree}(\mathscr{c})$ attaches the information of all possible running skeletons $\mathcal{Z}$ and observed interactions $\mathcal{B}$ for all runs from $\mathscr{c}$ reaching any probable deviation configurations with any probable deviation types.

For each node at stack status $\mathscr{g}$, with the collected $\mathcal{B}$ and $\mathcal{Z}$ information, if it is along a path on $\mathsf{tree}(\mathscr{c})$ that is a prefix of the stack pointer of $\mathscr{c}$, we let the additional information attached be $\Uparrow^{\mathscr{g}}_{(\mathcal{B},\mathcal{Z})}$; otherwise, it is $\Downarrow^{\mathscr{g}}_{(\mathcal{B},\mathcal{Z})}$.

We then demonstrate that $\mathscr{t}$ constitutes a reading tree of $\mathcal{T}$ accepting $\mathsf{tree}(\mathscr{c})$. This can be established by induction on the (length of) path in the domain of $\mathscr{t}$.

In the base case, the state is undoubtedly $\Uparrow^{\perp}_{(\{\varepsilon\},\{\varepsilon\})}$, which exactly matches the unique initial state of $\mathcal{T}$.

In the inductive case, assuming the correctness of the current node, each of its child nodes is examined, which is straightforward. It is important to note that at each step, the set of all the synchronisation paths that could occur will undoubtedly fulfil the local totality. At the leaves, the interactions $\mathcal{B}$ and $\mathcal{Z}$ will assuredly meet the criteria set out in Case 4.

Computing Probable Skeleton Patterns

The remaining work involves the local qualitative model checking necessary to prove Lemma 5.30. To demonstrate the theorem, we utilise the decidability of qualitative local LTL model checking, underpinned by the ability to translate an LTL formula into a DRA (Deterministic Rabin Automaton) specification.

Intuition Recall that the pre-runs of a skeleton *Up* pattern $Up^{\gamma}_{D^{\triangle} +\!\!+ [(\mathscr{g},\tau)]}$ consist of two parts: one from $Trips^{\gamma}_{D^{\triangle}}$ and another being an infinite run from one of the possible final tree-stack forms of type τ.

A type τ represents a subset of potential sub-formulas. Each type can be equivalently interpreted as an LTL formula by conjoining all sub-formulas within it and negating those not included. Formally, we define this formula as φ_τ, expressed as:

$$\varphi_\tau := \Big(\bigwedge_{\varphi' \in \tau} \varphi'\Big) \wedge \Big(\bigwedge_{\varphi' \in \mathsf{SubFml}(\varphi) \setminus \tau} \neg\varphi'\Big)$$

The issue of whether a skeleton *Up* pattern $v = Up^{\gamma}_{D^{\triangle} +\!\!+ [(q,\tau)]}$ is probable (having weight > 0) translates into the following question: Is there at least one possible tree-stack t existing at the end of some pre-run in accordance to $D^{\triangle}$ that, when resumed by q, the set of infinite runs without going *down* from the subject node again has non-0 probability of satisfying φ_τ? This equivalence could be illustrated by the following observation that: if such a tree-stack form exists as the tree-stack at the end of a pre-run $\vec{\mu} \in Trips^{\gamma}_{D^{\triangle}}$, then $wt(\vec{\mu}) \cdot \mathbb{P}\left[\{\mu \mid \vec{\mu} +\!\!+ [\mu] \in v, \mu \models \varphi_\tau\}\right] > 0$ – by definition, the final weight must exceed zero. Conversely, if no such tree-stack form exists, the cumulative weight of v equals zero, as it is a sum of infinitely many zeros.

Since any LTL formula can be transformed into an equivalent DRA, we convert the LTL formula φ_τ into a corresponding DRA $\mathcal{D}_\tau$. We then ascertain if tree-stack forms exist post any run in $Trips^{\gamma}_{D^{\triangle}}$, in its final form, the runs which do not go *down* again and are accepted by $\mathcal{D}_\tau$ have a non-zero probability.

To address the potentially infinite tree-forms arising after executing pre-runs in $TripsOrUp^{\gamma}_{D^{\triangle}}$, we define a specific type of patterns and utilise inductive relationships among these patterns to encapsulate all possibilities.

For simplicity, we specifically call the objects with respect to the product rPTSA that $\mathcal{A} \times \mathcal{D}$ to be the $\mathcal{D}$-*observed* objects to distinguish from the objects with respect to the original rPTSA $\mathcal{A}$.

Mix Patterns with Sort Histories Given a DRA $\mathcal{D} = (\mathfrak{D}, -, -, \mathscr{d}_{init}, -)$ (and its associated simple interpretation function), we introduce a novel type of pattern called the *mix pattern with sort histories*. This pattern, denoted as $TripsOrUp^{\gamma}_{[D^{\triangle}|D_{\times\mathcal{D}}]}$, involves: $D^{\triangle}$, representing a *Trips* full skeleton; $D_{\times\mathcal{D}}$, signifying an interaction sequence of the product rPTSA $\mathcal{A} \times \mathcal{D}$, which could manifest as even-length when the entire pattern is *Trips* or odd-length when it is *Up*.

Furthermore, we define the *base object*, obtained by the function $\mathsf{base}(-)$, as the entity derived by stripping all type annotations from any skeleton object.

This pattern typology captures all the pre-runs that, subsequent to the initial segments from $Trips^{\gamma}_{D^{\triangle}}$, proceed from the potential final tree-stack forms and then simulate the designated product rPTSA according to the interaction sequence $D_{\times\mathcal{D}}$.

Inductive Tree-Stack Forms We now inductively examine the possible tree-stack forms induced by a mixed pattern $v = TripsOrUp^{\gamma}_{[D^{\triangle}|D_{\times\mathcal{D}}]}$, denoted $\mathsf{Ts}(v)$ as a set of pairs (m, f), where $m \in \mathcal{M}$ represents a local memory and f is a mapping from each stack symbol γ in Γ to a mixed pattern $TripsOrUp^{\gamma}_{[-|-]}$. Function f encodes potential upward mixed patterns, thereby providing an inductive characterisation between these mixed patterns. For every pair $(-, f) \in \mathsf{Ts}(v)$, when v represents a mixed *Up* pattern, it is required that exactly one $\gamma \in \Gamma$ corresponds to a mixed *Up* pattern, while all others correspond to mixed *Trips* patterns; for a mixed *Trips* pattern, all values of f should be mixed *Trips* patterns.

This set $\mathsf{Ts}(v)$ essentially describes the tree form and the future execution within and above this node after executing the skeleton $D^{\triangle}$ – for each pair (m, f), the local memory m describes the local memory of the current node, and f encodes the information of the tree form and the future execution for each direction γ.

The formal definition of $\mathsf{Ts}(v)$ proceeds as follows: Let $SynPaths_{(\gamma,D)}$ denote the

union of all $\mathit{SynPaths}^{\mathbb{D}}_{(\gamma,D)}$ across all $\mathbb{D}$, and $D^{\triangle}$ be the following seqeuence

$$[q_1 \overbrace{\ \ \ q'_1}^{\vartheta_1}, \ldots, q_n \overbrace{\ \ \ q'_n}^{\vartheta_n}]$$

such that $D_{\times\mathcal{D}}$ be that $[(q_{n+1}, -), \ldots, (q_m, -)]$. Further define $D_o = \mathsf{base}(D^{\triangle})$, $D_p = [q_{n+1}, \ldots, q_m]$, and $D = D_o +\!\!+ D_p$. For each synchronisation path $\pi \in \mathit{SynPaths}_{(\gamma,D)}$, denote π_1 as its prefix such that $\pi_1 \restriction \Downarrow\!= D_o$ and the remainder postfix as π_2 so that $\pi_1 +\!\!+ \pi_2 = \pi$.

Then, identify all functions h such that it is a corresponding upward skeleton assignment function of π_1 such that $\mathsf{DownSkel}_{\Downarrow}(\pi_1, h) = D^{\triangle}$. Besides, identify all product synchronisation paths $\pi_{\times\mathcal{D}}$ with the downward interface being $D_{\times\mathcal{D}}$ such that, when all states of $\mathcal{D}$ are removed from it, π_2 is obtained.

With the stuff identified, let the last local memory of π_1 be m (or m_{init} if π_1 is empty), then a pair (m, f) exists in $\mathsf{Ts}(v)$, and for each $\gamma \in \Gamma$, $f(\gamma)$ is the mixed pattern $\mathit{TripsOrUp}^{\gamma}_{[h(\gamma)|\pi_{\times\mathcal{D}}\restriction_{\gamma}]}$.

Probable Mixed Patterns Notably, not every mixed pattern necessarily corresponds to a valid tree-form that behaves as anticipated. We describe those with at least one actual tree-form possessing a non-zero probability of fulfilment by the inductive descriptions as *probable mixed patterns*.

As the base case, we consider all mixed patterns of the form $\mathit{TripsOrUp}^{\gamma}_{[\varepsilon|D_{\times\mathcal{D}}]}$ with a weight greater than zero as probable. These patterns, devoid of any "history" of skeleton interaction sequences, are essentially just $\mathit{TripsOrUp}^{\gamma}_{D_{\times\mathcal{D}}}$. The decidability of their weights being greater than zero is established using previously introduced methodologies for pure ω-regular model checking.

In the inductive scenario, a mixed pattern v is probable if and only if there exists an element $(-, f)$ in $\mathsf{Ts}(v)$ such that for all γ, $f(\gamma)$ is probable. We subsequently define $\mathsf{Ts}'(-)$ for all mixed patterns v, stipulating that if v is not probable, then $\mathsf{Ts}'(v) = \emptyset$; otherwise, $\mathsf{Ts}'(v)$ includes all elements $(-, f)$ where the range of f comprises solely probable mixed patterns.

It is then straightforward to deduce that for each probable mixed pattern v, any actual tree-form constructed by recursively invoking $\mathsf{Ts}'(-)$ for each node until reaching the base case of probable patterns, has a non-zero probability of proceeding according to the $D_{\times\mathcal{D}}$ component at each node.

More specifically, we may regard the content of $\mathsf{Ts}'(v)$, associated with the pattern v, as the set of right-hand sides (RHS) of the production rules linked with v. These rules are used for constructing a tree form. Formally, we say a tree t is *produced* by a mixed pattern $v = \mathit{TripsOrUp}^{\gamma}_{[D^{\triangle}|D_{\times\mathcal{D}}]}$, where $D^{\triangle} \neq \varepsilon$, if and only if: (1) there exists a pair $(m, f) \in \mathsf{Ts}'(v)$ such that $t(\varepsilon) = m$; and, (2) for all $\gamma' \in \Gamma$, let $f(\gamma')$ be $v' = \mathit{TripsOrUp}^{\gamma'}_{[D^{\triangle\prime}|D'_{\times\mathcal{D}}]}$, then, if $D^{\triangle\prime} = \varepsilon$, $[\gamma'] \notin \mathrm{dom}(t)$; otherwise, the sub-tree from the path $[\gamma']$ is produced by v'.

Following this construction, we derive the following two conclusions.

Lemma 5.49. *A tree t is produced by a pattern $TripsOrUp^{\gamma}_{[D^{\triangle}|D_{\times\mathcal{D}}]}$ if and only if there exists at least one pre-run $\vec{\mu}$ in the skeleton pattern $Trips^{\gamma}_{D^{\triangle}}$ such that $\vec{\mu}$ ends with a configuration whose tree-stack t' has the sub-tree from the path $[\gamma]$ being exactly t.*

Proof Sketch. The proof follows directly from the construction. For every node in t, there must be a proper synchronisation path that links the node to its child nodes. □

Further, we have that after the tree is produced, the cohesive pre-runs actually using this tree is indeed accepted by $\mathcal{D}$.

Lemma 5.50. *Given a tree t produced by a pattern $Up^{\gamma}_{[D^{\triangle}|D_{\times\mathcal{D}}]}$, consider the set S of cohesive infinite pre-runs $\vec{\mu}$ that: (1) start from a specific configuration with stack pointer $[\gamma]$ and suprapointer sub-tree being t; (2) have interface $D_{\times\mathcal{D}}$. Then S has a non-zero probability of being accepted by $\mathcal{D}$.*

Proof Sketch. This conclusion follows directly from the definition. For each internal node, there must be possible synchronisation paths linking this and its child nodes. At the leaves, each $\mathcal{D}$-observed *Up* pattern is guaranteed by probableness to have a non-zero probability of being accepted by $\mathcal{D}$. □

Proving Lemma 5.30 We are now equipped to present a proof for the central theorem of this part.

Proof (Lemma 5.30). By the discussions above, it follows that for a skeleton *Up* pattern $Up^{\gamma}_{D^{\triangle}+\!\!+[(q,\tau)]}$, the condition for having a weight greater than zero is equivalent to the pattern $Up^{\gamma}_{[D^{\triangle}|(q,\mathcal{A}_{init})]}$ being probable. This probability is determined based on the DRA representing φ_τ, with $\mathcal{A}_{init}$ as its initial state.

To see this, by Lemma 5.49, we can construct a proper tree-stack t, ended in some pre-run $\vec{\mu}$ with the interaction sequence $D^{\triangle}$. Then, by Lemma 5.50, the pre-runs of the pattern $Up^{\gamma}_{[(q,\mathcal{A}_{init})]}$, starting from any constructed tree-stack t, also have a non-zero probability of being accepted by $\mathcal{D}_\tau$, which by construction, corresponds to having type τ.

Note that the $\mathcal{D}$ states in the $\mathcal{D}$-observed patterns starting from any configuration are not affecting the behavior of the original rPTSA by construction. Consequently, the pattern $Up^{\gamma}_{[q]}$ starting also from the tree-stack t will have a one-one correspondence with $Up^{\gamma}_{[(q,\mathcal{A}_{init})]}$. This concludes that the pattern $Up^{\gamma}_{[q]}$ starting from t will also have probability > 0 of being accepted by φ_τ, hence having type τ.

Similarly, for the skeleton *Trips* patterns $Trips^{\gamma}_{D^{\triangle}}$, this is equivalent to asking whether the mixed pattern $Trips^{\gamma}_{[D^{\triangle}|\varepsilon]}$ is probable based on an arbitrary DRA $\mathcal{D}$.

Given the construction, the probability of the mixed patterns is decidable. Therefore, for every skeleton pattern v, whether $wt(v) > 0$ is decidable. □

The future is not fixed, but rather a myriad of possibilities branching out from every moment. Each choice, each action, solidifies one path while leaving the others to fade into the shadows of what might have been.

— Henri Bergson, Creative Evolution

6 Branching-Time Model Checking

Contents

Branching-time model checking broadens the scope of verification by considering not only single linear-time trajectories but also the full spectrum of possible future executions. This kind of property is essential for advanced system analysis and verification, as it allows for a more comprehensive examination of how systems behave under various possible scenarios. In this chapter, we address the problem of branching-time model checking. Specifically, we aim to demonstrate the following theorem.

Theorem 6.1. *Given a* regular *valuation ν, an rPTSA $\mathcal{A}$, and a* qualitative *PCTL* formula Φ, the problem of whether $\mathcal{A} \models_\nu \Phi$ is decidable.*

As an overview, we approach this problem by utilising the recursive closure property of various operations on tree automata, which enables us to decompose the formula and solve each sub-formula individually before constructing the overall result through tree automata operations.

Within this procedure, the most challenging case involves resolving the formula that represents a qualitative assertion about an LTL formula. For this purpose, we rely on the global model checking algorithm developed in the previous chapter.

However, it is important to note that the developed algorithm applies to *simple* valuations, whereas we are dealing with *regular* valuations.

This necessitates a *simulation* procedure that converts the problem into a model checking problem with simple valuation and subsequently translates the result back to the original regular valuation.

In outline, in this chapter, we first present the overall algorithm, which progressively reduces the problem to global LTL model checking with simple valuation. The final gap to address is the simulation procedure, which is subsequently covered in the remainder of the chapter.

Notational Conventions Throughout the chapter, we fix a specific rPTSA $\mathcal{A}$, a regular valuation ν and a qPCTL* formula Φ. To ease exposition, we again assume that there is NO termination transition in $\mathcal{A}$. Also, we require that the $\mathcal{A}$ be *globally* restricted by a restriction number k.

6.1 Model Checking Procedure

To begin with, let us provide a constructive solution to the model checking problem. Specifically, we propose the following theorem, from which the global model checking theorem can be directly derived.

Theorem 6.2. *Given an rPTSA $\mathcal{A}$, a qPCTL* state formula Φ, and a regular valuation ν of $\mathcal{A}$, there exists a TA $\mathcal{T}$ such that: $[\![\Phi]\!]^{\mathsf{s}}_{\nu} = \mathcal{C}(\mathcal{T})$.*

Given this theorem, the decidability of the model checking problem reduces to determining whether $\mathsf{tree}(\mathscr{c}_{init})$ is accepted by the constructed TA $\mathcal{T}$. We then firstly provide a proof overview, which relegates the problem to a regular global model checking problem for LTL; and then we further decompose the problem to a simulation procedure and the simple global model checking.

6.1.1 Proof Overview

Our proof of the theorem proceeds via induction on the structure of Φ. The simple cases are addressed as follows:

$\Phi = \mathsf{p}$
: Given that the valuation is regular, a TA $\mathcal{T}$ for $\nu(\mathsf{p})$ can be found, which provides the result.

$\Phi = \Phi_1 \wedge \Phi_2$
: Using the induction hypothesis, we construct $\mathcal{T}_1$ and $\mathcal{T}_2$ such that $\mathcal{C}(\mathcal{T}_1) = [\![\Phi_1]\!]^{\mathsf{s}}_{\nu}$ and $\mathcal{C}(\mathcal{T}_2) = [\![\Phi_2]\!]^{\mathsf{s}}_{\nu}$. We then construct the automaton $\mathcal{T}_1 \cap \mathcal{T}_2$, where $\cap$ represent the standard intersection operation of tree automata.

$\Phi = \neg\Phi'$
: Again, using the induction hypothesis, we construct $\mathcal{T}$ such that $\mathcal{C}(\mathcal{T}) = [\![\Phi']\!]^{\mathsf{s}}_{\nu}$.

The result is then obtained by taking the complement automaton of $\mathcal{T}$. These are standard operations of tree automata.

We then address the more complex case: when Φ is of the form $\mathbb{P}^{=1}[\varphi]$.

In this cases, we consider each sub-state-formula in Φ, ranged over by Φ'. For each Φ', we define a unique fresh atomic proposition $\mathsf{p}_{\Phi'}$ (that is unique for each Φ' and is not presented in $\mathbb{A}$). We construct a new formula $\hat{\Phi}$ by substituting each Φ' in Φ with the corresponding new atomic proposition $\mathsf{p}_{\Phi'}$. Subsequently, we construct a new valuation $\hat{\nu}$ by updating ν with a new mapping for each $\mathsf{p}_{\Phi'}$, mapping to $[\![\Phi']\!]^{\mathsf{s}}_{\nu}$. By the induction hypothesis, for each Φ', a DTA $\mathcal{T}_{\Phi'}$ such that $\mathcal{C}(\mathcal{T}_{\Phi'}) = [\![\Phi']\!]^{\mathsf{s}}_{\nu} = \hat{\nu}(\mathsf{p}_{\Phi'})$, can be constructed. Thus, $\hat{\nu}$ remains a regular valuation.

The above constructions essentially reduce the problem to a global qualitative LTL model checking problem, which is then handled by the following lemma.

Lemma 6.3 (Global Qualitative LTL Model Checking)**.** *Given an rPTSA $\mathcal{A}$, an LTL formula φ, and a regular valuation ν over $\mathcal{A}$, there is a TA $\mathcal{T}$, such that:*

$$\mathcal{C}(\mathcal{T}) = \left\{ c \in \mathcal{C} \;\middle|\; \mathbb{P}\left[\left\{ u \in \mathsf{InfTraj}(c) \;\middle|\; u \models_{\nu} \varphi \right\}\right] = 1 \right\}$$

Thus, when $\Phi = \mathbb{P}^{=1}[\varphi]$, by applying Lemma 6.3, we construct a TA $\mathcal{T}$ such that:

$$\mathcal{C}(\mathcal{T}) = \left[\!\!\left[\widehat{\mathbb{P}^{=1}[\varphi]} \right]\!\!\right]^{\mathsf{s}}_{\hat{\nu}} = [\![\mathbb{P}^{=1}[\varphi]]\!]^{\mathsf{s}}_{\nu}$$

In this case, $\mathcal{T}$ is the desired result.

Given this proof structure, our remaining task is to prove Lemma 6.3. This lemma is the focus of the rest of this work — once it is resolved the whole theorem is proved.

6.1.2 Proving Model Checking with Regular Valuation

Then, we proceed to address Lemma 6.3. We further decompose the regular global model checking into two distinct parts: simulation and simple global model checking.

Formally, with a regular valuation ν and an rPTSA $\mathcal{A}$, we construct a simulated rPTSA, denoted $\check{\mathcal{A}}$, along with a corresponding *simple* valuation $\check{\nu}$. This construction facilitates the existence of a partial *surjective* mapping $\psi : \mathcal{C}_{\check{\mathcal{A}}} \rightharpoonup \mathcal{C}_{\mathcal{A}}$. For any configuration $\check{c}$ within $\mathrm{dom}(\psi)$, it will be established that $\psi(\check{c}) \models_{\nu} \Phi \iff \check{c} \models_{\check{\nu}} \Phi$, applicable to all qualitative LTL formulas Φ (specifically, $\Phi = \mathbb{P}^{=1}[\varphi]$).

The proof structure for regular global model checking encompasses the following three main stages:

1. Construction of the simulation rPTSA $\check{\mathcal{A}}$ and the simulated simple valuation $\check{\nu}$.

2. Execution of simple global model checking to derive $\check{\mathcal{T}}$, which ensures $\mathcal{C}(\check{\mathcal{T}}) = [\![\Phi]\!]^{\mathsf{s}}_{\check{\nu}}$.

3. Formation of the target $\mathcal{T}$ from $\check{\mathcal{T}}$ such that $\mathcal{C}(\mathcal{T}) = \psi(\mathcal{C}(\check{\mathcal{T}}))$. This implies that for each $\mathscr{c} \in \mathcal{C}_{\mathcal{A}}$, the following equivalences are valid:

$$\begin{aligned} & \mathscr{c} \models_{\nu} \Phi \\ \iff & (\exists \check{\mathscr{c}} \in \psi^{-1}(\mathscr{c}) \text{ with } \check{\mathscr{c}} \models_{\check{\nu}} \Phi) \\ \iff & (\exists \check{\mathscr{c}} \in \psi^{-1}(\mathscr{c}) \text{ with } \check{\mathscr{c}} \in \mathcal{C}(\check{\mathcal{T}})) \\ \iff & \mathscr{c} \in \mathcal{C}(\mathcal{T}) \end{aligned}$$

Accordingly, the primary objectives in regular global model checking are: (1) Formulating the simulation triple $(\check{\mathcal{A}}, \check{\nu}, \psi)$ for $\mathcal{A}$ and ν to satisfy the aforementioned criteria, and devising an algorithm to construct $\mathcal{T}$ from $\check{\mathcal{T}}$; (2) Proving simple global model checking for $\check{\mathcal{A}}$ and $\check{\nu}$ to generate $\check{\mathcal{T}}$.

The latter task has been handled by the simple global model checking algorithm established in the previous chapter. We are now able to concentrate on constructing the simulation triple and deriving $\mathcal{T}$ from $\check{\mathcal{T}}$.

6.2 Simulating Regular Valuation

Then we delve into the details of the simulation procedure. The simulation aims to emulate the acceptance of each automaton $\mathcal{T}_i$ in an *on-the-fly* manner. This involves traversing the tree stack in the configurations of $\mathcal{A}$ during runtime. To the best of our knowledge, the simulation procedure is novel, as we could not find a readily available counterpart in the non-probabilistic literature for rTSA.

Although the technical construction of the simulation might be somehow dense, the intuition behind is not that hard. Hence, to introduce the overall construct better, we first present an overview of the idea behind, followed by an example to further illustrate it. Finally, we will present a formal account to the construction.

6.2.1 Simulation Overview

Let us begin with an overview of the intuition behind the simulation procedure, where the key lies in constructing the simulation rPTSA. Generally, building the simulation automaton utilises both its control state and the local memory at each node to track the execution of the given TAs. Specifically, each node on the tree-stack should maintain information about both the accepting states of its sub-trees and the "required states" imposed by its parent node.

Technically, let us first establish notations for the simulation process throughout the following discussion. We consider an rPTSA $\mathcal{A} = (\mathcal{Q}, \mathcal{M}, \Gamma, \Delta, \mathscr{q}_{init}, \mathscr{m}_{init})$ and a regular valuation ν.

We also denote the finite set of atomic formulas as $\mathbb{A} = \{\mathsf{p}_1, \ldots, \mathsf{p}_n\}$. Using the regular valuation, we construct automata $\boxed{\mathcal{T}_1} = (\boxed{T_1}, -, -, \boxed{\mathscr{T}_1}, T^{(1)}_{init}), \ \ldots, \ \boxed{\mathcal{T}_n} = (\boxed{T_n}, -, -, \boxed{\mathscr{T}_n}, T^{(n)}_{init})$. Here, $\mathcal{C}(\mathcal{T}_i) = \nu(\mathsf{p}_i)$ for each $i \in \{1, \ldots, n\}$. As these automata describe configurations of $\mathcal{A}$, they share the index set Γ and the alphabet $\Sigma^{\mathcal{A}}$, which are omitted in the above definitions.

We are now prepared to present the simulation concept. During the simulation, our ultimate objective is to maintain, *for each node*, a structure called the *simulation map* σ. This map associates each direction in $\mathcal{K}$ with a vector of sets of tree automaton states $\vec{T} = [T'_1, \ldots, T'_n]$, where $T'_i \subseteq T_i$. Intuitively, σ serves a dual purpose. Firstly, for each stack symbol γ, in each dimension i of the vector $\sigma(\gamma)$, let $T'_i = \sigma(\gamma)[i]$. This set represents *all the accepting states of the sub-tree in direction γ above this node* with respect to the automaton $\mathcal{T}_i$. Secondly, for each dimension i, the set $\sigma(\Downarrow)[i]$ contains the "required states" that intuitively indicate *if any of these states is accepting for the sub-tree from the current node, then the entire configuration is accepted* by $\mathcal{T}_i$.

Thus, the configuration (q, t, ρ) is accepted by $\mathcal{T}_i$, iff there exists a rule $(t, (q, t(\rho)), f) \in \mathcal{T}_i$ such that: $t \in \sigma(\Downarrow)$ and for each γ, $f(\gamma) \in \sigma(\gamma)$. This is because, from the intuitive guarantee provided by $\sigma(\gamma)$ for each γ, it follows that the sub-tree at branch γ is accepted from $f(\gamma)$. By the definition of tree acceptance, the existence of the rule indicates that the entire sub-tree from the node pointed to by the stack pointer is accepted from t, which, in turn, implies that by the interpretation of $\sigma(\Downarrow)$, the entire configuration is accepted by $\mathcal{T}_i$.

It is crucial that this information is continually updated as the entire configuration evolves to maintain accuracy. However, a significant observation is that the information could NOT always be accurate.

To observe the inaccuracies, one can note that the information for $\Downarrow$ at the side branches typically changes as the configuration transitions — note to recall the definition of the main branch and side branches from Section 5.2.1. For instance, a change in the local memory at a node instantly affects all the required states of the side branches above this node, as set of the applicable rules for this node alters. However, since it is impossible to simultaneously modify all such information on the tree-stack for all side branches in an rPTSA — recall that our objective is to build a simulation rPTSA $\check{\mathcal{A}}$ to carry the information of the TAs, and that the transitions of an rPTSA only modify the node pointed at by the stack pointer. This implies that the information kept in the simulation map of a side branch node is almost always "outdated". Similarly, on the main branch, except for the node pointed at by the stack pointer, the upward information for the direction where the stack pointer is located is also outdated.

Nonetheless, these inaccuracies do not impede the overall simulation procedure, as our objective is to determine whether the entire configuration is accepted based on the current status. Thus, we only need to ensure that the simulation map at the node pointed at by the stack pointer is accurate and current, enabling us to make correct judgements — whether the mentioned rules exist to bridge between $\sigma(\Downarrow)$ and $\sigma(\gamma)$ for all γs. Notice that an interesting and helpful fact is that – *the direction for which the simulation map carries outdated information is always where the stack pointer is at or comes from.* By using the current control state of the configuration to carry the latest accurate information to *complement* the simulation map kept locally in the node, one can always obtain the accurate information to make correct judgements by simply using the status. This complemented simulation map is called the *updated* simulation map below. To avoid confusion, we explicitly

		ℓ	r
t_n	$\xrightarrow{m_\perp}$	t_n,	t_a
t_n	$\xrightarrow{m_\perp}$	t_a,	t_n
t_n	$\xrightarrow{m_\perp}$	t_n,	t_n
t_n	$\xrightarrow{m_X}$	t_n,	t_a
t_n	$\xrightarrow{m_X}$	t_a,	t_n
t_n	$\xrightarrow{m_X}$	t_n,	t_n
t_n	$\xrightarrow{m_3}$	t_ℓ,	t_a

		ℓ	r
t_ℓ	$\xrightarrow{(q_1,m_2)}$	t_∇,	t_r
t_ℓ	$\xrightarrow{m_X}$	t_ℓ,	t_∇
t_r	$\xrightarrow{m_1}$	t_a,	t_a
t_r	$\xrightarrow{m_2}$	t_∇,	t_r
t_r	$\xrightarrow{m_3}$	t_∇,	t_r
t_a	$\xrightarrow{m_X}$	t_a,	t_a
t_a	$\xrightarrow{\nabla}$		
t_∇	$\xrightarrow{\nabla}$		

*: Here, X ranges $1, 2$ and 3.

Figure 6.1: Rules of $\mathcal{T}_{123}$ in Example 6.4

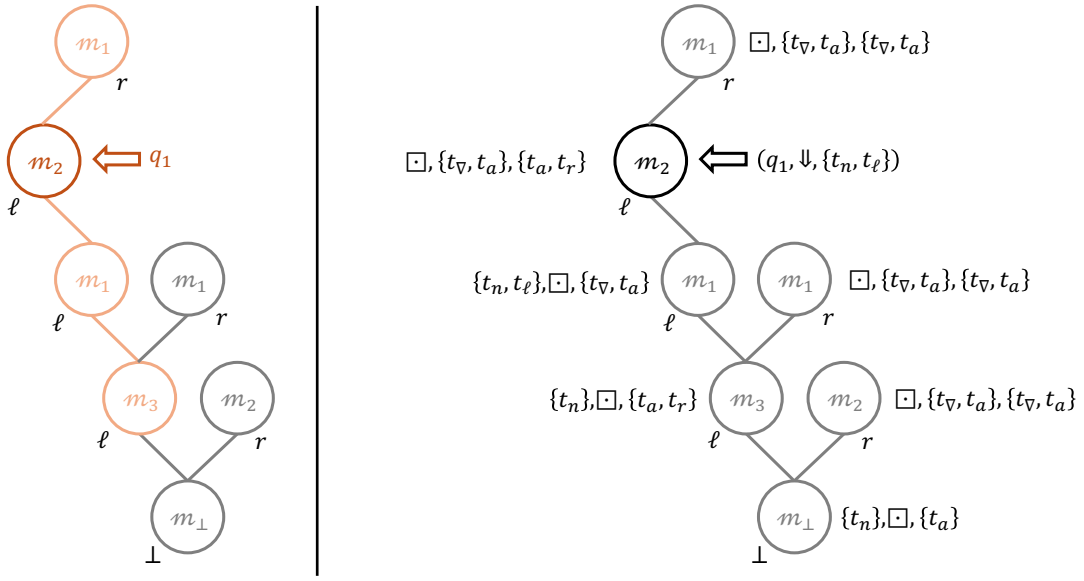

Figure 6.2: Configuration c_{123} and a Simulated Configuration $\check{c}_{123}$ based on $\mathcal{T}_{123}$

denote the outdated information with $\boxdot$ in the simulation maps σ.

6.2.2 Illustration Example

Let us now present an example to intuitively convey the general idea of the simulation, particularly how the (updated) simulation maps function and what a simulated configuration looks like.

Example 6.4. Consider an atomic proposition whose valuation is depicted by a tree automaton $\mathcal{T}_{123}$ for configurations represented as binary trees with branches ℓ and r. The rules for this automaton are shown in Figure 6.1, with the initial states being $\{t_n\}$. These rules follow the same format as in Example 3.21.

This automaton verifies whether the control state is q_1 and the stack pointer is positioned at a node on a single-left branch originating from an ancestor node with local memory m_3. The node itself must possess local memory m_2, with a single right branch leading to a node that has local memory m_1.

Formally, the state t_n, denoting a "normal" state, aims to identify such a structure. This structure could potentially reside on either the left or right branch, or on both, as reflected in the first six rules on the left rules list, where t_a represents a generic "any" state that trivially accepts all trees. Upon encountering the state m_3, denoted by the final line of rule at left list, it may signal the commencement of the structure, with the left side initiating the reading process using t_ℓ and the right side being read from t_a as an arbitrary structure.

Subsequently, the rules for t_ℓ attempt to read this structure, focusing initially on a tree with only a left branch. This is captured by the second rule on the right list, which necessitates that the right branch be empty, technically accepted from t_∇, which only has a rule accepting ∇. When the node pointed to by the stack pointer is reached, it checks if the control state is q_1 and whether the local memory is m_2, as indicated by the first rule on the right list that accepts (q_1, m_2). If these conditions are met, further inspection is required to determine if this node has a single right-branch tree extending all the way to a node with local memory m_1, which is the objective of t_r.

Finally, in t_r, until the local memory m_1 is encountered, the tree should continue to grow solely on the right list, as depicted by the fourth and fifth rules on the right list. Once m_1 is read, the third rule on the right list allows it to accept any further structure.

The left part in Figure 6.2 displays a configuration, $\mathscr{c}_{123}$, that satisfies the property described by $\mathcal{T}_{123}$, with the coloured structure visualises the intuition of $\mathcal{T}_{123}$ above.

On the right side of the figure, we display a simulated configuration $\check{\mathscr{c}}_{123}$ as a simulation of $\mathscr{c}_{123}$, carrying the information of $\mathcal{T}_{123}$ with the simulation map for each node attached alongside, aligning with the intuitive understanding introduced earlier. The simulation maps are represented as a sequence displaying the values for $\Downarrow$, ℓ, and r, respectively. At every node, an element marked $\boxdot$ appears in the simulation map, signifying that the information at that position is "outdated" and should not be considered, as explained in the overview.

Specifically, we assume that the stack pointer of the simulation comes from below, carrying the required states for $\Downarrow$, complementing the simulation map at the node.

Let us delve deeper into the rationale of the construction of $\check{\mathscr{c}}_{123}$ from $\mathscr{c}_{123}$, especially regarding the (updated) simulation map for each node. In the explanation, we denote the simulation map at a node at route ρ as σ_ρ. Especially, $\sigma_{\ell\ell\ell}$ denotes the *updated* map, namely $\{t_n, t_\ell\}, \{t_\nabla, t_a\}, \{t_a, t_r\}$.

We shall first discuss the values excluding $\Downarrow$ by examining each node individually in a (visually) top-down and topologically sorted manner. These values, as mentioned, intuitively corresponds to all the accepting states of the sub-tree at

the corresponding direction.

- For all the leaf nodes (at r, ℓr, $\ell\ell\ell r$), as only t_∇ and t_a can accept ∇, the values for both directions ℓ and r are $\{t_\nabla, t_a\}$. Since these nodes are all in the side branches, the value for $\Downarrow$ at these nodes should be marked as outdated by $\boxdot$.
- Next, let us consider the node at $\ell\ell\ell$. For this node, as its left is empty, $\sigma_{\ell\ell\ell}(\ell) = \{t_\nabla, t_a\}$ as previously discussed. On its right, the child node possesses local memory m_1 and a simulation map $\boxdot, \{t_\nabla, t_a\}, \{t_\nabla, t_a\}$. We could deduce that to accept this tree, only the rules $t_a \xrightarrow{m_1} t_a, t_a$ and $t_r \xrightarrow{m_1} t_a, t_a$ are applicable, thus, indicating that $\sigma_{\ell\ell\ell}(r) = \{t_a, t_r\}$.
- Now, we examine the node at $\ell\ell$. For this node, the right accepting set is trivial, $\{t_\nabla, t_a\}$, as discussed. The left accepting set is outdated because the stack pointer lies above at this branch.
- Similarly, the left accepting set of the node at ℓ is also outdated. In this node, as discussed for the node at $\ell\ell\ell$, the right value is $\{t_a, t_r\}$.
- For the bottom node, then, again, $\sigma_\perp(\ell)$ is outdated and hence represented by $\boxdot$. While for $\sigma_\perp(r) = \{t_a\}$, this is because the node at r, unlike those at ℓr and $\ell\ell\ell r$, has local memory m_2. Hence, there is only one rule $t_a \xrightarrow{m_2} t_a, t_a$ applicable.

Subsequently, we discuss the values for $\Downarrow$ on the main branch in a (visually) bottom-up manner. These states are intuitively the required states from below meaning that if any of the states are accepting, the whole configuration is accepted.

- Firstly, the root has $\sigma_\perp(\Downarrow)$ as $\{t_n\}$, which is fixed by definition.
- Then, for the node at ℓ, the information is computed from $\perp$ such that, for the entire tree to be accepting, only the rule $t_n \xrightarrow{m_\perp} t_n, t_a$ is applicable. Hence, the information $\sigma_\ell(\Downarrow)$ can only be $\{t_n\}$.
- The above situation also applies to the node at $\ell\ell$, with the additional rule $t_n \xrightarrow{m_3} t_\ell, t_a$ also applicable at the node ℓ. Hence, we have $\sigma_{\ell\ell}(\Downarrow) = \{t_n, t_\ell\}$.
- For $\sigma_{\ell\ell\ell}(\Downarrow)$, note that the two rules applicable to the node at $\ell\ell$ are: $t_n \xrightarrow{m_1} t_n, t_a$ and $t_\ell \xrightarrow{m_1} t_\ell, t_\nabla$.

Finally, we confirm that the entire configuration is accepted by $\mathcal{T}_{123}$, because the rule $t_\ell \xrightarrow{(q_1, m_2)} t_\nabla, t_r$ exists, with $t_\ell \in \sigma_{\ell\ell\ell}(\Downarrow)$, $t_\nabla \in \sigma_{\ell\ell\ell}(\ell)$, and $t_r \in \sigma_{\ell\ell\ell}(r)$. On the other hand, if we change the local memory of the node at $\ell\ell\ell$ to m_1, while the simulation maps are all not changed, the entire configuration is then not accepted as there is no combination of $t_\Downarrow \in \sigma_{\ell\ell\ell}(\Downarrow)$, $t_\ell \in \sigma_{\ell\ell\ell}(\ell)$, and $t_r \in \sigma_{\ell\ell\ell}(r)$ such that there is a rule $t_\Downarrow \xrightarrow{(q_1, m_1)} t_\ell, t_r$ in $\mathcal{T}_{123}$.

Remark 6.5. It is important to note that $\check{c}_{123}$ is not the only potential simulated configuration of c_{123} with respect to $\mathcal{T}_{123}$. In fact, several combinations for updating the simulation map are permissible. For instance, in the case of c_{123}, the following two configurations are both valid in simulating the runs of $\mathcal{T}_{123}$: (1) a configuration that modifies the simulation map of $\check{c}_{123}$ for the node $\ell\ell\ell$ to $\{t_n, t_\ell\}, \{t_\nabla, t_a\}, \boxdot$, with the control state modified to $(q_1, r, \{t_a, t_r\})$; or alternatively, (2) a configuration that employs the combination of $\{t_n, t_\ell\}, \boxdot, \{t_a, t_r\}$ and $(q_1, \ell, \{t_\nabla, t_a\})$. [1]

Both configurations are valid because they accurately encode the running information of $\mathcal{T}_{123}$ into the configuration c_{123}, enabling correct judgements about acceptance based on the current status. These valid configurations are referred to as "consistent" configurations below.

6.2.3 Simulation Construction

Given the intuition provided above, we then proceed to formally construct the simulation rPTSA $\check{\mathcal{A}} := (\check{\mathcal{Q}}, \check{\mathcal{M}}, \Gamma, \check{\Delta}, \check{q}_{init}, \check{m}_{init})$, where:

- The set $\check{\mathcal{Q}}$ is defined as:
$$\mathcal{Q} \times \left(\mathcal{K} \times \prod_{i=1}^{n} 2^{T_i}\right)$$
Thus, elements are in the form $(q, \kappa, \vec{T})$, with $\vec{T} = [T'_1, \ldots, T'_n]$ and each $T'_i \subseteq T_i$. Intuitively, the control states now also keep the information about: (1) where the control comes from – indicating by κ; and, (2) what is the information to be updated to the local memory for this direction κ, namely $\vec{T}$.

- For $\check{\mathcal{M}}$, it contains *all* pairs of the form (m, σ), where σ maps directions in $\mathcal{K}$ to either the special marker $\boxdot$ for outdated information, or a vector $\vec{T} = [T'_1, \ldots, T'_n]$, where $T'_i \subseteq T_i$. Additionally, exactly one element in $\mathcal{K}$ must map to $\boxdot$ in σ.

  This construction equips the local memory with the simulation map σ. As previously discussed, σ encapsulates the *accepting states* for each $\mathcal{T}_i$ in each direction of Γ and records the required states from below for $\Downarrow$.

  Outdated information appears in the direction where the stack pointer is located or comes from, the latter occurring when the stack pointer points to the current node and carries information from the direction the pointer comes from. Specifically, if the stack pointer is above in some direction γ, the accepting states continually change as the automaton progresses; thus, the information at $\sigma(\gamma)$ cannot be guaranteed to correspond precisely to the accepting states of the sub-tree. The same principle applies to $\Downarrow$ if the stack pointer is neither within nor above this node. It is crucial to note that such a direction must exist; hence, σ must have exactly one direction mapped to $\boxdot$.

[1] This might appear to be somewhat nonsensical — the left branch has not been created, yet the state intuitively claims to be coming back from this branch. However, as the simulation is working in conjunction with global model checking, it is valid to consider such configurations.

- The stack symbol set for $\check{\mathcal{A}}$ remains identical to that of $\mathcal{A}$, namely, Γ.
- The initial control state $\check{q}_{init}$ is set as $(q_{init}, \Downarrow, \boxed{\vec{T}_{init}})$, where the initial TA state is given by simply the vector of all initial states of the TAs for the atomic propositions:
$$\vec{T}_{init} = [T^{(1)}_{init}, \ldots, T^{(n)}_{init}]$$
- The default local memory $\check{m}_{init}$ is:
$$(m_{init}, \{\Downarrow \mapsto \boxdot\} \uplus \{\gamma \mapsto \boxed{\vec{T}_{acc}} \mid \gamma \in \Gamma\})$$
Here, $\vec{T}_{acc} = [T^{(1)}_{acc}, \ldots, T^{(n)}_{acc}]$, with each $T^{(i)}_{acc} \subseteq T_i$ containing states $t \in T_i$ with a rule $(t, \nabla, \emptyset) \in \mathcal{T}_i$.
This is intuitively because, when a node is created, its child nodes are guaranteed to be non-created, and therefore, they carry the special tree-top mark ∇ in the configuration tree. Consequently, for each sub-tree above, its accepting states must precisely be those with a rule to accept ∇ — which aligns exactly with the meaning of $T^{(i)}_{acc}$ for each atomic proposition. An example of a T_{acc} set can be seen above for $\mathcal{T}_{123}$ — namely, $\{t_\nabla, t_a\}$.

Hence, we are now utilising both the control states and the local memories to maintain the simulation maps. Notably, at the node pointed at by the stack pointer, the updated simulation map preserves the precise and accurate information required for the simulation.

Then, we are to construct the set $\check{\Delta}$ of simulation transition rules. We progress by considering for each rule $(q, m, g) \xrightarrow{p} op$ in Δ, and each $\check{q} \in \check{Q}$, $\check{m} \in \check{\mathcal{M}}$, where $\check{q} = (q, \boxed{\kappa}, \boxed{\vec{T}})$ and $\check{m} = (m, \boxed{\sigma})$ with the unique direction of σ that maps to $\boxdot$ being κ. We define $\boxed{\sigma'}$ as $\sigma[\kappa \mapsto \vec{T}]$.

σ' is the *updated* simulation map that is intuitively understood to represent the most current and accurate information about the tree's structure from both below and above. That is, for each $\gamma \in \Gamma$ and each i, each $\sigma'(\gamma)[i]$ precisely contains the accepting states for the sub-tree in direction γ above the current node, according to $\mathcal{T}_i$. While $\sigma'(\Downarrow)$ holds the states provided by the parent node of the current node, so that if any of the state is accepting, the whole configuration is accepted.

The rules in $\check{\Delta}$ are outlined as follows:

- For *down* operations $op = (q', \boxed{m'}, down)$, with $g \in \Gamma$, set $\check{q}'$ as $(q', g, \boxed{\vec{T}'})$. Here, $\vec{T}'$ is $[T'_1, \ldots, T'_n]$, where each T'_i includes tree states $\boxed{t} \in T_i$ linked to a rule $(t, m', \boxed{f}) \in \mathcal{T}_i$, ensuring for every $\gamma \in \Gamma$, $f(\gamma)$ is in $\sigma(\gamma)[i]$. Set $\check{m}'$ as $(m', \sigma'[\Downarrow \mapsto \boxdot])$, and then add the following rule to $\check{\Delta}$:
$$(\check{q}, \check{m}, g) \xrightarrow{p} (\check{q}', \check{m}', down)$$
This construction updates the local memory with the upcoming local memory and the freshest information, σ'. It ensures that the next control state $\check{q}'$

precisely represents the accepting states for the sub-tree beginning at the current node. Therefore, in $\check{q}'$, for each T'_i in $\vec{T}'$, the sub-tree initiating from the current node is accepted by a state $t \in T_i$ of $\mathcal{T}_i$ if and only if there exists a rule (t, m', f) for t, the newly updated m', and f such that for every $\gamma \in \Gamma$, $f(\gamma)$ is among the accepting states of the sub-tree of direction γ by $\mathcal{T}_i$, precisely what is defined in $\sigma'(\gamma)[i]$.

Finally, as the stack pointer moves *down* from this node, the required states in $\sigma'(\Downarrow)$ become outdated. Therefore, we must additionally update $\sigma'(\Downarrow)$ to $\boxdot$ in $\check{m}'$.

- For *up* operations $op = (q', \boxed{m'}, up_{\boxed{\gamma}})$, define $\check{q}'$ as $(q', \Downarrow, \vec{T}')$. Each set T'_i in $\vec{T}'_i$ is determined as the *smallest* set of the following. For each rule $(t', m', f) \in \mathcal{T}_i$ with $t' \in \sigma'(\Downarrow)[i]$ and for $\boxed{\gamma'}$ in $\Gamma \setminus \{\gamma\}$, $f(\gamma')$ is in $\sigma'(\gamma')[i]$, then include $f(\gamma)$ in T'_i. Set $\check{m}'$ as $(m', \sigma'[\gamma \mapsto \boxdot])$, and include in $\check{\Delta}$ the rule:

$$(\check{q}, \check{m}, g) \xrightarrow{p} (\check{q}', \check{m}', up_\gamma)$$

In this configuration, the local memory update reflects the new local memory m' and captures the latest information, σ'. Obtaining the new control state $\check{q}'$ is more complex. Here, the direction $\Downarrow$ indicates that the next transition is a control pass from below. In the new $\vec{T}' = [T'_1, \ldots, T'_n]$, each T'_i must contain the states necessary for acceptance in the sub-tree of direction γ. Essentially, if the sub-tree in direction γ is accepted by any state in T'_i, the entire configuration is accepted by $\mathcal{T}_i$. Formally, this implies that for each $t' \in \sigma'(\Downarrow)[i]$, and each rule $(t', m', f) \in \mathcal{T}_i$, $f(\gamma)$ need not to be directly in T'_i, but f must meet the acceptance conditions from *other* directions, ensuring the states are accepting for all these directions. Otherwise, even if $f(\gamma)$ is accepting from the sub-tree in direction γ, the entire configuration is not accepted by $\mathcal{T}_i$.

Finally, update $\sigma'(\gamma)$ in $\check{m}'$ to be $\boxdot$, as the stack pointer moves to this direction, and consequently, the information in this direction cannot be guaranteed to be accurate.

To understand the transition set more intuitively, let us consider the following example for performing operations on the simulated configuration $\check{c}_{123}$.

Example 6.6. Referring back to Example 6.4. We consider the two cases of operations to be performed on $\check{c}_{123}$. Let the tree-stack associated with $\check{c}_{123}$ be $\check{t}_{123}$, the simulation map of $\check{t}_{123}(\ell\ell\ell)$ be $\sigma_{\ell\ell\ell}$ and the updated map be simply σ.

We first examine an *up* operation. Suppose there is a rule in the rPTSA associated with c_{123}:

$$(q_1, m_2, \ell) \xrightarrow{p} (q_2, m_1, up_\ell)$$

For convenience, we denote the new path $\ell\ell\ell\ell$ as ℓ^4. In this scenario, the simulated maps at other nodes remain accurate, except for the node originally indicated by

the stack pointer. Both $\sigma(\Downarrow)$ and $\sigma(r)$ remain unaffected. However, by moving the stack pointer to ℓ^4, the information at $\sigma(\ell)$ becomes invalid and should therefore be assigned $\boxdot$. Let us now consider the updated simulation map σ' associated with the node at ℓ^4. As introduced in the motivating example, we have $\sigma'(\ell) = \sigma'(r) = \{t_\nabla, t_a\}$. The key element to examine next is $\sigma'(\Downarrow)$. By modifying the local memory of the node at $\ell\ell\ell$ to $\mathscr{m}_1$, the only applicable rule, considering $\sigma(\Downarrow)$ and $\sigma(r)$, is:

$$t_n \xrightarrow{\mathscr{m}_1} t_n, t_a$$

This is the sole rule where the left list $t_n \in \sigma(\Downarrow)$ and the right list $t_a \in \sigma(r)$. If, for example, we want to consider the left list state $t_\ell \in \sigma(\Downarrow)$, the only applicable rule for local memory $\mathscr{m}_1$ is:

$$t_\ell \xrightarrow{\mathscr{m}_1} t_\ell, t_\nabla$$

However, since $t_\nabla \notin \sigma(r)$, it does not apply here. Therefore, we conclude that $\sigma'(\Downarrow) = \{t_n\}$. This updated information is computed from the node at ℓ^4, and thus the corresponding rule in the simulation rPTSA should be: let $\check{\mathscr{m}}_1$ be $(\mathscr{m}_1, \sigma[\ell \mapsto \boxdot])$,

$$\Big((q_1, \Downarrow, \{t_n, t_\ell\}), (\mathscr{m}_2, \sigma_{\ell\ell\ell}), \ell\Big) \xrightarrow{p} \Big((q_2, \Downarrow, \{t_n\}), \check{\mathscr{m}}_1, up_\ell\Big)$$

Applying this rule results in a consistent configuration:

$$\Big((q_2, \Downarrow, \{t_n\}), \check{t}_{up}, \ell\ell\ell\Big)$$

Here, $\check{t}_{up}$ is defined as:

$$\check{t}[\ell\ell\ell \mapsto \check{\mathscr{m}}_1, \ell^4 \mapsto \check{\mathscr{m}}_{init}]$$

where $\check{\mathscr{m}}_{init}$ is defined as $(\mathscr{m}_{init}, \sigma_{init})$, with $\mathscr{m}_{init}$ being the initial local memory of the rPTSA associated with $\mathscr{c}_{123}$, and σ_{init} given by $\sigma_{init}(\Downarrow) = \boxdot$, $\sigma_{init}(\ell) = \sigma_{init}(r) = \{t_\nabla, t_a\}$. Note that this definition of σ_{init} is a proper instance of the mentioned $\vec{T}_{acc}$ as a singleton.

Similarly, we could consider the case of a *down* operation, for which, suppose there is a rule:

$$(q_1, \mathscr{m}_2, \ell) \xrightarrow{p} (q_2, \mathscr{m}_2, down)$$

The corresponding simulation rule would then be:

$$\Big((q_1, \Downarrow, \{t_n, t_\ell\}), (\mathscr{m}_2, \sigma_{\ell\ell\ell}), \ell\Big) \xrightarrow{p} \Big((q_2, \ell, \{t_a, t_r\}), (\mathscr{m}_2, \sigma[\Downarrow \mapsto \boxdot]), down\Big)$$

This is because, for the entire sub-tree from $\ell\ell\ell$ to be accepted, the applicable rules are:

$$t_r \xrightarrow{\mathscr{m}_2} t_\nabla, t_r \quad \text{and} \quad t_a \xrightarrow{\mathscr{m}_2} t_a, t_a$$

This information is conveyed by the control state $(q_2, \ell, \{t_a, t_r\})$, indicating that the transition originates from the left branch, for which the accepting states of the sub-tree on this branch are precisely $\{t_a, t_r\}$.

Simulation Consistency It is crucial to recognise that not all configurations of the simulated rPTSA $\check{\mathcal{A}}$ correspond to the simulation of configurations for $\mathcal{T}_1, \ldots, \mathcal{T}_n$. In particular, if the (updated) simulation maps encoded within the tree-stack and the control state of the configuration do not reflect the true information — which is likely in the context of global model checking, where configurations are not guaranteed to be reachable from the initial configuration and can be arbitrary — the simulation may fail to be accurate. To address this, we introduce the concept of *consistent* simulation configurations — they refer to configurations in which the (updated) simulation maps correctly reflect the intended information.

Moreover, to inductively define the consistency of a configuration, we first technically define *path consistency* to express that the (updated) simulation map at a specified node on the path is correct.

Definition 6.7 (Path Consistency)**.** The term *path consistency* refers to the following criteria. A simulation configuration $\check{c} = \left((q, \kappa, \boxed{\vec{T}}), \check{t}, \boxed{\rho}\right)$ is deemed consistent from a path $\boxed{\rho'} \in \mathrm{dom}(\check{t})$ if and only if it satisfies these conditions: Let $(\boxed{m}, \boxed{\sigma}) := \check{t}(\rho')$, and let $\boxed{\sigma'} := \sigma$ if $\rho' \neq \rho$; otherwise, let $\sigma' := \sigma[\kappa \mapsto \vec{T}]$. For every upward direction $\boxed{\gamma} \in \Gamma$, the following requirements must be met:

- If $\rho'\gamma \notin \mathrm{dom}(\check{t})$, then $\sigma'(\gamma)$ should equal $\vec{T}_{acc}$.
- If $\rho'\gamma \in \mathrm{dom}(\check{t})$, we first verify the configuration's consistency from $\rho'\gamma$. Subsequently, we determine $(\boxed{m_\gamma}, \boxed{\sigma_\gamma}) := \check{t}(\rho'\gamma)$ and proceed with a case analysis based on whether $\rho'\gamma$ is on the main branch:
  - If $\rho'\gamma \not\sqsubseteq \rho$, each $\sigma'(\gamma)[i]$ must exclusively include those $t \in T_i$ for which a rule $(t, m_\gamma, f) \in \mathcal{T}_i$ is present and for each direction $\boxed{\gamma'} \in \Gamma$, $f(\gamma')$ belongs to $\sigma_\gamma(\gamma')[i]$.
  - If $\rho'\gamma \sqsubseteq \rho$, implying $\rho' \sqsubseteq \rho$, we first let $\boxed{\sigma'_\gamma}$ be $\sigma_\gamma[\kappa \mapsto \vec{T}]$ if $\rho'\gamma = \rho$; otherwise, it remains σ_γ. Each $\sigma'_\gamma(\Downarrow)[i]$ should be the smallest set encompassing the following: For any $\boxed{t} \in \sigma(\Downarrow)$, if there is a rule $(t, m, f) \in \mathcal{T}_i$ such that for every $\boxed{\gamma'} \in \Gamma \setminus \{\gamma\}$, $f(\gamma')$ is a member of $\sigma(\gamma')[i]$, then $f(\gamma)$ is incorporated in $\sigma'_\gamma(\Downarrow)[i]$.

Additionally, if $\rho' = \varepsilon$, it is imperative that $\sigma'(\Downarrow)$ equals $\vec{T}_{init}$. Also, regarding the outdated information, we require that it exists precisely in the correct direction, indicating the position of the stack pointer — formally: (1) when $\rho' = \rho$, then $\sigma(\kappa) = \boxdot$; (2) when $\rho'\gamma \sqsubseteq \rho$ for some γ, then $\sigma(\gamma) = \boxdot$; and (3) when $\gamma' \not\sqsubseteq \rho$, then $\sigma(\Downarrow) = \boxdot$.

Given the definition above, the configuration $\check{c}$ is considered a *consistent configuration* if and only if it is path consistent from ε.

Constructing $\check{\nu}$ For a consistent configuration $\check{c}$, write its status $\mathsf{status}(\check{c})$ as $\Big((q, \kappa, \vec{T}), (m, \sigma), g\Big)$. Let σ' be $\sigma[\kappa \mapsto \vec{T}]$. We then define the simple simulated valuation $\check{\nu}$ such that a configuration belongs to $\check{\nu}(\mathsf{p}_i)$ if and only if there exists a state $t \in \sigma'(\Downarrow)[i]$ for which there is a rule $\Big(t, (q, m), f\Big)$ where, for every $\gamma \in \Gamma$, $f(\gamma)$ is in $\sigma'(\gamma)[i]$.

Constructing ψ The function ψ is essentially a *recovery* function, stripping away additional information from consistent configurations. Formally, the domain of ψ includes all consistent configurations. For any consistent configuration $\check{c} = \Big((q, -, -), \check{t}, \rho\Big)$, we define $\psi(\check{c}) := (q, t, \rho)$, where t is constructed as $\Big\{\rho' \mapsto m \;\Big|\; \rho' \in \mathrm{dom}(\check{t}), (m, -) = \check{t}(\rho')\Big\}$.

Property of ψ Our next goal is to demonstrate that ψ satisfies the aforementioned property.

Firstly, it is important to note that ψ is a surjective partial function:

Lemma 6.8 (Surjectiveness)**.** *For every configuration $c \in \mathcal{C}^{(\mathcal{A})}$, the set $\psi^{-1}(c)$ is non-empty.*

Proof. We could construct a simulated consistent configuration straightforwardly by computing an (updated) simulation map for each node of c, mimicking exactly the same way as shown for constructing $\check{c}_{123}$ from c_{123} in Section 6.2.2. □

Then, we establish a direct connection through the following lemma:

Lemma 6.9 (Atomic Simulation)**.** *For any consistent configuration $\check{c}$ and for all i, the relation $\check{c} \in \check{\nu}(\mathsf{p}_i) \iff \psi(\check{c})$ is accepted by $\mathcal{T}_i$ holds true.*

The proof of this lemma is straightforward. Additionally, it is important to note that the rules of $\check{\mathcal{A}}$ ensure the propagation of consistency.

Lemma 6.10 (Consistency Propagation)**.** *For any transition in $\check{\mathcal{A}}$, represented as $\check{c} \xrightarrow{\check{\delta}} \check{c}'$, if $\check{c}$ is consistent, then $\check{c}'$ remains consistent.*

The proof for this lemma involves a case analysis based on the operation of possible enabled rules. This analysis allows us to assert the next key property of the simulation triple.

Lemma 6.11. *For any consistent configuration $\check{c}$ and any qPCTL* (state) formula Φ, it holds that $\check{c} \models_{\check{\nu}} \Phi \iff \psi(\check{c}) \models_{\nu} \Phi$.*

Proof Sketch. The proof is conducted via an induction on Φ. The base case is supported by Lemma 6.9, while the inductive steps are straightforward. The proof for path formulas is facilitated by applying Lemma 6.10. □

6.2.4 Result Extraction from Simulation

We now focus on constructing $\mathcal{T}$ from $\check{\mathcal{T}}$ so that $\mathcal{C}(\mathcal{T}) = \psi(\mathcal{C}(\check{\mathcal{T}}))$, fulfilling the final result of the regular global model checking. We begin by assuming the existence of the fixed objects described in Section 6.2.1.

Given a simulated TA $\check{\mathcal{T}} = (\check{T}, \Gamma, \Sigma^{\check{\mathcal{A}}}, \mathscr{T}, \check{T}_{init})$, we construct the TA $\mathcal{T}$ as $\left(\check{T} \times \{\Downarrow, \Uparrow\} \times \prod_{i=1}^{n} 2^{T_i}, \Gamma, \Sigma^{\mathcal{A}}, \mathscr{T}, (\check{T}_{init}, \Uparrow, [T_{init}^{(1)}, \ldots, T_{init}^{(n)}])\right)$.

In this new state set, tagged information is included to maintain consistency. In particular, we identify two types of states:

- $(\check{t}, \Uparrow, \vec{T})$ represents the *main branch* information, indicating that the stack pointer is within or above this node. In this case, $\vec{T}$ refers to the required states such that if any state in $\vec{T}[i]$ is accepting, the whole configuration is accepted by $\mathcal{T}_i$. This corresponds precisely to the states of $\Downarrow$ in the (updated) simulation map.

- $(\check{t}, \Downarrow, \vec{T})$ represents the *side branches* information, indicating that the stack pointer is neither within nor above this node. Here, $\vec{T}$ denotes all the accepting states of the entire sub-tree from the current node downwards, corresponding to the information that must be retained for this branch in the (updated) simulation map *in the parent node*.

We proceed by performing a detailed case analysis for the rules in $\mathscr{T}$, examining each rule in $\check{\mathscr{T}}$ individually.

- Consider every rule of the form $(\check{t}, (m, \sigma), \boxed{f})$ in $\check{\mathscr{T}}$. In this case, we perform a further analysis of the stack configuration σ to discern whether the rule pertains to the main branch or to one of the side branches.

  - If $\sigma(\Downarrow) = \boxdot$, this indicates that the rule involves reading from one or more side branches. Let $\vec{T}$ be defined as follows: for each index i, $\vec{T}[i]$ includes precisely those TA states $t_i \in T_i$ for which there exists a rule of the form (t_i, m, f') such that, for every stack symbol $\gamma \in \Gamma$, the condition $f'(\gamma) \in \sigma(\gamma)[i]$ is satisfied. Consequently, the following rule is incorporated into $\mathscr{T}$:

    $$\left((\check{t}, \Downarrow, \vec{T}), m, \left\{\gamma \mapsto (f(\gamma), \Downarrow, \sigma(\gamma)) \,\middle|\, \gamma \in \Gamma\right\}\right)$$

  - On the other hand, if $\sigma(\Downarrow) \neq \boxdot$, there must exist a unique direction γ such that $\sigma(\gamma) = \boxdot$. This indicates that the stack pointer is positioned above the current context at the direction γ, which requires special handling.

    We then define a vector $\vec{T}$ by the following procedure. For each index i, the set $\vec{T}[i]$ is defined as the minimal set satisfying the condition that if a state t is present in $\sigma(\Downarrow)$ and there exists a corresponding rule $(t, m, f') \in \mathscr{T}_i$ such that, for every stack symbol $\gamma' \in \Gamma \setminus \{\gamma\}$, the

condition $f'(\gamma') \in \sigma(\gamma')$ is fulfilled, then the element $f'(\gamma)$ must be included in $\vec{T}[i]$.
Following this, the next rule is added to $\mathscr{T}$:

$$\Big((\check{t}, \Uparrow, \sigma(\Downarrow)), \mathit{m}, f''\Big)$$

where the function f'' is explicitly defined as:

$$f''(\gamma') := \begin{cases} (f(\gamma), \Uparrow, \vec{T}) & \text{if } \gamma' = \gamma \\ (f(\gamma'), \Downarrow, \sigma(\gamma')) & \text{if } \gamma' \neq \gamma \end{cases}$$

- Next, for every rule of the form $\Big(\check{t}, \big((\mathit{q}, \kappa, \vec{T}), (\mathit{m}, \sigma)\big), f\Big) \in \check{\mathscr{T}}$, there is a prerequisite that $\sigma(\kappa) = \boxdot$. In this context, let σ' be defined as $\sigma[\kappa \mapsto \vec{T}]$. Subsequently, the following rule is incorporated into $\mathscr{T}$:

$$\Big((\check{t}, \Uparrow, \vec{T}), (\mathit{q}, \mathit{m}), \{\gamma \mapsto (f(\gamma), \Downarrow, \sigma'(\gamma)) \mid \gamma \in \Gamma\}\Big)$$

- Finally, consider every rule of the form $(\check{t}, \nabla, \emptyset)$ in $\check{\mathscr{T}}$. In this case, the corresponding rule added to $\mathscr{T}$ is as follows:

$$\Big((\check{t}, \Downarrow, \vec{T}_{acc}), \nabla, \emptyset\Big)$$

This construction essentially transfers the information maintained within the rPTSA control states and the local memories to the TA states. As a result, the following target validity can be straightforwardly established.

Theorem 6.12. *From the construction, we have:* $\mathcal{C}(\mathcal{T}) = \psi(\mathcal{C}(\check{\mathcal{T}}))$.

Proof. The validity of the theorem can be observed from the fact that the construction effectively defines a surjective map g from $\check{\mathscr{T}}$ to $\mathscr{T}$. This mapping ensures that every rule in $\check{\mathcal{T}}$ corresponds uniquely to a rule in $\mathcal{T}$, and conversely, every rule in $\mathcal{T}$ has at least one corresponding source rule in $\check{\mathcal{T}}$. Moreover, the sets of source rules corresponding to distinct rules in $\mathcal{T}$ are disjoint.

With this mapping, for every configuration $\check{\mathscr{c}} \in \mathcal{C}(\check{\mathcal{T}})$, and for any rule r applied at each node of the tree $\mathsf{tree}(\check{\mathscr{c}})$, the rule $g(r)$ can be consistently applied to the corresponding node in the tree $\mathsf{tree}(\psi(\check{\mathscr{c}}))$.

Conversely, for any configuration $\mathscr{c} \in \mathcal{C}(\mathcal{T})$ and for any rule r applied at each node of the tree $\mathsf{tree}(\mathscr{c})$, one can construct a consistent configuration $\check{\mathscr{c}}$ by selecting any rule from $g^{-1}(r)$ to apply at each corresponding node and inserting the information of the (updated) simulation map from the source rule selected. The existence of such a rule is guaranteed by the surjectiveness of g, ensuring that every rule in $\mathcal{T}$ is mapped back to at least one rule in $\check{\mathcal{T}}$. Furthermore, the consistency of the constructed configuration is preserved by the properties of r as defined by the construction. □

Following the rationale established in the overview, the overall validity of the simulation, and consequently the branching-time model checking algorithm, are shown.

When we look about us towards external objects, and consider the operation of causes, we are never able, in a single instance, to discover any power or necessary connexion; any quality, which binds the effect to the cause, and renders the one an infallible consequence of the other. We only find, that the one does actually, in fact, follow the other. The impulse of one billiard-ball is attended with motion in the second. This is the whole that appears to the outward senses.

— David Hume
"An Enquiry Concerning Human Understanding"

7 Restriction Validation

Contents

The k-restriction is pivotal for analysability, distinguishing an rPTSA from a Turing machine. Therefore, further analysis of this property is crucial for a deeper understanding of the proposed automaton. In this chapter, we investigate the problem of deciding the restriction.

Specifically, we demonstrate that, given any PTSA, while it is undecidable whether the automaton is restricted, it is decidable whether a proposed number k is a true restriction for it. Specifically, we have:

Theorem 7.1. *Given a PTSA $\mathcal{A}$ and a natural number $k > 0$, it is decidable whether $\mathcal{A}$ satisfies (local / global) k-restriction.*

Throughout this chapter, we fix a specific PTSA $\mathcal{A} = (\mathcal{Q}, \mathcal{M}, \Gamma, \Delta, q_{init}, m_{init})$ and a restriction number $k > 0$ for validation. We also adopt the assumption from the model checking chapters (i.e., Chapters 5 and 6) that $\mathcal{A}$ has no termination transition. It is noteworthy that, by employing similar techniques as in previous chapters, one could construct a PTSA devoid of such transitions. This construction does not hinder the validation process, since in this scenario, the terminating runs are simply extended by an infinite run satisfying the 1-restriction.

We begin this chapter by addressing the problem of *local validation*, which investigates whether all *initial runs* satisfy the k-restriction. This serves as the

basis for the more general *global validation*, which examines whether all runs of $\mathcal{A}$ satisfy the k-restriction.

Finally, we provide a brief discussion on the undecidability of whether any PTSA is restricted in Remark 7.12. This completes the picture of restriction validation. Additionally, note that the following validation algorithms are applicable to non-probabilistic tree-stack automata, as discussed in Remark 7.13.

7.1 Local Validation

In this section, we present the local validation for restriction, starting by introducing the theoretical foundation the validation algorithm based on, in which an illustrative example is provided. Finally, we provide the algorithmic description.

7.1.1 Theoretical Foundation

We present the theories behind the local checking algorithm.

The Key Theorem

To begin with, we present the key theorem that guides the design of the algorithm for validating k.

To state the result, we first define the concept of *correctness*. We say that the declared k is a *(globally) correct* if and only if all runs comply with the k restriction. Otherwise, we say that k is a *(globally) incorrect*.

When restricting the runs under consideration to only initial runs, we define the *local (in)correctness*.

We are now ready to present the key theorem as follows.

Theorem 7.2. *Given a PTSA $\mathcal{A}$ with no terminating transition, a specified restriction number k is locally* incorrect *if and only if: $wt(Up^{\perp}_{[q_{init}]}) < 1$.*

Proof. By the requirement that the pattern definition must conform to the k-restriction, we can express $wt(Up^{\perp}_{[q_{init}]})$ as:

$$\mathbb{P}\left[\{u \in \mathsf{InfTraj}^{(\mathscr{M}(\mathcal{A}))}_{c_{init}} \mid u \text{ satisfies } k\text{-restriction}\}\right]$$

We now begin the proof with the forward direction. If k is locally incorrect, there exists an initial run where a node is visited from below more than k times. We consider the prefix of such a run up to the $(k+1)$st visit to this node. Any runs sharing this prefix will violate the k-restriction. Given the definition of the probability space, the set of paths leading to such a violation has a probability greater than 0 and is distinct from the set of initial runs adhering to the k-restriction. Thus, the probability of initial runs satisfying the k-restriction is less than 1.

Conversely, the reverse direction is more straightforward: if the set of initial runs not complying with the k-restriction has a non-zero probability, it implies their existence. □

To get a closer look at the intuition behind the theorem, let us see the following example.

Example 7.3. Consider the example PTSA depicted in Figure 7.1. Let q_{init} be q and m_{init} be m_1. This PTSA is simple and deterministic, adhering to the non-terminating assumption. Essentially, it begins by moving *up* towards direction γ, then transitions to γ' three times, before returning to the base and continuing infinitely *up* towards the termination stack symbol $\gamma_\top$.

From this description, it is clear that the correct local restriction number must be at least 3. However, let us explore the scenario where the restriction number k is set to just 1 or 2, following the perspective of Theorem 7.2.

According to the theorem, we must examine the value of $wt(\mathit{Up}^{\perp}_{[q]})$. For the purpose, we inspect the decomposition equation Equation (5.9) for it. With $1 \leq k < 3$, the only possible $\mathbb{D}$ that results in a non-empty $\mathit{SynPaths}^{\mathbb{D}}_{(\perp,[q])}$ is defined as follows:

$$\mathbb{D}(\gamma) = [q, q] \qquad \mathbb{D}(\gamma') = \varepsilon \qquad \mathbb{D}(\gamma_\top) = [q]$$

Thus, the only viable synchronisation path is:

$$[(q, q, m_2, \gamma), (q, q, m_2, \gamma_\top)]_{(m_1,\perp)}$$

Consequently, the decomposition equation for $wt(\mathit{Up}^{\perp}_{[q]})$ becomes:

$$wt(\mathit{Up}^{\perp}_{[q]}) = wt(\mathit{Trips}^{\gamma}_{[q,q]}) \cdot wt(\mathit{Up}^{\gamma_\top}_{[q]})$$

The pattern $\mathit{Up}^{\gamma_\top}_{[q]}$ progresses infinitely *up* to itself, hence we arrive instantly at that: $wt(\mathit{Up}^{\gamma_\top}_{[q]}) = 1$.

However, upon reaching the pattern $\mathit{Trips}^{\gamma}_{[q,q]}$, we find that there is no possible $\mathbb{D}$ with interactions adhering to the k-restriction that leads to a non-empty $\mathit{SynPaths}^{\mathbb{D}}_{(\gamma,[q,q])}$, resulting in that:

$$wt(\mathit{Trips}^{\gamma}_{[q,q]}) = 0$$

In summary, it means when we stick to an incorrect restriction number $1 \leq k < 3$, we will derive that $wt(\mathit{Trips}^{\perp}_{[q]}) = 0$.

Conversely, when we consider a correct restriction number $k \geq 3$, we observe:

$$wt(\mathit{Trips}^{\gamma}_{[q,q]}) = wt(\mathit{Trips}^{\gamma'}_{[q,q,q,q,q,q]})$$

with

$$wt(\mathit{Trips}^{\gamma'}_{[q,q,q,q,q,q]}) = 1$$

leading to the final result:

$$wt(\mathit{Trips}^{\perp}_{[q]}) = 1$$

The core issue then becomes calculating the value of $wt(\mathit{Up}^{\perp}_{[q_{init}]})$. The focus of this section revolves around the methodology to ascertain this value through the decomposition paradigm.

Before exploring the specifics, we provide some symbolic conventions for the equation systems that will be used throughout the subsequent discussion.

$$
\begin{array}{rcl|rcl}
(q, m_1, \bot) & \to & (q, m_2, up_{\gamma'}) & (q, m_1, \gamma) & \to & (q, m_2, up_{\gamma'}) \\
(q, m_Y, \bot) & \to & (q, m_Y, up_{\gamma_\top}) & (q, m_2, \gamma) & \to & (q, m_3, up_{\gamma'}) \\
(q, m_X, \gamma_\top) & \to & (q, m_X, up_{\gamma_\top}) & (q, m_3, \gamma) & \to & (q, m_4, up_{\gamma'}) \\
(q, m_X, \gamma') & \to & (q, m_X, down) & (q, m_4, \gamma) & \to & (q, m_4, down)
\end{array}
$$

$*$: Y ranges over m_2, m_3 and m_4.

X ranges over m_1, m_2, m_3 and m_4.

Figure 7.1: Rules for PTSA in Example 7.3

- For simplicity and clarity, we refer to the functions $\mathcal{F}$ that maps from a patterns-indexed vector to another such vector as directly the *equation system* as it contains all necessary information. Although this usage is slightly imprecise, it simplifies reference to the system $\vec{x} = \mathcal{F}(\vec{x})$. Note that by *patterns-indexed* vectors, we may refer to only a subset of patterns as indices, and the actual indexing set is usually indicated from the context.

- The function $\mathcal{F}_{wt(\mathit{TripsOrUp})}$ now corresponds to the equation system established in Equation (5.9), indicating the inclusion of the bottom ones (Equation (5.8)), so does $\mathcal{F}_{wt(\mathit{Up})}$.

- We fix a specific set of variables $\vec{x}$ and denote the variable corresponding to pattern v as $\langle v \rangle$, that is: $\vec{x}[v] = \langle v \rangle$.

- The notation $\vec{\Psi}_{wt(\mathit{TripsOrUp})}$ represents the pattern-indexed vector of weights associated with *TripsOrUp* patterns, with:

$$\vec{\Psi}_{wt(\mathit{TripsOrUp})}[\mathit{TripsOrUp}^q_D] := wt(\mathit{TripsOrUp}^q_D)$$

  We write $\vec{\Psi}_{wt(\mathit{Trips})}$ and $\vec{\Psi}_{wt(\mathit{Up})}$ to denote the vectors containing only the indicated dimensions.

Target Value as GFP

As have been discussed, the value of $\vec{\Psi}_{wt(\mathit{Trips})}$ corresponds to the Least Fixed Point (LFP) of $\mathcal{F}_{wt(\mathit{Trips})}$. Yet, for the value of $\vec{\Psi}_{wt(\mathit{Up})}$, there is no established result. Evidently, it cannot be the LFP again, as the LFP would trivially equal 0. This follows from the fact that the equation system for the *Up* patterns is *homogeneous*. Specifically, each summand on the RHS must also contain an *Up* pattern [1], which then implies that the LFP is trivially the zero solution. For clarity, let us introduce a new equation system, $\mathcal{G}$, where the variables $\langle \mathit{Trips}^\gamma_D \rangle$ are substituted with their respective $wt(\mathit{Trips}^\gamma_D)$ values, resulting in an equation system for the *Up* patterns.

[1] Recall that the definition of $\mathbb{D}$ requires a match with an *Up* pattern to have exactly one dimension of odd length.

We establish that the value of $\vec{\Psi}_{wt(Up)}$ is not the LFP, but rather precisely the Greatest Fixed Point (GFP) of $\mathcal{G}$. However, the GFP does not conform to the natural range of $[0, 1]$. Indeed, we have:

Theorem 7.4. *The value $\vec{\Psi}_{wt(Up)}$ is the greatest fixed point (GFP) of $\mathcal{G}$ within the following range. For any variable $\left\langle Up^{\mathscr{g}}_{D \Vdash[\mathscr{q}]} \right\rangle$ in $\mathcal{G}$, its range is $\left[0, wt(Trips^{\mathscr{g}}_{D})\right]$* [2].

Proof. Consider the target Greatest Fixed Point (GFP) of $\mathcal{G}$ falling within the specified ranges, denoted as ν. For any pattern $Up^{\mathscr{g}}_{D \Vdash[\mathscr{q}]}$, it follows from the definition that $0 \leq wt(Up^{\mathscr{g}}_{D \Vdash[\mathscr{q}]}) \leq wt(Trips^{\mathscr{g}}_{D})$. Consequently, $\vec{\Psi}_{wt(Up)}$, as a fixed point, falls within the aforementioned range, implying $\vec{\Psi}_{wt(Up)} \leq \nu$.

Next, we aim to establish that $\nu \leq \vec{\Psi}_{wt(Up)}$. We introduce an infinite series: $\vec{V}_0, \vec{V}_1, \vec{V}_2, \ldots$, where for a given h, the *Up*-patterns indexed vector $\vec{V}_h$ at dimension $Up^{\mathscr{g}}_{D}$ is a sub-*Up*-pattern $\langle Up^{\mathscr{g}}_{D} \rangle^{\uparrow h}$, which contains the set of all pre-runs that: (1) meet the condition D; (2) adhere to the k-restriction; and (3) end with a *visit-from-below* to a node at height of h levels above the subject node of the pattern.

Naturally, as h approaches infinity, it holds that $Up^{\mathscr{g}}_{D} = \lim_{h\to\infty} \langle Up^{\mathscr{g}}_{D} \rangle^{\uparrow h}$. Additionally, when $h = 0$, we assume it has already reached the target level 0 (the current/subject node), hence the set $\left\langle Up^{\gamma}_{D \Vdash[\mathscr{q}]} \right\rangle^{\uparrow 0}$ contains exactly the pre-runs of the form $\vec{\mu} \Vdash [\mathscr{c}]$, where $\vec{\mu} \in Trips^{\gamma}_{D}$ and $\mathscr{c} = (\mathscr{q}, \mathscr{t}, [\gamma])$, with $\mathscr{t}$ being the same as the last configuration of $\vec{\mu}$.

By the same derivation as for the sub-*Trips*-patterns in the proof for Theorem 4.43, namely, a pre-path decomposition derivation followed by a weight decomposition, we could easily obtain the following equation system:

$$wt(\vec{V}_{h+1}) = \mathcal{G}(\vec{V}_h)$$

We then prove by induction that for any finite h, $wt(\langle Up^{\mathscr{g}}_{D} \rangle^{\uparrow h}) \geq \nu[Up^{\mathscr{g}}_{D}]$ for every pattern $Up^{\mathscr{g}}_{D}$. In the base case, for every *Up* pattern, we have the following equation holds by definition:

$$wt(\left\langle Up^{\mathscr{g}}_{D \Vdash[\mathscr{q}]} \right\rangle^{\uparrow 0}) = wt(Trips^{\mathscr{g}}_{D}) \geq \nu[Up^{\mathscr{g}}_{D \Vdash[\mathscr{q}]}]$$

For the inductive step, we consider the decomposition for each pattern:

$$\begin{aligned}
& wt(\langle Up^{\mathscr{g}}_{D} \rangle^{\uparrow h+1}) \\
= & \sum_{\mathbb{D}} wt(SynPaths^{\mathbb{D}}_{(\mathscr{g},D)}) \cdot wt(\left\langle Up^{\gamma_{\mathbb{D}}}_{\mathbb{D}(\gamma_{\mathbb{D}})} \right\rangle^{\uparrow h}) \cdot \prod_{\gamma \in \Gamma \setminus \{\gamma_{\mathbb{D}}\}} wt(Trips^{\gamma}_{D}) \\
\geq & \sum_{\mathbb{D}} wt(SynPaths^{\mathbb{D}}_{(\mathscr{g},D)}) \cdot \nu[Up^{\gamma_{\mathbb{D}}}_{\mathbb{D}(\gamma_{\mathbb{D}})}] \cdot \prod_{\gamma \in \Gamma \setminus \{\gamma_{\mathbb{D}}\}} wt(Trips^{\gamma}_{D}) \\
= & \nu[Up^{\mathscr{g}}_{D}]
\end{aligned}$$

Here, $\mathbb{D} \sim$: $Up^{\mathscr{g}}_{D}$ with the sole direction of odd length denoted $\gamma_{\mathbb{D}}$. Note that the first equality stems from $\mathcal{G}$.

This leads to the conclusion that $\vec{\Psi}_{wt(Up)} = \lim_{h\to\infty} wt(\vec{V}_h) \geq \nu[Up]$. Considering the initial inequality, the theorem is thus proven. □

[2] Recall that $wt(Trips^{\gamma}_{\varepsilon}) = 1$. For simplicity, we temporarily define $wt(Trips^{\perp}_{\varepsilon}) = 1$.

7.1.2 Local Checking Algorithm

Given the aforementioned results, we now introduce an algorithm for validating k. This method utilises solely the *Existential fragment* of the first-order Theory of the Reals (EToR) (Theorem 2.8).

We first construct the complete equation system $\mathcal{F}$ as per Equation (5.9) for all patterns. As for qualifying the LFP of the involved *Trips* patterns, we adopt the method introduced in Section 5.1.3 to $\mathcal{F}$. Upon incorporating the aforementioned qualifications, the final query formula consists of the conjunction of the following terms:

- For each $\mathit{Trips}^{\gamma}_{D}$ pattern on the LHS of $\mathcal{F}$:

$$0 \leq \langle \mathit{Trips}^{\gamma}_{D} \rangle \leq 1$$

- For every $\mathit{Up}^{\mathcal{Q}}_{D +\!\!+ [\mathcal{Q}]}$ pattern on the LHS of $\mathcal{F}$:

$$0 \leq \langle \mathit{Up}^{\mathcal{Q}}_{D +\!\!+ [\mathcal{Q}]} \rangle \leq \langle \mathit{Trips}^{\mathcal{Q}}_{D} \rangle$$

  when $D \neq \varepsilon$, or alternatively:

$$0 \leq \langle \mathit{Up}^{\mathcal{Q}}_{D +\!\!+ [\mathcal{Q}]} \rangle \leq 1$$

  if $D = \varepsilon$.

- The equation system:

$$\vec{x} = \mathcal{F}(\vec{x})$$

- For each anchor *Trips* pattern $\mathit{Trips}^{\gamma}_{D}$ and its corresponding LFP upper bound, denoted as c below, as introduced in Section 5.1.3:

$$\langle \mathit{Trips}^{\gamma}_{D} \rangle \leq c$$

- The target formula:

$$\langle \mathit{Up}^{\perp}_{[\mathcal{Q}_{init}]} \rangle \geq 1$$

Thus, if the EToR procedure yields `True`, it indicates that the GFP value of the pattern $\mathit{Up}^{\perp}_{[\mathcal{Q}_{init}]}$ is at least 1, signifying that k is (reachably) correct. Conversely, a result of `False` implies the incorrectness of k.

7.2 Global Validation

Building upon the results of local validation, we proceed to delineate the global validation methodology. The principal concept involves transforming the global validation challenge into a local validation scenario.

The crux of this approach lies in the observation that a run of a PTSA can be decomposed into two segments: the segment dedicated to “constructing” the

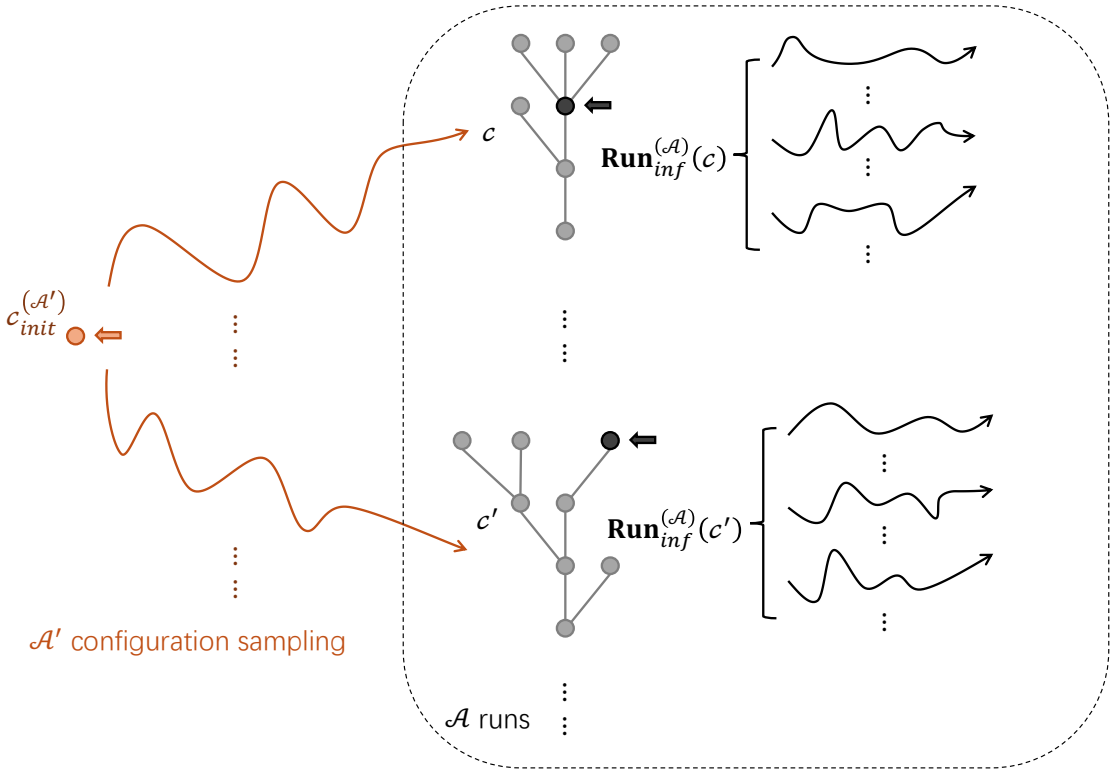

Figure 7.2: From Local to Global Validation

starting configuration, and the segment that executes from the constructed starting configuration according to the rules in $\mathcal{A}$. This necessitates the construction of an auxiliary simulation rPTSA $\mathcal{A}'$ derived from $\mathcal{A}$, wherein configurations are firstly constructed through "random sampling". In this manner, each run of $\mathcal{A}$ is "weighted" in $\mathcal{A}'$ by the process of constructing the starting configuration. With the construction, the critical observation would then be that $k+1$ is valid for $\mathcal{A}'$ if and only if k is valid for $\mathcal{A}$. We now proceed to formalise the aforementioned process.

This idea is depicted in Figure 7.2. The dotted frame in the figure encompasses all possible runs of $\mathcal{A}$, originating from all configurations of $\mathcal{A}$ such as c and c' and extending infinitely. These runs are represented by the curves within the figure. To verify the global correctness for a given k, we introduce an automaton $\mathcal{A}'$ that begins with a random "sampling" process to construct configurations of $\mathcal{A}$, as shown by the orange curves indicating the sampling runs. Should this procedure ensure that each configuration, such as c and c', within $\mathcal{A}$ can be reached from the initial configuration of $\mathcal{A}'$ with a non-zero probability, the global validation problem for $\mathcal{A}$ is thus converted into a local validation problem for $\mathcal{A}'$.

7.2.1 Constructing Auxiliary Automaton

To formalise this process, we commence by defining $\mathcal{A}' = (\mathcal{Q} \uplus \{q_{cons}\}, \mathcal{M}' \uplus \mathcal{M}, \Gamma \uplus \{\nabla\}, \Delta' \uplus \Delta, q_{cons}, m_{init})$ [3], where:

- q_{cons} is a distinct new control state representing the "sampling" process of constructing the configurations of $\mathcal{A}$.
- $\mathcal{M}' := \mathcal{M} \times (\Gamma \to \{0, 1\})$ is a new set of local memory which includes information regarding which tree form "has been constructed". Notably, this information is essential to avoid revisiting the constructed branches so to maintain the whole construction procedure to be 1-restricted.
- ∇ is an auxiliary, new stack symbol used to manage the control state transition [4].

Finally, Δ' constitutes a new set of rules that account for the construction process, incorporating the new construction control state q_{cons} as well as the new local memories in $\mathcal{M}'$.

More specifically, Δ' is defined by the following rules. It is important to note that we present only the essential rules that may be reachable from the initial configuration of $\mathcal{A}'$, as these are the only parts relevant to the subsequent discussion. For those status that is not addressed, such as (q_{cons}, m, g) for $m \in \mathcal{M}$ where $m \neq m_{init}$, their distribution of operations can be arbitrary, as they are not engaged in the subsequent analysis.

1. Δ' contains the following rules for q_{cons}. Intuitively, with q_{cons} being the current state, the automaton randomly constructs the tree stack with regards to the visiting information stored locally so to maintain the 1-restriction during the construction of the tree stack. To formalise the random construction, notice also a special case to handle that as when the node is created the initial local memory m_{init} is not in $\mathcal{M}'$. Yet, we may regard it as the same as with the visiting information that no node at above has been visited.

   Formally, the following two sets of rules for m_{init} and local memories in $\mathcal{M}'$ respectively are in Δ' for q_{cons}.

   - For m_{init}, let $f_\emptyset := \{\gamma \mapsto 1 \mid \gamma \in \Gamma\}$, indicating that no node has yet been visited:
     - For every $m \in \mathcal{M}$, the random *up* transitions to a direction $\gamma \in \Gamma$, ensuring to record the visit to this direction:

$$(q_{cons}, m_{init}, g) \xrightarrow{p^g_{init}} (q_{cons}, (m, f_\emptyset[\gamma \mapsto 0]), up_\gamma)$$

[3] Note, in particular, that Δ and Δ' have disjoint domains, with Δ' accounting for the newly added control states and local memories.

[4] This is essentially an adoption of the horizontal elimination procedure, as discussed in Section 9.3.2.

– For $g \neq \bot$ and each $m \in \mathcal{M}$, the random *down* transitions proceed as follows. Given the 1 restriction to maintain during the constructing phase, it is safe to omit recording any visit information at this node. Furthermore, as it is assumed that no terminating rule exists, when $g = \bot$, no *down* transition occurs.

$$(q_{cons}, m_{init}, g) \xrightarrow{p^g_{init}} (q_{cons}, m, down)$$

– Additionally, the rule that indicates the end of the random construction of the tree stack:

$$(q_{cons}, m_{init}, g) \xrightarrow{p^g_{init}} (q_{cons}, m_{init}, up_\nabla)$$

Here, p^g_{init} is given by:

$$\frac{1}{1 + (\mathbf{1}[g \neq \bot] + |\Gamma|) \cdot |\mathcal{M}|}$$

where $\mathbf{1}[P]$ is the usual indicator function that takes the value 1 if the proposition P holds, and 0 otherwise.

- For any function $f : \Gamma \to \{0, 1\}$ and any $m \in \mathcal{M}$:
  – For each γ with $f(\gamma) = 1$, the random *up* transitions, ensuring to update the visit information:

$$(q_{cons}, (m, f), g) \xrightarrow{p^g_f} (q_{cons}, (m, f[\gamma \mapsto 0]), up_\gamma)$$

  – For $g \neq \bot$, the random *down* transitions, given the 1 restriction, it is safe to clear the visit information from the local memory:

$$(q_{cons}, (m, f), g) \xrightarrow{p^g_f} (q_{cons}, m, down)$$

  – Additionally, the rule signalling the end of the random construction of the tree stack:

$$(q_{cons}, (m, f), g) \xrightarrow{p^g_f} (q_{cons}, m, up_\nabla)$$

  Here, the probability p^g_f is given by:

$$\frac{1}{1 + \mathbf{1}[g \neq \bot] + \sum_{\gamma \in \Gamma} f(\gamma)}$$

2. Additionally, Δ' contains the following rules for $\mathcal{M}'$ for non-q_{cons} control states that simulate the original rules: For every rule in Δ:

$$(q, m, g) \xrightarrow{p} op$$

the corresponding rules in Δ' are:

$$(q, (m, f), g) \xrightarrow{p} op$$

for all $f : \Gamma \to \{0, 1\}$.

3. Additionally, the auxiliary rules transition the control state from ∇. Specifically, for each control state $q \in Q$:

$$(q_{cons}, m_{init}, \nabla) \xrightarrow{\frac{1}{|Q|}} (q, m_{init}, down)$$

Given a configuration $c = (q, t, \rho)$ of $\mathcal{A}$, a configuration $c' = (q, t', \rho)$ of $\mathcal{A}'$ is said to *simulate* c, iff: the current control state q and stack pointer ρ are the same as in c, and for the tree stack t':

- There is a sequence $\rho_1 \in \mathrm{dom}(t)$ with $\mathrm{dom}(t') = \mathrm{dom}(t) \uplus \{\rho_1 \nabla \mapsto m_{init}\}$ [5].
- There exists a prefix ρ' of ρ such that:
  - For all $\rho'' \sqsubseteq \rho'$ [6], $t'(\rho'') = (t(\rho''), f)$ with $f(\gamma) = \mathbf{1}[\rho''\gamma \notin \mathrm{dom}(t)]$ for all $\gamma \in \Gamma$; and,
  - For all other sequences $\rho'' \not\sqsubseteq \rho'$, $t'(\rho'') = t(\rho'')$.

We now state the following properties to establish the correctness of the construction of $\mathcal{A}'$.

Lemma 7.5. *Any initial run of $\mathcal{A}'$ that concludes with the current control state being q_{cons} conforms to the 1-restriction.*

Proof. This follows directly from the definition of the rules. □

Lemma 7.6. *For any configuration c of $\mathcal{A}$, there exists an initial run $\ldots c''c'$ of $\mathcal{A}'$ such that $c' = (-, t, \rho)$ simulates c and $c'' = (q_{cons}, t, \rho\nabla)$.*

Proof. This is straightforward from the definition. □

The above property essentially establishes the soundness of the construction of $\mathcal{A}'$ by demonstrating its capability to simulate every configuration. Next, we state the completeness property, indicating that every such run concludes with a proper simulation of some configuration.

Lemma 7.7. *For any initial run $\mu = \ldots cc'$ of $\mathcal{A}'$ where $c = (q_{cons}, t, \rho\nabla)$ and $c' = (q, t, \rho)$ with $q \neq q_{cons}$, c' corresponds to a configuration of $\mathcal{A}$.*

Proof. To prove this, we can show by straightforward induction on the length of the run that the tree t appearing in every initial run of $\mathcal{A}'$ that ends with a configuration of the form (q_{cons}, t, ρ'), where ρ' does not include ∇, meets the criteria for simulation. The only missing aspect is a unique path ending with ∇ in its domain. Thus, by proceeding to ∇ immediately after this point (which is always feasible by the rules definition), the complete conditions for simulation are fulfilled. □

Finally, we declare the propagation of the simulation property:

Lemma 7.8. *For any run $\mu = c_1 c_2 \ldots c_n$ of $\mathcal{A}'$, if c_1 is a simulation of some configuration of $\mathcal{A}$, so is c_n.*

Proof. This follows by straightforward induction on the length of the run μ. □

[5] Note that ∇ is NOT in Γ, hence $\mathrm{dom}(t)$ will never include sequences containing ∇.

[6] Recall that $\sqsubseteq$ denotes the prefix relation.

7.2.2 From Local to Global

Based on the established properties, we are now poised to state the property of run simulation. Building upon the definition of simulation, we naturally define the function of *desimulation* which simply removes the additional visiting information in the local memories of the tree stack. Formally, given a configuration c' of $\mathcal{A}'$ that properly simulates some configuration c of $\mathcal{A}$, we define $\mathsf{DeSimul}(c')$ as follows: let $c' = (q, t', \rho)$, then $\mathsf{DeSimul}(c') = (q, t, \rho)$, where t is given by:

$$\{\rho' \mapsto RmF(t'(\rho')) \mid \rho' \in \mathrm{dom}(t'), \nabla \notin \rho'\}$$

where the function RmF intuitively removes the visiting information, defined as $RmF(m) := m$ when $m \in \mathcal{M}$, and otherwise, when $(m, -) \in \mathcal{M}'$, $RmF((m, -)) := m$. Clearly, it holds that $\mathsf{DeSimul}(c') = c$.

We define a run μ of $\mathcal{A}'$ as a *simulating run* if it is an initial run and concludes with a configuration whose current control state is in $\mathcal{Q}$ (i.e., not equal to q_{cons}). We then naturally introduce the function $\mathsf{DeSimulRun}(-)$ as an extension of $\mathsf{DeSimul}(-)$ to simulating runs. Given a simulating run μ, we first divide μ into two parts at the last occurrence of q_{cons}. That is, let $\mu = \mu_1 +\!\!+ \mu_2$, where μ_1 ends with a configuration with the current state q_{cons} (implying also it being the unique configuration in μ with stack pointer ending with ∇), while μ_2 starts with a configuration whose current state is not q_{cons}. Then, $\mathsf{DeSimulRun}(\mu)$ results in a sequence of configurations [7] of $\mathcal{A}$ given by a pointwise application of $\mathsf{DeSimul}(-)$ to each configuration of μ_2. Note that this pointwise application is always possible – i.e., for every simulating run μ, every configuration of μ_2 must be a proper simulation of some configuration by Lemma 7.7 and Lemma 7.8.

Thus, we have the following key property of the relationship between simulating runs and runs in $\mathcal{A}$.

Lemma 7.9. *The function* $\mathsf{DeSimulRun}(-)$ *is surjective onto all finite runs of* $\mathcal{A}$. *More specifically, for every simulating run* μ' *of* $\mathcal{A}'$, $\mathsf{DeSimulRun}(\mu')$ *is always a valid run. Additionally, for every finite run* μ *of* $\mathcal{A}$, *there exists some simulating run* μ' *of* $\mathcal{A}'$ *such that* $\mathsf{DeSimulRun}(\mu') = \mu$.

Proof. For the former condition that $\mathsf{DeSimulRun}(\mu')$ invariably results in a proper run of $\mathcal{A}$, this can be demonstrated through a simple induction on the length of the postfix of μ' with the current states being non-q_{cons}. Note that the base case is addressed by Lemma 7.7.

Regarding the latter case, it can be established by a straightforward induction on the length of μ, with the base case handled by Lemma 7.6.

In both scenarios, at the inductive step, the key will all be the existence of corresponding transitions. This essentially involves the Item 2 in the definition of Δ' above – that a rule $(q, (m, -), g) \xrightarrow{p} op$ exists in $\mathcal{A}'$ iff the rule $(q, m, g) \xrightarrow{p} op$ exists in $\mathcal{A}$. □

[7] It is not valid to refer to this as a run yet because it is unknown here whether there is a transition between every configuration. But indeed there is – see the forthcoming Lemma 7.9.

Finally, we present the advantageous property that delineates the computation of global restriction through local restriction.

Theorem 7.10. *$k+1$ constitutes a correct local restriction of the newly constructed $\mathcal{A}'$ if and only if k represents a correct global restriction of $\mathcal{A}$.*

To establish this theorem, we provide further observations regarding runs and their initial configurations.

Lemma 7.11. *For any configurations c_1 and c_2 of $\mathcal{A}$, assuming they share the same current state and stack pointer, i.e., let $c_1 = (q, t, \rho)$, and $c_2 = (q, t', \rho)$, then, when $dom(t) \subseteq dom(t')$ and for any $\rho' \in dom(t') \setminus dom(t)$, $t(\rho') = m_{init}$, the two configurations admit the same set of enabled finite paths.*

Proof. By straightforward induction on the length of the paths. □

We are now prepared to prove Theorem 7.10 with all the results established.

Proof (Theorem 7.10). The reverse direction is straightforward by Lemma 7.9.

For the forward direction, we use a proof by contradiction. Assume $k+1$ is a correct local restriction of $\mathcal{A}'$, yet k is not a correct global restriction of $\mathcal{A}$. Then, there must be a finite run $\mu = c_1 \dots c_n$ of $\mathcal{A}$ that ends with the $(k+1)$th visit to a node (at the stack pointer of c_n). Let $c_1 = (-, t_1, -)$ and $c_n = (-, -, \rho_n)$. We consider two cases based on whether $\rho_n \in \mathrm{dom}(t_1)$.

If $\rho_n \in \mathrm{dom}(t_1)$, then by Lemma 7.6, there must be a initial run that simulates c_1, so that any simulating run μ' with $\mathsf{DeSimulRun}(\mu') = \mu$ will visit ρ_n for $k+2$ times, hence violating the assumption that $k+1$ is a correct local restriction of $\mathcal{A}'$.

If $\rho_n \notin \mathrm{dom}(t_1)$, it is still possible to construct a initial run that follows the same path as μ starting from a configuration $c_1' = (-, t_1', -)$ such that $\rho_n \in \mathrm{dom}(t_1')$ – we need to ensure that all $\rho \sqsubseteq \rho_n$ are in $\mathrm{dom}(t_1')$, and for all $\rho \notin \mathrm{dom}(t_1)$, let $t_1'(\rho) = m_{init}$. The validity of such a construction is given by Lemma 7.11. Thus, the run μ' for c_1' with the same path as μ will visit ρ_n for $k+1$ times. Then, by the same reasoning as the previous case, there must be a simulating run of $\mathcal{A}'$ that visits ρ_n for $k+2$ times.

In both cases, this violates the assumption that $k+1$ is a correct local restriction of $\mathcal{A}'$. Hence the conclusion is shown. □

Algorithm for Checking Global Correctness To check the global correctness of k for $\mathcal{A}$, we need to construct the corresponding $\mathcal{A}'$ and then verify if $k+1$ is a locally correct restriction number for $\mathcal{A}'$.

Remark 7.12 (Undecidability of Restriction)**.** While deciding if a PTSA $\mathcal{A}$ conforms to a specific k-restriction is possible, determining whether any PTSA $\mathcal{A}$ is restricted for some k is undecidable. This can be proven by contradiction using the halting problem of a deterministic Turing machine.

Assume an algorithm exists to decide if $\mathcal{A}$ is restricted for all PTSA $\mathcal{A}$. For any deterministic Turing machine $\mathcal{T}$, which can be seen as a PTSA with each rule having probability 1, this algorithm could determine if $\mathcal{T}$ is restricted.

If $\mathcal{T}$ is restricted, we could enumerate to find a k making $\mathcal{T}$ k-restricted by the local correctness checking algorithm developed above, implying a decidable halting problem. If $\mathcal{T}$ is not restricted, there is a run, which, by determinism, is the unique run, that visits a node infinitely, implying $\mathcal{T}$ does not halt. Thus, the halting problem would be decidable, which is impossible. Therefore, no algorithm can decide if all PTSA are restricted. This result aligns perfectly with the renowned Rice's theorem [195].

Remark 7.13 (Validating Non-Probabilistic Automaton)**.** The property of restriction fundamentally concerns the existence of runs that contravene the restriction, rather than the probabilities themselves. Consequently, validation algorithms are directly applicable to non-probabilistic (non-deterministic) tree-stack automata (TSA). Given that a Turing machine can be essentially viewed as a TSA, the validation algorithm can be similarly applied to Turing machines.

To elucidate, in validating the restriction of a TSA $\mathcal{A}$, we can assign a random non-zero probability to each transition rule, thereby forming an auxiliary PTSA $\mathcal{A}'$. Subsequently, the aforementioned algorithm can be employed. It can be observed that, given a restriction nubmer k, the original TSA $\mathcal{A}$ violates the k-restriction if and only if the constructed auxiliary PTSA $\mathcal{A}'$ violates the k-restriction. This equivalence follows immediately from the one-to-one correspondence between the runs of $\mathcal{A}$ and $\mathcal{A}'$.

The true is the whole. But the whole is nothing other than the essence consummating itself through its development. It is of an absolute necessity that this self-fulfillment of essence should be conceived not as immediate, but as a process, as the becoming of essence, and hence as the eternal simple self-identity of essence in its absolute opposition.

— Hegel, G.W.F. Phenomenology of Spirit, 1807

8 Interrelationships

Contents

In this chapter, we delve into the application of the rPTSA formalism across various contexts, comparing it with several established formalisms. Our focus is on understanding the relative expressive power and distinct features of rPTSA in comparison to these approaches, with the aim of elucidating its theoretical characteristics and potential applications.

Firstly, we examine the relationship between rPTSA and pPDA, emphasising the extra-expressiveness of rPTSA.

Next, we will explore in detail the interconnections between rPTSA and the *probabilistic additive higher-order recursion schemes (PAHORS)*, demonstrating the equivalence of these formalisms. This discussion relates our findings to the context of higher-order program verification, following the work by Clairambault et al. [102]. Furthermore, identifying this relationship situates the rPTSA model within the strict hierarchy of higher-order probabilistic grammars.

Additionally, we will investigate the non-probabilistic variant of rPTSA, known

as rTSA, with particular focus on its association with the field of linguistics – the formalism of *multiple context-free grammar* (MCFG). The equivalence between these two formalisms was first demonstrated by Denkinger [99]. In this study, beyond reiterating the equivalence, we introduce a novel methodology to convert rTSA back to MCFG, using a structure inspired by the equation system approach discussed in earlier analyses. Finally, we will provide a concise discussion on the incorporation of probability (stochasticity) into these grammars and try to address their termination problems.

8.1 pPDA

As the probabilistic extension of the well-known pushdown automaton, the probabilistic pushdown automaton (pPDA), along with its equivalent model, the recursive Markov chain (RMC), has been extensively researched. To better position our model, which claims additional expressiveness, it is useful to discuss the relationship between rPTSA and pPDA.

8.1.1 Definitions

Let us first present the definition of pPDA. In the literature [87, 119], the model of pPDA is typically presented as a set of probabilistic rewriting rules of the form:

$$q\ X \xrightarrow{p} q'\ X_1\ \ldots\ X_n \tag{8.1}$$

Here, q ranges over a finite set of control states, with X and X_i ranging over a finite set of stack symbols. The rule intuitively describes matching the control state and the stack symbol, and replacing the two by the RHS of the rule.

However, to better discuss the relationship, we present the definition of pPDA in a way that is closer to rPTSA. They are essentially rPTSA without (or, equivalently, with a fixed single) local memories. More specifically:

Definition 8.2. A *Probabilistic Pushdown Automaton* (pPDA) is presented as a tuple $(Q, \Gamma, \Delta, q_{init})$, where: Q is a finite set of control states; Γ is a finite set of stack symbols; $q_{init} \in Q$ is a control state called the *initial control state*; $\Delta : Q \times (\Gamma \uplus \{\bot\}) \to \mathsf{Dist}(Q \times (\{push_\gamma \mid \gamma \in \Gamma\} \uplus \{pop\}))$. We write $(q, \gamma) \xrightarrow{p} (q', op)$ to denote $\Delta(q, \gamma)(q', op) = p$ as in rPTSA, where op ranges over either pop or $push_\gamma$ for some $\gamma \in \Gamma$.

Note that here, we use γ, following the symbolic convention of rPTSA, rather than X, Y like those in eq. (8.1) to denote the stack symbols. Again, we use γ to range over both elements in Γ and $\bot$.

This definition is clearly equivalent to the previous one, where a rule like eq. (8.1) can be converted to a pop followed by a consequential push rule. More specifically, it is converted to the following rule:

$$(q, X) \xrightarrow{p} (\ulcorner q' X_1 \ldots X_n \urcorner, pop)$$

Along with the supportive rules for each newly introduced intermediate state $\ulcorner q' X_1 \dots X_i \urcorner$ for $i \in \{2, \dots, X_n\}$ (when $n = 0$, no additional supportive rules): for each $Y \in \Gamma \uplus \{\bot\}$, there is a rule:

$$(\ulcorner q' X_1 \dots X_i \urcorner, Y) \to (\ulcorner q' X_1 \dots X_{i-1} \urcorner, push_{X_i})$$

Additionally, there is a rule:

$$(\ulcorner q' X_1 \urcorner, Y) \to (q', push_{X_1})$$

Running Behaviour of pPDA A configuration of pPDA is simply a sequence $q\ \gamma_1\ \dots\ \gamma_n$ where the stack symbol sequence captures the stack with γ_1 being the stack top. The transitions match one of the following patterns:

$$q\ \gamma_1\ \dots\ \gamma_n \xrightarrow{(q,\gamma_1) \xrightarrow{p} (q', pop)} q'\ \gamma_2\ \dots\ \gamma_n$$

$$q\ \gamma_1\ \dots\ \gamma_n \xrightarrow{(q,\gamma_1) \xrightarrow{p} (q', push_\gamma)} q'\ \gamma\ \gamma_1\ \dots\ \gamma_n$$

$$q\ \varepsilon \xrightarrow{(q,\bot) \xrightarrow{p} (q', push_\gamma)} q'\ \gamma$$

$$q\ \varepsilon \xrightarrow{(q,\bot) \xrightarrow{p} (-, pop)} \top$$

Hence, similar to the rPTSA case, a run is a sequence of configurations that might end with the special symbol $\top$, where there is a transition between consecutive elements.

8.1.2 pPDA $\Rightarrow$ 1-PTSA

With our definition established, we now discuss the conversion to rPTSA, or, more specifically, 1-PTSA. The challenge arises from the 1-restriction, which precludes the most obvious conversion by simply adding a single local memory. To work around this restriction, we can utilise the branching property to create "proxy" nodes that act as the original tree node.

More specifically, given a pPDA $A = (Q, \Gamma, \Delta, q_{init})$, we construct an equivalent 1-PTSA as $\mathcal{A} = (Q, \mathcal{M}, \Gamma^{(\mathcal{A})}, \Delta^{(\mathcal{A})}, q_{init}, m_{Init})$, where: Q is exactly the same as the original pPDA; $\mathcal{M}$ consists of three kinds of local memories m_{Init}, m_{Relay}, and m_{Del}; $\Gamma^{(\mathcal{A})} = \Gamma \uplus \{\mathsf{Proxy}(\gamma) \mid \gamma \in \Gamma\} \uplus \{\mathsf{Proxy}(\bot)\}$ contains three disjoint parts—the original stack symbols set Γ, a mirror set called the "proxy stack symbols", and the proxy bottom symbol $\mathsf{Proxy}(\bot)$; the initial control state q_{init} remains exactly the same as the original pPDA; finally, the transitions function $\Delta^{(\mathcal{A})}$ contains rules of the following three kinds:

- The bottom rules:

  $$(q_{init}, m_{Init}, \bot) \to (q_{init}, m_{Del}, up_{\gamma_\bot})$$

  and for all $q \in Q$:

  $$(q, m_{Del}, \bot) \to (q, m_{Del}, down)$$

- The rules directly converted from corresponding rules of the original pPDA—for each rule $(q, \mathscr{g}) \xrightarrow{p} (q', op)$, let op' be up_γ if $op = push_\gamma$ or $down$ if $op = pop$, then we add the following two rules simultaneously:

$$(q, \mathscr{m}_{Init}, \mathscr{g}) \xrightarrow{p} (q', \mathscr{m}_{Relay}, op')$$
$$(q, \mathscr{m}_{Init}, \mathsf{Proxy}(\mathscr{g})) \xrightarrow{p} (q', \mathscr{m}_{Relay}, op')$$

  That is, we define the rules corresponding to the same rule for $\mathscr{g}'$ and its proxy node $\mathsf{Proxy}(\mathscr{g}')$ simultaneously.

- The supportive relaying rules for each $q \in Q$ and $\gamma \in \Gamma'$:

$$(q, \mathscr{m}_{Relay}, \gamma) \to (q, \mathscr{m}_{Del}, up_{\mathsf{Proxy}(\gamma)})$$
$$(q, \mathscr{m}_{Relay}, \mathsf{Proxy}(\gamma)) \to (q, \mathscr{m}_{Del}, up_{\mathsf{Proxy}(\gamma)})$$
$$(q, \mathscr{m}_{Del}, \gamma) \to (q, \mathscr{m}_{Del}, down)$$
$$(q, \mathscr{m}_{Del}, \mathsf{Proxy}(\gamma)) \to (q, \mathscr{m}_{Del}, down)$$

The technical intuition is that when a node is visited from above – i.e., when its upper node is popped, a proxy node for the current node is created. This ensures that any subsequent pushes to the current node adhere to the 1-restriction. Upon the creation of the proxy node, the current node is marked as to-be-deleted. When the node is revisited from above, it implies that the proxy node has been popped, thus the current node should also be popped.

This intuition is represented through three types of local memories. The memory $\mathscr{m}_{Init}$ essentially indicates the state of the tree-stack being ready to execute the pPDA rule. The memory $\mathscr{m}_{Relay}$ signifies that the rule has been executed at this node, and any subsequent revisit should "relay" to its proxy node for further execution. The memory $\mathscr{m}_{Del}$ intuitively suggests that the execution at this node has been relayed to its proxy node; thus, when revisited, it indicates that its proxy node has been popped, necessitating the removal of the current node as well.

Validity of Conversion To verify the validity of the conversion, we first propose the concept of *consistency* (with respect to the conversion). Intuitively, we want the tree-stack to simulate the information of the linear stack at the stack pointer by all those elements with local memory at the tree-stack as either $\mathscr{m}_{Init}$ or $\mathscr{m}_{Relay}$.

To verify that this simulation holds, we first need to restrict our discussion to tree-stacks with this form, which we call *consistent*. Given a sequence of stack symbols ρ, if $\mathscr{g}(\rho)$ is of the form $\mathsf{Proxy}(\mathscr{g}')$, then, let $getG(\rho) = \mathscr{g}'$, otherwise, let $getG(\rho)$ be simply $\mathscr{g}(\rho)$.

We say a configuration $\mathscr{c} = (\mathscr{q}, \mathscr{t}, \rho)$ of a rPTSA converted from a pPDA is *consistent* if: (1) $\mathscr{t}(\rho) = \mathscr{m}_{Init}$; (2) for every $\rho' \sqsubset \rho$, if $\mathscr{t}(\rho') = \mathscr{m}_{Relay}$ then all sequences in the form $\rho'\mathsf{Proxy}(-)$ are not within $\mathrm{dom}(\mathscr{t})$; otherwise, if $\mathscr{t}(\rho') = \mathscr{m}_{Del}$ then $\rho'\mathsf{Proxy}(getG(\rho')) \sqsubseteq \rho$, and $\mathscr{t}(\rho')$ cannot be $\mathscr{m}_{Init}$.

When the configuration is consistent, then it essentially encodes, or *simulates*, a specific configuration of pPDA. To be more specific, the state information is

trivially and undoubtedly q, the current control state of the rPTSA configuration. The tree-stack t and ρ then encode the entire stack information – the stack symbols in ρ with m_{Relay} and m_{Init} as corresponding local memory on t are exactly the pPDA stack information.

We now define $\mathsf{ToStk}(c)$ to extract the stack from the tree-stack:

1. Construct a new sequence s by attaching the local memory information on the stack pointer ρ: $s := [(\gamma_1, m_1), \ldots, (\gamma_n, m_n)]$ where $\gamma_1 \ldots \gamma_n = \rho$, and each m_i is $t(\gamma_1 \ldots \gamma_i)$.

2. Filter the sequence s to keep only those pairs (γ_i, m_i) such that $\gamma_i \neq \mathsf{Proxy}(\bot)$ and $m_i \neq m_{Del}$, giving rise to s'.

3. Map s' so that only the first element (stack symbol) remains, then reverse the sequence (in pPDA, the stack top is on the left, in comparison to the rPTSA stack pointer), which gives us the result.

We formally define that a configuration $c = (q, t, \rho)$ *simulates* an original configuration $q\ \gamma_1\ \ldots\ \gamma_n$ of the source pPDA, if: (1) c is consistent; (2) $q = q$; and, (3) $\mathsf{ToStk}(c) = \gamma_1 \ldots \gamma_n$.

We claim the validity of the conversion with the following theorem, which asserts stepwise transition simulation.

Theorem 8.3. *For every configuration $\alpha = q\ \gamma_1\ \ldots\ \gamma_n$ of a pPDA A, and every configuration c of its converted rPTSA that simulates it, which must exist, for every transition $\alpha \xrightarrow{p} \alpha'$, there is a configuration c' that simulates α' such that c' is reachable by a probability p transition from c and some finite probability* 1 *transitions.*

Proof. This follows straightforwardly by rule definitions. □

Additionally, notice that the initial configuration $(q_{init}, \{\varepsilon \mapsto m_{Init}\}, \varepsilon)$ of the rPTSA simulates the initial configuration $q_{init}\ \varepsilon$ of the original pPDA.

Remark 8.4 (1-PTSA ⇒ pPDA)**.** The conversion from 1-PTSA back to pPDA is relatively straightforward. In essence, since each node is visited from below at most once, one can simply *pop* the node upon descending from it.

Specifically, adopting the rewriting rule style (i.e., Equation (8.1)), we could let the stack symbols in the pPDA be of the form $\ulcorner m, g \urcorner$, with m and g ranging over the local memories and stack statuses of the source rPTSA, the conversion can be represented as follows:

- A rule $(q, m, g) \xrightarrow{p} (q', m', \mathit{down})$ translates directly to:

$$q\ \ulcorner m, g \urcorner \xrightarrow{p} q'$$

- A rule $(q, m, g) \xrightarrow{p} (q', m', up_\gamma)$ simply translates to:

$$q\ \ulcorner m, g \urcorner \xrightarrow{p} q'\ \ulcorner m_{init}, \gamma \urcorner\ \ulcorner m', g \urcorner$$

$$\frac{\mathcal{N}(F) = \theta}{\mathcal{N} \mid \mathcal{E} \vdash F : \theta} \qquad \frac{\mathcal{E}(x) = \theta}{\mathcal{N} \mid \mathcal{E} \vdash x : \theta} \qquad \frac{}{\mathcal{N} \mid \mathcal{E} \vdash \top : o}$$

$$\frac{\mathcal{N} \mid \mathcal{E}_1 \vdash t_1 : \theta \multimap \theta' \qquad \mathcal{N} \mid \mathcal{E}_2 \vdash t_2 : \theta}{\mathcal{N} \mid \mathcal{E}_1, \mathcal{E}_2 \vdash t_1\, t_2 : \theta'} \qquad \frac{\mathcal{N} \mid \mathcal{E} \vdash t_i : o \quad i = 1, \cdots, n}{\mathcal{N} \mid \mathcal{E} \vdash \langle t_1, \cdots, t_n \rangle : o^n}$$

$$\frac{\mathcal{N} \mid \mathcal{E} \vdash t : o^n \quad i \in \{1, \cdots, n\}}{\mathcal{N} \mid \mathcal{E} \vdash \pi_i\, t : o} \qquad \frac{\mathcal{N} \mid \mathcal{E}, x : \theta \vdash t : \theta'}{\mathcal{N} \mid \mathcal{E} \vdash \lambda x.t : \theta \multimap \theta'}$$

Figure 8.1: Affine additive typing rules

8.2 PAHORS

Higher-order recursion schemes (HORS) are a computational formalism based on rewriting typed terms. Each reduction step corresponds to unfolding a non-terminal according to a corresponding rule, which involves substituting actual arguments (which may include functions) into the function body. In standard HORS, the type discipline follows that of the simply-typed λ-calculus. Recent work [196] introduced a probabilistic extension of HORS and demonstrated that undecidability (of almost sure termination) occurs at order 2, i.e., when arguments can be of base type or first-order functions.

In this work, we impose no restrictions on the type-theoretic order. However, we adopt a stricter type discipline to constrain the form of admissible terms. This discipline is derived from linear logic [103], specifically the affine constraint, which intuitively restricts the usage of every variable to *at most once*, resulting in a model called the *probabilistic additive HORS* (PAHORS).

In this section, we first provide a formal definition of PAHORS. Next, we present a conversion from rPTSA to PAHORS. Finally, we describe the conversion from PAHORS to rPTSA, demonstrating that the termination and model-checking problems of PAHORS are, unlike those of general PHORS, decidable.

8.2.1 Definitions

We use types defined by the grammar $\theta := o^n \mid \theta \multimap \theta$, where o^n stands for the n-fold product of o, i.e. $\underbrace{o \,\&\, \cdots \,\&\, o}_{n}$ ($n \geq 1$), and $\multimap$ is the linear function space. Let $\mathcal{N}$ be a finite set of typed variables, called *non-terminals*. We use upper-case letters $F, G, \cdots$ to range over $\mathcal{N}$ and write $\mathcal{N}(F)$ for the type of F. We shall consider probabilistic (affine additive) λ-terms with non-terminals from $\mathcal{N}$ and constant (terminal) $\top : o$, which represents termination. The typing judgments $\mathcal{N} \mid \mathcal{E} \vdash t : \theta$, where $\mathcal{E}$ is a partial function from variable names to types and $\mathrm{dom}(\mathcal{E}) \cap \mathcal{N} = \emptyset$, are defined by induction over the rules given in Figure 8.1.

The rules restrict the way in which function parameters (free variables) occur

$$\frac{\mathcal{R}(F) = \lambda x_1 \cdots x_k.t}{F\, t_1 \cdots t_k \xrightarrow{1} t[t_1/x_1, \cdots, t_k/x_k]} \qquad \pi_i\langle t_1, \cdots, t_n\rangle \xrightarrow{1} t_i$$

$$t_1 \oplus_{p_1} \cdots \oplus_{p_n} t_{n+1} \xrightarrow{p_i} t_i \qquad t_1 \oplus_{p_1} \cdots \oplus_{p_n} t_{n+1} \xrightarrow{1-\sum_{i=1}^{n} p_i} t_{n+1}$$

Figure 8.2: PAHORS reduction rules

inside a term. However, non-terminals from $\mathcal{N}$ are excluded from these restrictions. In addition, we include probabilistic branching

$$\frac{\forall i \in \{1, \ldots, n\}.\ \mathcal{N} \mid \mathcal{E} \vdash t_i : o \qquad \sum_{i=1}^{n} p_i < 1}{\mathcal{N} \mid \mathcal{E} \vdash t_1 \oplus_{p_1} \cdots \oplus_{p_n} t_{n+1} : o}$$

with the intention that p and $1-p$ will be the probabilities of choosing the left and right branches respectively. Note that, like in the rule for pairing, the variables in $\mathcal{E}$ can be shared. In contrast, in the rule for function application, they cannot be shared.

Definition 8.5. A ***Probabilistic Additive HORS (PAHORS)*** is a tuple $\mathcal{G} = (\mathcal{N}, \mathcal{R}, S)$, where $\mathcal{N}$ is a finite set of typed non-terminals, $S \in \mathcal{N}$ is the *start symbol* such that $\mathcal{N}(S) = o$, and $\mathcal{R}$ is a function that associates to each $F \in \mathcal{N}$ a term $\mathcal{N} \mid \emptyset \vdash \lambda x_1 \cdots x_k.t : \mathcal{N}(F)$ such that t does not contain λ-abstractions and $\mathcal{N} \mid x_1, \cdots, x_k \vdash t : o$.

Example 8.6. Let $p_1, p_2, p_3 \in \mathcal{Q}$. Consider the PAHORS $\mathcal{G} = (\mathcal{N}, \mathcal{R}, S)$, where $\mathcal{N} = \{S \mapsto o, I \mapsto o \multimap o, F \mapsto (o \multimap o) \multimap o, G \mapsto (o \multimap o) \multimap o \multimap o\}$, and $\mathcal{R}$ is specified below.

$$S = F\,(G\,I) \qquad I\,x = x$$
$$F\,f = F\,(G\,f) \oplus_{p_1} f\,\top \qquad G\,f\,x = f\,(x \oplus_{p_3} \top) \oplus_{p_2} \top$$

Because F, G use functional arguments, $\mathcal{G}$ is classified as order-2.

Remark 8.7. PAHORS subsume order-1 probabilistic HORS [196]. To simulate an order-1 non-terminal $F : \underbrace{o \to \cdots \to o}_{n} \to o$, one can take $F : \underbrace{o \,\&\, \cdots \,\&\, o}_{n} \multimap o$. A typical rule, such as $F\,x\,y = F\,(F\,x\,x)\,(F\,y\,y) \oplus_p x$, can then be simulated by

$$F\,z = F\,\langle F\,\langle \pi_1 z, \pi_1 z\rangle, F\,\langle \pi_2\, z, \pi_2\, z\rangle\rangle \oplus_p (\pi_1\, z).$$

In order to define the termination probability $tp(\mathcal{G})$, we rely on reduction rules of the shape $t \xrightarrow{p} t'$ given in Figure 8.2, where, as usual, we abbreviate $\xrightarrow{1}$ by $\to$.

We write $Red^{\top}(\mathcal{G})$ for the set of *terminating* reduction sequences, i.e. those from S to $\top$. The subset of $Red^{\top}(\mathcal{G})$ consisting of reductions of length not exceeding n will be written $Red^{\top}_{\leq n}(\mathcal{G})$. Similarly, we write $Red^{\top}_{n}(\mathcal{G})$ for the subset of $Red^{\top}_{\leq n}(\mathcal{G})$ comprising reductions of length n. Given $\zeta = (S \xrightarrow{p_1} \ldots \xrightarrow{p_n} \top) \in Red^{\top}_{n}(\mathcal{G})$, we define $tp(\zeta) := \prod_{i=1}^{n} p_i$.

Definition 8.8. The *termination probability* $tp(\mathcal{G})$ of a PAHORS $\mathcal{G}$ is given by the sum over all terminating reductions, namely: $\sum_{\zeta \in Red^{\top}(\mathcal{G})} tp(\zeta)$. Given a PAHORS $\mathcal{G}$, the associated *almost sure termination* (AST) problem asks whether $tp(\mathcal{G}) = 1$.

Definition 8.9. The *expected termination time* $ett(\mathcal{G})$ of a PAHORS $\mathcal{G}$ is $\sum_{n=0}^{\infty}(1 - \sum_{\zeta \in Red^{\top}_{\leq n}(\mathcal{G})} tp(\zeta))$. Given a PAHORS $\mathcal{G}$, the associated *positive almost sure-termination* problem (PAST) asks whether $ett(\mathcal{G})$ is finite.

Remark 8.10. [197] It is known that PAST implies AST, i.e. $tp(\mathcal{G}) < 1$ implies $ett(\mathcal{G}) = \infty$. Moreover, if $tp(\mathcal{G}) = 1$, then $ett(\mathcal{G}) = \sum_{n=0}^{\infty} n \cdot (\sum_{\zeta \in Red^{\top}_{n}(\mathcal{G})} tp(\zeta))$.

By analogy to rPTSA, one can also study threshold problems for $tp(\mathcal{G})$ and $ett(\mathcal{G})$.

Remark 8.11. For $\mathcal{G}$ from Example 8.6, one can show that $tp(\mathcal{G}) = 1$ and $ett(\mathcal{G})$ is finite as long as $p_1 \neq 1$ (otherwise $tp(\mathcal{G}) = 0$ and $ett(\mathcal{G}) = \infty$). For example, for $p_1 = p_2 = p_3 = 0.5$, we have $ett(\mathcal{G}) = \frac{176}{21}$. In the setting of probabilistic HORS, the problem of establishing AST is already undecidable for order-2 probabilistic HORS [196]. Nonetheless, we will show that the AST and PAST problems for PAHORS are decidable, even though the order of schemes in PAHORS is unrestricted.

Remark 8.12. If we replace probabilistic branching $\oplus_{p_1}, \oplus_{p_2}, \oplus_{p_3}$ in $\mathcal{G}$ from Example 8.6 with terminal symbols $a, b, c : o \to o \to o$, we obtain an order-2 recursion scheme featuring the following rules.

$$S = F\ (G\ I) \qquad I\ x = x$$
$$F\ f = a\ (F\ (G\ f))\ (f\ \top) \qquad G\ f\ x = b\ (f\ (c\ x\ \top))\ \top$$

Unfolding the start symbol S will now generate an infinite tree where terminating branches (those that end with $\top$) will have one of the forms: $a^n\,\top$, $a^n\,b^m\,\top$ and $a^n\,b^n\,c^m\,\top$, where $1 \leq m \leq n$. Note that the corresponding word language is not context-free (because of the third case). Consequently, no order-1 probabilistic scheme can mimic the branching structure of $\mathcal{G}$.

Example 8.13. Imagine a game in which the objective is to match the outcome of a coin toss made by the game host, for which the probability of Heads is p_H. To enter the game, the player has to pay an entry fee of £2. To generate answers, the player must bring his own coin (probability p_P). If the outcomes match, the player earns £1 and the game continues. If the outcomes are different, the game ends. The course of the game could be represented by the PAHORS $\mathcal{G}_{p_H,p_P}$ given below.

$$S = F\ U \qquad U\ x = \pi_1 x \oplus_{p_H} \pi_2 x \qquad F\ f = f\ \langle F\ U \oplus_{p_P} \top, \top \oplus_{p_P} F\ U \rangle$$

One can show that $ett(\mathcal{G}) = 1 + \frac{5}{p_H + p_P - 2p_H p_P}$ (each round corresponds to 5 reduction steps and start-up takes 1 step). Given p_H, p_P, it is interesting to check whether the player can expect to win his original £2 back, i.e. whether the expected time to termination is 16 (3 rounds). Among others, this turns out to be the case for $\mathcal{G}_{\frac{1}{4},\frac{1}{6}}$.

8.2.2 rPTSA ⇒ PAHORS

Then, we first present the conversion from rPTSA to PAHORS, ensuring that every configuration of rPTSA is simulated by a term of the PAHORS.

To ease exposition, instead of writing all the rewriting terms at once as:

$$F\ x_1\ \ldots\ x_n = t_1 \oplus_{p_1} \cdots \oplus_{p_m} t_{m+1}$$

we write a single branch of it as:

$$F\ x_1\ \ldots\ x_n \xrightarrow{p_i} t_i$$

We hereby fix a specific k-restricted PTSA $\mathcal{A} = (\mathcal{Q}, \mathcal{M}, \Gamma, \Delta, q_{init}, m_{init})$.

Let us first consider the simulation of the computation in the context of the untyped λ-calculus and then consider adding proper types to the construction.

Tree Stack as Untyped λ-Calculus

The key idea of the rPTSA to PAHORS conversion is an encoding of the rPTSA configuration. The conversion is rather technical, so we present an initial conversion from a configuration of rPTSA $c = (q, t, \rho)$ to a general λ-calculus rewriting rule, temporarily ignoring the typing requirement.

Let us enumerate the rPTSA stack symbol set as $\Gamma = \{\gamma_1, \ldots, \gamma_\ell\}$ for simplicity.

In overview, given a configuration $c = (q, t, \rho)$, our aim is to encode the whole configuration in the following form:

$$\ulcorner q, m, g \urcorner\ t_1\ \ldots\ t_\ell\ t_\Downarrow$$

Here, the symbol $\ulcorner q, m, g \urcorner$ as a whole is a *non-terminal*, encoding the current status $\mathsf{status}(c)$, each t_i corresponds to the sub-tree in and above the current node, and $t_\Downarrow$ marks the whole tree information of t *except* the sub-tree in and above the subject node.

Each tree information t_i is encoded as a tuple of functions, each degree encoding continuing the computation with a specific control state q_i.

The conversion of rules is as follows. For convenience, we use the finite set elements in $\mathcal{Q}$ as indices for projection and directly employ λ-abstraction in the terms (which could be easily resolved by introducing new non-terminals specifically for these terms).

- For an *up* rule:

  $$(q, m, g) \xrightarrow{p} (q', m', up_{\gamma_i})$$

  The λ-calculus encoding is of the following form:

  $$\ulcorner q, m, g \urcorner\ x_1\ \ldots\ x_\ell\ x_\Downarrow \xrightarrow{p}$$
  $$\pi_{q'}\ (x_i\ (\lambda x'_i.\langle \ulcorner q'', m', g \urcorner\ x_1\ \ldots\ x_{i-1}\ x'_i\ x_{i+1}\ \ldots\ x_\ell\ x_\Downarrow \mid q'' \in \mathcal{Q}\rangle))$$

- For a *down* rule:
$$(q, m, g) \xrightarrow{p} (q', m', down)$$
The λ-calculus encoding is of the following form:
$$\ulcorner q, m, g \urcorner\ x_1\ \ldots\ x_\ell\ x_\Downarrow \xrightarrow{p} \pi_{q'}\ (x_\Downarrow\ (\lambda x'_\Downarrow.\langle \ulcorner q'', m', g \urcorner\ \ldots\ x_\ell\ x'_\Downarrow \mid q'' \in \mathcal{Q}\rangle))$$

Typing

The conversion itself is straightforward, but a major technical hurdle is the typing – as HORS requires typing with a simple single base type. Note that the above conversion is not directly valid as P(A)HORS rules due to recursive typing. For example, on the RHS for the *down* rule:
$$\pi_{q'}\ (x_\Downarrow\ (\lambda x'_\Downarrow.\langle \ulcorner q'', m', g \urcorner\ \ldots\ x_\ell\ x'_\Downarrow \mid q'' \in \mathcal{Q}\rangle))$$
Because the type of every non-terminal is fixed, the type of $x_\Downarrow$ and $x'_\Downarrow$ will be the same, which then causes the problem of recursive typing – invalid in simply typed λ-calculus.

To avoid this problem, we should additionally mark the type information. Hence, we differentiate the non-terminals that we use $\langle q, m, g\rangle^{(n_1,\ldots,n_\ell,n_\Downarrow)}$, where each n_i (and/or $n_\Downarrow$) is $\leq k$, the restriction number, marking the rest number of maximum visiting.

With the information of remaining maximum visits, we are ready to type the whole terms. More specifically, the type of a parameter with maximum visiting number n, denoted $\theta^{(n)}$, is recursively defined as:
$$\theta^{(0)} := \underbrace{o\& \ldots \&o}_{|\mathcal{Q}|}$$
$$\theta^{(n+1)} := \theta^{(n)} \multimap \underbrace{o\& \ldots \&o}_{|\mathcal{Q}|}$$

So that a non-terminal $\ulcorner q, m, g \urcorner^{(n_1,\ldots,n_\ell,n_\Downarrow)}$ will be of type:
$$\theta^{(n_1)} \multimap \cdots \multimap \theta^{(n_\ell)} \multimap \theta^{(n_\Downarrow)} \multimap o$$

The conversion rules are as follows:

1. For an *up* rule:
$$(q, m, g) \xrightarrow{p} (q', m', up_{\gamma_i})$$
For all $n_1, \ldots, n_\ell$ and $n_\Downarrow$ within the range $[0, k]$, we have the following rule:
   (a) When $n_i = 0$, the dummy rule:
   $$\ulcorner q, m, g \urcorner^{(n_1,\ldots,n_\ell,n_\Downarrow)}\ x_1\ \ldots\ x_\ell\ x_\Downarrow \xrightarrow{p} \Omega$$
   Here Ω is a special non-terminal standing for infinite looping:
   $$\Omega \to \Omega$$

(b) When $n_i > 0$, we have:

$$\ulcorner q, m, g \urcorner^{(n_1,\ldots,n_\ell,n_\Downarrow)}\ x_1\ \ldots\ x_\ell\ x_\Downarrow \xrightarrow{p} \\ \pi_{q'}\ (x_i\ (\lambda x_i'.\langle F_{q''}\ x_1\ \ldots\ x_{i-1}\ x_i'\ x_{i+1}\ \ldots\ x_\ell\ x_\Downarrow \mid q'' \in \mathcal{Q}\rangle))$$

Here, $F_{q''} := \ulcorner q'', m', g \urcorner^{(n_1,\ldots,n_{i-1},n_i-1,n_{i+1},\ldots,n_\ell,n_\Downarrow)}$.

2. For a *down* rule:

$$(q, m, g) \xrightarrow{p} (q', m', down)$$

For all $n_1, \ldots, n_\ell$ and $n_\Downarrow$ within the range $[0, k]$, we have the following rule:

(a) When $n_\Downarrow = 0$, the following dummy rule is included:

$$\ulcorner q, m, g \urcorner^{(n_1,\ldots,n_\ell,n_\Downarrow)}\ x_1\ \ldots\ x_\ell\ x_\Downarrow \xrightarrow{p} \Omega$$

(b) When $n_\Downarrow > 0$, we have:

$$\ulcorner q, m, g \urcorner^{(n_1,\ldots,n_\ell,n_\Downarrow)}\ x_1\ \ldots\ x_\ell\ x_\Downarrow \xrightarrow{p} \\ \pi_{q'}\ (x_\Downarrow\ (\lambda x_\Downarrow'.\langle \ulcorner q'', m', g \urcorner^{(n_1,\ldots,n_\ell,n_\Downarrow-1)}\ x_1\ \ldots\ x_\ell\ x_\Downarrow' \mid q'' \in \mathcal{Q}\rangle))$$

3. Supportive rules for the initial visit of a node: For every q and g:

$$\ulcorner q, m_{init}, g \urcorner\ x_\Downarrow \to \ulcorner q, m_{init}, g \urcorner^{(\overbrace{k,\ldots,k}^{\ell+1})}\ F_1\ \ldots F_\ell\ x_\Downarrow$$

Here, each F_i is defined as:

$$\lambda x_\Downarrow'.\langle \ulcorner q', m_{init}, \gamma_i \urcorner\ x_\Downarrow' \mid q' \in \mathcal{Q}\rangle$$

The initial visit of a node, with the non-terminal $\ulcorner q, m_{init}, g \urcorner$, will deterministically transition to its transition non-terminal with the maximum remaining calls in each direction initiated to the restriction number k.

Note that at a node, by the k restriction, the *down* visit to the parent node from the subject node is also limited to k.

Additionally, note that the type of $\ulcorner q, m_{init}, g \urcorner$ is naturally:

$$\theta^{(k)} \multimap o$$

4. The initial rule:

$$S \to \ulcorner q_{init}, m_{init}, \bot \urcorner\ (\lambda -.\langle \overbrace{\top,\ldots,\top}^{|\mathcal{Q}|}\rangle)$$

So that when it gets *down* from the bottom, the term will transition to $\top$, representing termination.

Validity of Conversion Given the intuitive explanations, by a straightforward induction, one obtains the following theorem:

Theorem 8.14. *For a given k-PTSA $\mathcal{A}$ and its converted PAHORS $\mathcal{G}$, for every finite run $\mu = c_0 c_1 \dots c_n$ of $\mathcal{A}$, where c_0 is the initial configuration c_{init}, let $\overset{p}{\rightsquigarrow}$ denote a reduction with probability p followed by some probability 1 transitions, then there exists a reduction sequence of $\mathcal{G}$:*

$$S \to t_0 \overset{p_1}{\rightsquigarrow} t_1 \overset{p_2}{\rightsquigarrow} \cdots \overset{p_n}{\rightsquigarrow} t_n$$

such that:

1. *For every i, let $t_i = \ulcorner q, m, g \urcorner^{\neg(-,\dots,-)}$, then $\mathsf{status}(c_i) = (q, m, g)$.*
2. *As a special case, $t_n = \top$ iff $c_n = \top$.*
3. *$\prod_i p_i = wt(\mu)$.*

8.2.3 PAHORS ⇒ rPTSA

Further, we discuss how to relate PAHORS to rPTSA and hence the decision algorithms established for rPTSA are then adaptable to PAHORS. However, unlike the conversions from pPDA to rPTSA and rPTSA to PAHORS, which have clear and intuitive formal connections, the conversion from PAHORS to rPTSA is based on a more complex mechanism. This mechanism is grounded in the conversion from *Multiplicative Additive HORS* (MAHORS) [142], the non-probabilistic version of PAHORS, to rTSA, the non-probabilistic version of rPTSA. In this section, we progressively introduce the conversions for deciding termination probability, expected termination time, and model checking. Each conversion builds upon the previous one, gradually revealing the complete process.

For convenience, we assume that every PAHORS rule discussed below has at most one probabilistic choice $\oplus_p$, so that we may refer to the terms as left (L) and right (R), respectively.

Remark 8.15. In this part, for convenience, we may include *horizontal transitions* in our discussion, whose semantics are discussed in Remark 3.5. These can be eliminated by methods to be discussed in Section 9.3.2.

Let $\mathcal{G} = (\mathcal{N}, \mathcal{R}, S)$ be a PAHORS. As mentioned, the connection to rPTSA is based on a relationship between the deterministic version of rPTSA and a class of recursion schemes called MAHORS. MAHORS are tuples $(\Sigma, \mathcal{N}, \mathcal{R}, S)$, where Σ is a finite alphabet of terminals $a : o^n \multimap o$ and $\mathcal{N}, \mathcal{R}, S$ are defined as for PAHORS, except that probabilistic choice is disallowed. S is then reduced using (unweighted versions of) rules for unfolding non-terminals and projecting in contexts $C := \bullet \mid a\langle t_1, \cdots, t_i, C, t_{i+1}, \cdots, t_n\rangle$. Let a *ranked* alphabet be a function that maps each element of the alphabet to a natural number. Thus, MAHORS generate a potentially infinite tree over a finite ranked alphabet of terminals, whereas PAHORS do not generate any structure at all (indeed termination is the

only observable aspect of PAHORS computation). In what follows, we recast the termination probability of a PAHORS as a summation over terminating branches of a tree generated by a MAHORS.

From PAHORS to MAHORS

The translation is based on replacing probabilistic branching $\oplus_p$ with a terminal $a_p : o \,\&\, o \multimap o$. Let $\mathcal{Q}_{\mathcal{G}}$ be the set of rational numbers that occur in the rules of $\mathcal{G}$. Define the alphabet $\Sigma_{\mathcal{G}}$ to consist of terminals $a_p : o \,\&\, o \multimap o$ for each $p \in \mathcal{Q}_{\mathcal{G}}$. We write $\bar{t}$ for the term t in which each subterm of the form $u_1 \oplus_p u_2$ is replaced by $a_p\langle u_1, u_2\rangle$ inductively. Let $\overline{\mathcal{R}(F)}$ be $\mathcal{R}(\overline{F})$. Then $\overline{\mathcal{G}} = \langle \Sigma_{\mathcal{G}} \cup \{\top\}, \mathcal{N}, \overline{\mathcal{R}}, S\rangle$ is a MAHORS. Accordingly, one can view $\mathcal{T}_{\overline{\mathcal{G}}}$ as a partial function from $\{L, R\}^*$ to $\Sigma_{\mathcal{G}} \cup \{\top\}$ with a prefix-closed domain. Let $Top(\mathcal{T}_{\overline{\mathcal{G}}}) = \{\pi \in \{L, R\}^* \mid \mathcal{T}_{\overline{\mathcal{G}}}(\pi) = \top\}$. Given $\pi = \ell_1 \cdots \ell_k \in Top(\mathcal{T}_{\overline{\mathcal{G}}})$, with $\ell_i \in \{L, R\}$, let $a_{p_1} \cdots a_{p_k} \top$ be the corresponding branch of $\mathcal{T}_{\mathcal{G}'}$. Let the corresponding weight $tp(\overline{\mathcal{G}}), \pi$ be $\prod_{i=1}^{k} p'_i$, where $p'_i = p_i$ (if $\ell_i = L$) and $p'_i = 1 - p_i$ (if $\ell_i = R$). Notice that there is a 1-1 correspondence between maximal reduction sequences of the PAHORS $\mathcal{G}$ and maximal branches of the tree $\mathcal{T}_{\overline{\mathcal{G}}}$. Moreover we have $tp(\mathcal{G}) = \sum_{\pi \in Top(\mathcal{T}_{\overline{\mathcal{G}}})} tp(\overline{\mathcal{G}}), \pi$.

Lemma 8.16. *For any PAHORS $\mathcal{G}$, there exists an rPTSA whose termination probability is $tp(\mathcal{G})$.*

Proof. We take advantage of the correspondence between PAHORS and MAHORS. Consider the MAHORS $\overline{\mathcal{G}}$. In [142], it was shown to construct a restricted *tree-generating* tree-stack automaton that generates the same tree as $\overline{\mathcal{G}}$. In particular, the automaton will have transitions of the form $\Delta(q, m, g) = a_p(q_1, q_2)$ and $\Delta(q, m, g) = \top$ associated with terminals of $\overline{\mathcal{G}}$. To obtain an rPTSA, we replace $\Delta(q, m, g) = a_p(q_1, q_2)$ with $(q, m, g) \xrightarrow{p} q_1$ and $(q, m, g) \xrightarrow{1-p} q_2$. To handle $\Delta(q, m, g) = \top$, we add a new state q_{term} and transitions that will trigger a cascade of *down*'s: $(q, m, g) \to q_{term}$ and $(q_{term}, m', g') \to (q_{term}, m', \mathit{down})$ for any stack symbol g' and memory m'. By earlier remarks, the resultant rPTSA has the same termination probability as $tp(\mathcal{G})$. □

The result makes it possible to apply techniques developed in Chapter 4 for rPTSA for termination probability analysis to PAHORS. In particular, we have:

Theorem 8.17. *The AST problem and the threshold problem for PAHORS are decidable.*

From PAHORS to MAHORS with steps

Next we will consider another conversion of PAHORS to MAHORS which, in addition to generating a tree corresponding to the structure of probabilistic branching, will also keep track of the number of reduction steps related to unfolding non-terminals and the projection rules. For short, we shall call them *u/p rules*.

This will be done with the help of a dedicated terminal $\mathit{step} : o \multimap o$. Given a term $\mathcal{N} \mid \emptyset \vdash t : o$, let $\hat{t}$ be t in which each subterm of the form $u_1 \oplus_p u_2$ is

replaced with $a_p\langle u_1, u_2\rangle$ (as in $\bar{t}$) and each subterm of the form $\pi_i\, u$ is replaced with $\mathit{step}(\pi_i u)$ (all in an inductive manner).

Definition 8.18. Given a PAHORS $\mathcal{G} = \langle \mathcal{N}, \mathcal{R}, S\rangle$, let $\hat{\mathcal{G}} = \langle \Sigma_{\mathcal{G}} \cup \{\top, \mathit{step}\}, \hat{\mathcal{R}}, S\rangle$ be a MAHORS in which, given $\mathcal{R}(F) = \lambda x_1 \cdots x_k.t$, we set $\hat{\mathcal{R}}(F) = \lambda x_1 \cdots x_k.\mathit{step}(\hat{t})$.

Let us write $\longrightarrow_H$ for a u/p step of $\hat{\mathcal{G}}$ executed in an evaluation context H, defined by $H := \bullet \mid \mathit{step}\, H$. Then $\hat{\mathcal{G}}$ turns out to track the number of u/p steps in $\mathcal{G}$ in the following sense.

Lemma 8.19. *Suppose $\mathcal{G} = \langle \mathcal{N}, \mathcal{R}, S\rangle$ is a PAHORS and $\mathcal{N} \vdash t : o$. We have $t \xrightarrow{\epsilon,1}{}^{n} t'$ via u/p steps if and only if $\hat{t} \longrightarrow_H^n \mathit{step}^n(\hat{t'})$.*

Proof. Observe that $t \xrightarrow{\epsilon,1} t'$ via a u/p step if and only if $\hat{t} \longrightarrow_H \mathit{step}(\hat{t'})$. To conclude the Lemma, it suffices to iterate the above observation, because $\longrightarrow_H$ permits reductions under *step*. □

Lemma 8.19 shows that u/p steps in $\mathcal{G}$ are faithfully represented in $\hat{\mathcal{G}}$. In particular, an infinite sequence of u/p steps in $\mathcal{G}$ will generate infinitely many occurrences of *step*. On the other hand, if a sequence of u/p steps in $\mathcal{G}$ cannot be extended with another u/p step then the reduction must have encountered $\top$ or $u_1 \oplus_p u_2$. In this case, by Lemma 8.19, the corresponding $\longrightarrow_H^*$ reduction in $\hat{\mathcal{G}}$ will get stuck at $H[\top]$ or $H[a_p\langle \hat{u_1}, \hat{u_2}\rangle]$ respectively, for some context H. In the former case, both $\mathcal{G}$ and $\hat{\mathcal{G}}$ are stuck. In the latter case, in $\mathcal{G}$ the reduction will continue from u_1 or u_2. Correspondingly, in $\hat{\mathcal{G}}$ we can start reducing $\hat{u_1}$ or $\hat{u_2}$ (using $\longrightarrow_H$) respectively. Consequently, reduction sequences in $\mathcal{G}$ are in 1-1 correspondence with branches of $\hat{\mathcal{G}}$. We will rely on $\hat{\mathcal{G}}$ when calculating $\mathit{ett}(\mathcal{G})$. Note that the definitions of $\mathit{ett}(\cdot)$ are slightly different for rPTSA and PAHORS, but they coincide if $\mathit{tp}(\mathcal{G}) = 1$ (Remark 8.10). Recall that $\mathit{tp}(\mathcal{G}) < 1$ implies $\mathit{ett}(\mathcal{G}) = \infty$.

Lemma 8.20. *For any PAHORS $\mathcal{G}$ with $\mathit{tp}(\mathcal{G}) = 1$, there exists an rPTSA whose expected time to termination is $\mathit{ett}(\mathcal{G})$.*

Proof. This time, we exploit the MAHORS $\hat{\mathcal{G}}$. Via [142], we can obtain a restricted *tree-generating* tree-stack automaton that generates the same tree as $\hat{\mathcal{G}}$. In particular, the automaton will have transitions of the form $\Delta(q, m, g) = \mathit{step}(q')$, $\Delta(q, m, g) = a_p(q_1, q_2)$ and $\Delta(q, m, g) = \top$.

To obtain an rPTSA, we replace them with horizontal transitions. To handle $\Delta(q, m, g) = \mathit{step}(q')$, we add $(q, m, g) \to q'$. For $\Delta(q, m, g) = a_p(q_1, q_2)$, we add $(q, m, g) \xrightarrow{p} q_1$ and $(q, m, g) \xrightarrow{1-p} q_2$; for $\Delta(q, m, g) = \top$, we add a new state q_{term} along with $(q, m, g) \to q_{\mathit{term}}$ and $(q_{\mathit{term}}, m', g) \to (q_{\mathit{term}}, m', \mathit{down})$ for any stack status g and memory m'.

Finally, to compute the correct expected termination time value, we may adopt the expected total accumulated reward computation for the converted rPTSA. Note that we shall adopt a simple reward function f that counts the first two cases of rules above. Note also that the extra *down* transitions related to termination will not be counted. By Lemma 8.19 and the following discussion, the expected total

accumulated reward in the resultant rPTSA with the simple reward function f will coincide with $ett(\mathcal{G})$. □

The above result enables us to bring the results developed in Sections 4.3 and 4.4 to bear on PAHORS. Recall that before referring to rPTSA, it is necessary to check $tp(\mathcal{G}) = 1$, which can be done using Theorem 8.17. If $tp(\mathcal{G}) < 1$, $\mathcal{G}$ can be classified immediately as a negative instance of PAST.

Theorem 8.21. *The PAST problem for PAHORS is decidable.*

From Steps to Atomic Formulas

Using similar conversions to the above-mentioned PAHORS-to-rPTSA translations, one can also deploy our results on the model checking algorithms developed in Chapter 5 and Chapter 6 in the setting of PAHORS. In order to specify interesting properties of PAHORS, we can reintroduce terminals (in the spirit of *step*), which will map to special states using the translation from [142]. One can then use these states in the model checking specifications. In turn, access to Γ gives DRA the ability to talk about the current active procedure (non-terminal).

The above translations to rPTSA can be implemented in polynomial time, because the underpinning translation in the tree-generating case has this complexity [142, Theorem 7]. Also, the restriction parameter k is polynomial in the size of $\mathcal{G}$. Consequently, the exponential-space complexity extends to decision problems for PAHORS. Using the techniques of [142], it is also possible to translate rPTSA to PAHORS.

Remark 8.22 (1-P(A)HORS to pPDA)**.** As stated, order-1 PHORS is equivalent to order-1 PAHORS. According to [102], the conversion from order-1 MAHORS is guaranteed to result in 1-restricted TSA, by the nature of linear game semantics, which, in our context, ensures that 1-PAHORS converts to 1-PTSA.

Therefore, to convert 1-PHORS to pPDA, one should first convert 1-PHORS to 1-PAHORS, which then translates to 1-PTSA and, finally, to pPDA.

8.3 MCFG and Stochastic Grammars

Finally, we introduce the relationship between rPTSA and Multiple Context-Free Grammars (MCFG) and its probabilistic version.

8.3.1 MCFG and MCFL

In this section, we formally present the formalism of MCFG and its related definitions.

Throughout this section, we fix a distinct countably infinite set of variables $\mathcal{X}$. We use α or β to refer to sequences of symbols. We denote the concatenation of sequences α and β by writing $\alpha\beta$.

Definition 8.23 (Multiple Context-Free Grammar)**.** A Multiple Context-Free Grammar (MCFG) is a tuple (Σ, N, R, S), where Σ is a finite set whose elements are called terminals, N is a ranked alphabet whose elements are called non-terminals, R is a finite set of rules, whose elements are in the form:

$$F(\alpha_1, \ldots, \alpha_{N(F)}) \leftarrow F_1(x_1^{(1)}, \ldots, x_{N(F_1)}^{(1)}), \ldots, F_m(x_1^{(m)}, \ldots, x_{N(F_m)}^{(m)})$$

where we intuitively divide the rule by "$\leftarrow$" and call the left part the "left-hand side" (LHS), and the right part the "right-hand side" (RHS), F and $F_1, \ldots, F_m$ are all non-terminals, each $x_i^{(j)}$ on the RHS is a distinct variable in $\mathcal{X}$, $\alpha_1, \ldots, \alpha_{N(F)}$ on the LHS are all sequences of terminals and variables on the RHS. Specifically, we require that every single variable on the RHS can appear at most once in the sequence $\alpha_1 \ldots \alpha_{N(F)}$. Finally, S is a special non-terminal called the starting non-terminal, with $N(S) = 1$.

Given a rule r, generally denoted as:

$$F(\alpha_1, \ldots, \alpha_{N(F)}) \leftarrow F_1(x_1^{(1)}, \ldots, x_{N(F_1)}^{(1)}), \ldots, F_m(x_1^{(m)}, \ldots, x_{N(F_m)}^{(m)})$$

we call m the *degree* of the rule. We explicitly denote a grammar to be n-MCFG(d), if for all non-terminals $F \in N$, $N(F) \leq n$ and the maximum degree of all rules is $\leq d$. We also denote $\mathsf{rank}(\mathcal{G})$ as the (minimum) rank of $\mathcal{G}$, and denote $\mathsf{degree}(\mathcal{G})$ as the (minimum) degree of $\mathcal{G}$. Sometimes, when we only want to emphasise the rank, we write n-MCFG for $\bigcup_{i \in \mathbb{N}} n$-MCFG($i$).

For a tuple of the form $F(\beta_1, \ldots, \beta_{N(F)})$, we say the ith *dimension* to refer to the ith position of the tuple, namely β_i.

Next, we define the language defined by a given MCFG. Firstly, we define the concept of *derivation*. The object of a derivation is a non-terminal with a tuple of strings that consist of pure terminals with length exactly that of the rank of the non-terminal, written $F(\beta_1, \ldots, \beta_{N(F)})$. If there is a rule in R with an empty RHS, that is, of the form:

$$F(\alpha_1, \ldots, \alpha_{N(F)}) \leftarrow \varepsilon$$

we say $F(\alpha_1, \ldots, \alpha_{N(F)})$ is derivable. Then, if the string tuples $F_1(\beta_1^{(1)}, \ldots, \beta_{N(F_1)}^{(1)})$, $\ldots$, $F_m(\beta_1^{(m)}, \ldots, \beta_{N(F_m)}^{(m)})$ are all derivable, and there is a rule in R of the form:

$$F(\alpha_1, \ldots, \alpha_{N(F)}) \leftarrow F_1(x_1^{(1)}, \ldots, x_{N(F_1)}^{(1)}), \ldots, F_m(x_1^{(m)}, \ldots, x_{N(F_m)}^{(m)})$$

we say $F(\beta_1, \ldots, \beta_{N(F)})$ is derivable, where β_k for all $k \in \{1, \ldots, N(F)\}$ is given by: $\beta_k := \alpha_k[x_i^{(j)} \leftarrow \beta_i^{(j)}]$, for all $j \in \{1, \ldots, m\}$, $i \in \{1, \ldots, N(F_j)\}$. Then, a derivation tree of a derivable tuple of strings is simply a proof tree by applying the above two inductive rules of derivability.

Moreover, we say the rule is *non-deleting* if every variable on the RHS appears *exactly once* in the sequence $\alpha_1 \ldots \alpha_{N(F)}$. Further, we say the whole grammar $\mathcal{G}$ is non-deleting iff all of its rules are non-deleting.

Then, the language defined by a given MCFG with starting non-terminal S is defined by: $\mathcal{L} := \left\{ w \,\middle|\, S(w) \text{ is derivable in the grammar} \right\}$. We call the languages definable through an MCFG to be *multiple context-free languages (MCFL)*, likewise for other qualified MCFG, e.g., n-MCFL(r).

$$
\begin{array}{rcl|rcl}
(q_f, m_{init}, \gamma_\perp) & \to & (q_f, m_F, up_\gamma) & (q_d, m_F, \gamma_\perp) & \to & (q_s, m_S, up_\gamma) \\
 & & & & & \\
(q_f, m_{init}, \gamma) & \xrightarrow{A} & (q_f, m_A, up_\gamma) & (q_s, m_A, \gamma) & \xrightarrow{A} & (q_s, m_R, up_\gamma) \\
(q_f, m_{init}, \gamma) & \xrightarrow{B} & (q_f, m_B, up_\gamma) & (q_s, m_B, \gamma) & \xrightarrow{B} & (q_s, m_R, up_\gamma) \\
(q_f, m_{init}, \gamma) & \to & (q_d, m_T, down) & (q_s, m_T, \gamma) & \to & (q_d, m_R, down) \\
(q_d, m_A, \gamma) & \to & (q_d, m_A, down) & (q_d, m_R, \gamma) & \to & (q_d, m_R, down) \\
(q_d, m_B, \gamma) & \to & (q_d, m_B, down) & (q_d, m_S, \gamma_\perp) & \to & (q_d, m_R, down)
\end{array}
$$

Figure 8.3: Rules of rTSA Generating the COPY Language

8.3.2 Mutual Conversions

Next, we discuss the mutual conversion between the restricted tree-stack automaton (rTSA) and MCFG, demonstrating the equivalence between these formalisms. In exploring the mutual conversions between MCFGs and tree-stack automata, it is essential to first define a string-generating rTSA, as MCFGs are inherently string generators.

String-Generating rTSA

A string-generating rTSA essentially functions as an rPTSA with its rules map Δ modified. Recall that an ordinary rule in Δ of an rPTSA looks like: $(q, m, g) \xrightarrow{p} op$. Now, as we are simply generating string α and are not concerned with the probability p, we will need a rule to have the form: $(q, m, g) \xrightarrow{\alpha} op$. Intuitively, the modification to the configurations remains the same, but now, instead of probabilistically executing the operation, it non-deterministically executes the operation and produces a string.

Given the close relationship between rTSA and rPTSA, we will adopt the terminologies and concepts of rPTSA for string-generating rTSA to avoid redundancy.

Definition 8.24. A *string-generating rTSA* is a tuple $(\mathcal{Q}, \mathcal{M}, \Gamma, \Sigma, \Delta, q_{init}, m_{init})$, where $\mathcal{Q}$, $\mathcal{M}$, Γ, q_{init}, and m_{init} retain their meaning from those in rPTSA. Σ represents an (unranked) alphabet. The function $\Delta : \mathcal{Q} \times \mathcal{M} \times (\Gamma \uplus \{\perp\}) \to \mathbf{Op} \rightharpoonup \Sigma^*$ defines the rules. Analogous to rPTSA, a rule δ is denoted as $(q, m, g) \xrightarrow{\alpha} op$, signifying that $\Delta(q, m, g)(op) = \alpha$, thus implying $op \in \mathrm{dom}(\Delta(q, m, g))$. This time, we abbreviate $\xrightarrow{\varepsilon}$ as $\to$. Similarly, $\alpha^{(\delta)}$ is used to extract the string α from the rule δ.

Example 8.25. The rules of string-generating version of 2-TSA that generates the COPY language $\{ww \mid w \in (A|B)^*\}$ are presented in Figure 8.3. The general idea is identical to that of the probabilistic simulation in Example 3.8, where the first w is printed and stored on the tree-stack, and in the second pass from bottom to top, the string w is reproduced.

For convenience, we shall henceforth fix a specific string-generating rTSA $\mathcal{A} = (\mathcal{Q}, \mathcal{M}, \Gamma, \Sigma, \Delta, q_{init}, m_{init})$ throughout the rest of the section.

The concepts of configurations, initial configuration, and status are identical to those in rPTSA. The definition of transitions $c \xrightarrow{\delta} c'$ is naturally derived from rPTSA. Like in rPTSA, the rule δ is uniquely determined given the initial configuration c and the subsequent configuration c'. Thus, the concepts of run, pre-run, path, pre-path, and initial run are similarly derived.

A string α is said to be *generated* by $\mathcal{A}$ if and only if there exists an initial run and terminating μ such that $\alpha = \mathsf{concat}([\alpha^{(\delta)} \mid \delta \in \mathsf{path}(\mu)])$. The *language* of $\mathcal{A}$ is the set of all strings generated by $\mathcal{A}$.

MCFG ⇒ rTSA

To begin with, we discuss how to convert an MCFG to a string-generating rTSA. The key idea behind the construction is to use the tree-stack structure to simulate the structure of the derivation tree of the strings from the MCFG. A run of the rTSA then essentially walks along the derivation tree of a string derivable from the host MCFG and prints out the terminals along the walk.

To encode this walk, we take advantage of the local memory at each node on the tree-stack to keep track of the information of which rule to apply and the progress of the printing-out, with the signal of the visit passed along by the control-states.

Formally, given an MCFG $\mathcal{G} = (\Sigma, N, R, S)$, let $N = \{F_1, \ldots, F_n\}$, then we construct an rTSA $\mathcal{A} = (\mathcal{Q}, \mathcal{M}, \Gamma, \Sigma, \Delta, \mathsf{Dim}(1), m_{init})$ that generates the same language as $\mathcal{G}$ where:

- $\mathcal{Q}$ is $\{\mathsf{Dim}(i) \mid i \in \{0, 1, \ldots, \mathsf{rank}(\mathcal{G})\}\}$. Each element represents the number of dimension.
- $\mathcal{M}$ is either m_{init}, a placeholder for a newly created node, m_{end}, a special marker, or elements of the form (r, α), where r is a rule of $\mathcal{G}$ that is of the form:
$$F(\alpha_1, \ldots, \alpha_n) \leftarrow F_1(\ldots), \ldots, F_m(\ldots)$$
and α is a postfix of one of α_i for $i \in \{1, \ldots, n\}$. Each pair represents a rule r and the current reading progress α.
- Γ is $\{1, \ldots, \mathsf{degree}(\mathcal{G})\} \times \mathrm{dom}(N)$, where n is the maximum number of occurrences of a non-terminal on the RHS across all rules of $\mathcal{G}$.
- Σ is the same as that in $\mathcal{G}$.
- The initial control state is $\mathsf{Dim}(1)$.
- The initial local memory is m_{init}.

Further, the transition Δ contains the following rules (all states not mentioned are never visited and hence can refer to any rule).

The first part consists of the auxiliary rules at the bottom:

- The initial rule $(\mathsf{Dim}(1), \mathscr{m}_{init}, \bot) \to (\mathsf{Dim}(1), \mathscr{m}_{end}, up_{(1,S)})$; and,
- For all $\mathscr{q} \in \mathcal{Q}$, $(\mathscr{q}, \mathscr{m}_{end}, \bot) \to (\mathsf{Dim}(1), \mathscr{m}_{end}, down)$ which leads to termination.

Then, for each rule r of MCFG:

$$F(\alpha_1, \ldots, \alpha_n) \leftarrow F_1(\ldots), \ldots, F_m(\ldots)$$

We define a function $\mathsf{GenNext}(\alpha)$ for all strings α that are postfixes of some α_i at the rule. If α contains a variable, let $\alpha = \mathsf{a}_1 \ldots \mathsf{a}_\ell x_j^{(i)} \alpha'$, where $\mathsf{a}_i \in \Sigma$, the function $\mathsf{GenNext}(\alpha)$ returns a triple $(\mathsf{a}_1 \ldots \mathsf{a}_\ell, (c, F_i, j), \alpha')$, where c represents the cth occurrence of the same non-terminal F_i on the RHS of r. Otherwise, the function returns a triple $(\alpha, \Downarrow, \varepsilon)$.

Then, for all $c \in \{1, \ldots, \mathsf{degree}(\mathcal{G})\}$, the following rTSA rules for stack status (c, F) are in $\mathcal{A}$:

- For all $i \in \{1, \ldots, N(F)\}$, if $\mathsf{GenNext}(\alpha_i) = (\beta, (c', F', j), \alpha')$ for some c', F', the rule:
$$(\mathsf{Dim}(i), \mathscr{m}_{init}, (c, F)) \xrightarrow{\beta} (\mathsf{Dim}(j), (r, \alpha'), up_{(c',F')})$$
and the rule:
$$(\mathsf{Dim}(i), (r, \varepsilon), (c, F)) \xrightarrow{\beta} (\mathsf{Dim}(j), (r, \alpha'), up_{(c',F')})$$
Here, it means to visit the jth dimension of the c'th non-terminal F' upward, while leaving the rest of the string α' for further exploration after getting back *down* at this node.
Otherwise, if $\mathsf{GenNext}(\alpha_i) = (\alpha, \Downarrow, \varepsilon)$, the rule:
$$(\mathsf{Dim}(i), \mathscr{m}_{init}, (c, F)) \xrightarrow{\alpha} (\mathsf{Dim}(0), (r, \varepsilon), down)$$
and the rule:
$$(\mathsf{Dim}(i), (r, \varepsilon), (c, F)) \xrightarrow{\alpha} (\mathsf{Dim}(0), (r, \varepsilon), down)$$
For the latter two, the *down* with control state $\mathsf{Dim}(0)$ is just a placeholder representing a visit back from above.
- For visits from above – the control state is $\mathsf{Dim}(0)$, if there is some c', F' such that $\mathsf{GenNext}(\alpha) = (\beta, (c', F', j), \alpha')$, then the following rule is included:
$$(\mathsf{Dim}(0), (r, \alpha), (c, F)) \xrightarrow{\beta} (\mathsf{Dim}(j), (r, \alpha'), up_{(c',F')})$$
Otherwise, if $\mathsf{GenNext}(\alpha) = (\alpha, \Downarrow, \varepsilon)$, the rule:
$$(\mathsf{Dim}(0), (r, \alpha), (c, F)) \xrightarrow{\alpha} (\mathsf{Dim}(0), (r, \varepsilon), down)$$

With the formal construction, we may explain the simulation in more details. This construction intuitively allows the local memory to retain the information of which rule r to adhere to at this node, along with the information of the remaining string to be read at this dimension when getting *up* to explore a dimension of one of its RHS non-terminal. The rule to follow is determined during the creation of this node.

Then, at each visit, either during the first visit (m_{init}) of the node, or subsequent visits from below (when the local memory is (r, ε)), it prints out the preceding terminals at the required dimension i of the rule, hinted by the control state $\mathsf{Dim}(i)$, keeps track of the remaining string, and then proceeds to the next position indicated by the string. If this position stands a variable $x_j^{(i)}$, it means we should generate the string at the corresponding dimension j and non-terminal F_i indicated by the variable. Otherwise, if the remaining string is empty, it means the printing has ended, and it should return to the parent node below. The same procedure is repeatedly applied to the rest of the string α kept in the local memory when the control state is $\mathsf{Dim}(0)$, hinting at a visit from above.

For a detailed proof of the conversion, we refer to [99].

rTSA ⇒ MCFG

The conversion from a string-generating rTSA to an MCFG is more intricate than the intuitive conversion from MCFG to rTSA. Nevertheless, the proposed equation system facilitates this task.

Given the strong resemblance between string-generating rTSA and rPTSA, it is verifiable that the equation for pre-paths Equation (4.21) applies to string-generating rTSA as well.

Let us define $\mathsf{Str}(\phi)$ to concatenate the strings of terminals carried across all rules, and $\mathsf{Str}(\vec{\phi})$ to apply to each path within, respectively. This definition, as always, also extends to sets of such structures as pointwise application.

By a direct application of $\mathsf{Str}(-)$ (as an extended function applied to a set of objects) on both sides of Equation (4.21), one can readily derive that [1]:

$$\mathsf{Str}(\mathsf{path}(v)) = \bigcup_{\mathbb{D} \sim: v} \left\{ \mathsf{Str}\Big(\mathsf{Sync}\big(\pi, \{\gamma \mapsto \vec{\phi}_\gamma \mid \gamma \in \Gamma\}\big)\Big) \;\middle|\; \begin{array}{c} \pi \in \mathit{SynPaths}^{\mathbb{D}}_{(\gamma_0, D)} \\ \forall \gamma \in \Gamma. \vec{\phi}_\gamma \in \mathsf{path}(V^\gamma_{\mathbb{D}}) \end{array} \right\} \tag{8.26}$$

where $v := \mathit{Trips}^{\gamma_0}_D$ and $V^\gamma_{\mathbb{D}} := \mathit{Trips}^\gamma_{\mathbb{D}(\gamma)}$.

Notice that in:

$$\mathsf{Str}\Big(\mathsf{Sync}\big(\pi, \{\gamma \mapsto \vec{\phi}_\gamma \mid \gamma \in \Gamma\}\big)\Big)$$

the function $\{\gamma \mapsto \vec{\phi}_\gamma \mid \gamma \in \Gamma\}$ can be substituted with the corresponding sequence of variables, which gives rise to a string tuple.

[1] Note that the results between each $\mathbb{D}$ may not be disjoint, hence $\uplus$ is replaced with $\cup$.

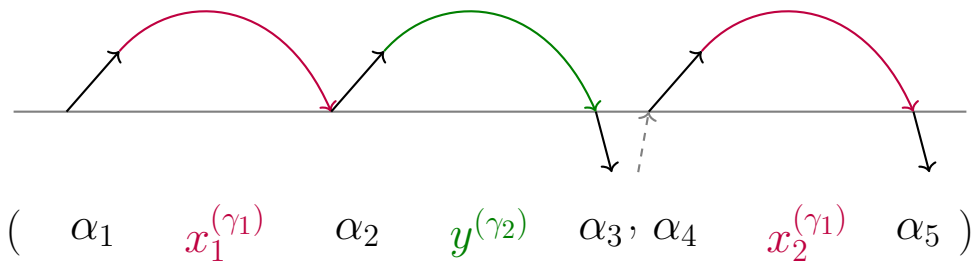

Figure 8.4: Path Synchronisation to Tuple with Variable

For instance, let us examine an example as displayed in Figure 8.4. The upper curves illustrate the entire pre-path sequence as a result of synchronising the synchronisation path (the black arrows), and the upper transitions at γ_1 (purple) and γ_2 (green). This synchronisation can be interpreted as forming a string tuple as depicted in the below half of the figure, where the strings carried along each transition on the synchronisation path are printed (i.e. $\alpha_1, \ldots, \alpha_5$), with the strings at the upper nodes substituted by corresponding variables as placeholders.

More specifically, this implies that, given a string-generating rTSA, one can construct an MCFG that utilises the *Trips* patterns of the rTSA as non-terminals, and for each synchronisation path, there will be a corresponding rule by substituting the upward interactions with the corresponding variables.

Formally, for a string-generating rTSA $\mathcal{A} = (\mathcal{Q}, \mathcal{M}, \Gamma, \Sigma, \Delta, q_{init}, m_{init})$, we can construct an MCFG $\mathcal{G} = (\Sigma, N, R, S)$ that generates the same language as $\mathcal{A}$. For simplicity, we also assume, as in the termination chapter (Chapter 4), that $\mathcal{A}$ has a special proxy bottom stack symbol $\gamma_\perp \in \Gamma$ such that the real bottom symbol simply goes *up* to $\gamma_\perp$ and then terminates after getting *down* from $\gamma_\perp$. This assumption allows us to discuss *Trips* patterns easily and uniformly.

- Σ is identical to $\mathcal{A}$.
- N consists of S along with all the *Trips* patterns of $\mathcal{A}$.
- S is a unique symbol for the start symbol.

The rules set R is also divided into two parts. The first part pertains to the starting non-terminal S such that for all $q \in \mathcal{Q}$:

$$S(x) \leftarrow \mathit{Trips}^{\gamma_\perp}_{[q_{init}, q]}(x)$$

The second part contains rules for each *Trips* pattern $v := \mathit{Trips}^{\gamma_0}_D$. Then, for each $\mathbb{D} \sim : v$, and each $\pi \in \mathit{SynPaths}^{\mathbb{D}}_{(\gamma, D)}$, there is a rule [2]:

$$\mathit{Trips}^{\gamma_0}_D(\alpha_1, \ldots, \alpha_{\frac{|D|}{2}}) \leftarrow \forall \gamma \in \gamma. \mathit{Trips}^{\gamma}_{\mathbb{D}(\gamma)}(x_1^{(\gamma)}, \ldots, x^{(\gamma)}_{\frac{|\mathbb{D}(\gamma)|}{2}})$$

[2]Naturally, we shall ignore patterns $\mathit{Trips}^-_\varepsilon$ on the RHS.

Here, the sequence $\alpha_1, \ldots, \alpha_{\frac{|D|}{2}}$ is generated from π such that for each *up* visit, we leave a variable referring to the corresponding RHS pattern (non-terminal) uniquely indexed by γ. We can formalise this as $\mathsf{RuleGen}(\pi, \{\gamma \mapsto 1 \mid \gamma \in \Gamma\})$, where $\mathsf{RuleGen}(-,-)$ is defined as:

$$\mathsf{RuleGen}(\varepsilon, -) := \varepsilon$$

$$\mathsf{RuleGen}(\delta :: \pi', f) := \begin{cases} \alpha^{(\delta)} :: \mathsf{RuleGen}(\pi', f) & \kappa(\delta) = \Downarrow \\ \mathsf{hdApp}(\alpha^{(\delta)} +\!\!+ [x_\gamma^{(f(\gamma))}], \mathsf{RuleGen}(\pi', f[\gamma \mapsto f(\gamma)+1])) & \kappa(\delta) = \gamma \in \Gamma \end{cases}$$

Here, $\mathsf{hdApp}(\alpha, s)$ appends α to the first sequence of s.

With the construction, the above observation as shown in Figure 8.4 could be further formalised as the following lemma:

Lemma 8.27. *A tuple of string $Trips_D^\gamma(\beta_1, \ldots, \beta_{\frac{|D|}{2}})$ is derivable iff there is a pre-run $\vec{\mu} \in Trips_D^\gamma$ such that $\mathsf{Str}(\mathsf{path}(\vec{\mu})) = [\beta_1, \ldots, \beta_{\frac{|D|}{2}}]$.*

Proof. Notice that the height of the derivation tree (e.g. Figure 2.9) as a proof to a derivable tuple of string corresponds exactly to the maximum height above the subject node of the pre-run. Hence, for the forward direction, induction on the height of the derivation tree. For the reverse direction, induction on the maximum height that $\vec{\mu}$ could reach. The inductive steps of both cases follow instantly from Equation (8.26). □

As a corollary, the following correctness could be stated.

Theorem 8.28. *$\mathcal{G}$ generates the same language as $\mathcal{A}$.*

8.3.3 Stochastic Grammars

Finally, we present a brief discussion on adding probability to MCFG and the termination problem thereof.

SMCFG

A Stochastic Multiple Context-Free Grammar (SMCFG) $\mathcal{G} = (\Sigma, N, R, S)$ is simply an MCFG where each rules is additionally attached with a probability p, usually written on the arrow of the rule. The natural stochastic requirement is: $0 < p \leq 1$ and the sum of probabilities of all rules for a single non-terminal equals 1. That is, for every non-terminal F:

$$\sum_{\{F(\cdots) \xleftarrow{p} \cdots \in R\}} p = 1$$

The other notions are inherited directly from MCFG.

To define the termination probability, we present a linear way of delineating *reduction*. A *term* of the SMCFG, denoted by t, is in the very same form of a

rule of an MCFG but is not necessarily in R. Then, the *rewriting* of a term is given by selecting a rule of the leftmost RHS non-terminal and substituting the variables by the terms of the corresponding rule, while adding the new RHS of the selected rule to the left of the RHS of the current term.

Symbolically, the rewriting is given by: let F_n be the first RHS non-terminal whose variable(s) appear(s) on the LHS:

$$\begin{aligned} &F(\alpha_1, \ldots, \alpha_{N(F)}) \leftarrow \ldots, F_{n-1}(\ldots), F_n(x_1^{(n)}, \ldots, x_{N(F_n)}^{(n)}), F_{n+1}(\ldots), \ldots \\ \overset{r}{\Rightarrow}\ &F(\alpha'_1, \ldots, \alpha'_{N(F)}) \leftarrow \ldots, F_{n-1}(\ldots), F'_1(\ldots), \ldots, F'_{m'}(\ldots), F_{n+1}(\ldots), \ldots \end{aligned}$$

Here, r is a rule in R for F_n. Let it be:

$$F_n(\alpha_1^{(n)}, \ldots, \alpha_{N(F_n)}^{(n)}) \overset{p}{\leftarrow} F'_1(\ldots), \ldots, F'_{m'}(\ldots)$$

Then each α'_i above is given by $\alpha_i[\alpha_1^{(n)}/x_1^{(n)}, \ldots, \alpha_{N(F_n)}^{(n)}/x_{N(F_n)}^{(n)}]$. The probability of this rewriting is simply the probability p attached in r. Additionally, the variables of the newly added RHS non-terminals must be renamed to maintain the uniqueness of the variables, so that the variables for the newly added F'_i do not conflict with those for the remaining F_i $(i \neq n)$.

Then, a term t is called *terminating* iff its LHS does not contain any variable on RHS.

We naturally extend the rewriting from a single rule to a sequence of rules. The probability of such a rewriting is then naturally the product of the probabilities of all involved rules.

Then, a term with an empty RHS is called a terminating term. Hence, the *termination probability* of the SMCFG is defined as the sum of all rewritings from the terms in R for S to all possible terminating terms. We denote this by $\mathbb{P}[\mathcal{G}] := \sum_{r=(S(\ldots)\leftarrow\ldots)\in R} \sum_{\{s \mid r \overset{s}{\Rightarrow} t, t \text{ is terminating}\}} \mathbb{P}[s]$, where s ranges over finite sequences of rules.

Termination of SMCFG

The termination problem of a given SMCFG is decidable and is essentially straightforward, just as it is for Stochastic Context-Free Grammars (SCFG) [86, 109, 140, 198].

Firstly, we should normalise the SMCFG into a non-deleting SMCFG with the same termination probability, which is analogous to the procedure for MCFG, and thus trivial. In fact, the attribute of being *non-deleting* is not a genuine restriction – one can perform straightforward normalisation to obtain an equivalent non-deleting MCFG from any MCFG [100].

Next, we observe that the visiting order of the derivation tree is not crucial, and what matters in probability computation is merely the rule choice. With the non-deleting property, it is guaranteed that every rule choice will have an impact on the LHS. Therefore, a non-deleting SMCFG rule:

$$F(\ldots) \overset{p}{\leftarrow} F_1(\ldots), \ldots, F_n(\ldots)$$

is equivalent to the following SCFG rule in terms of termination probability:

$$F \xrightarrow{p} F_1 \dots F_n$$

Thus, we transform the non-deleting SMCFG into an SCFG with the same termination probability, which then renders the termination problem decidable.

To formalise the intuition, we can state the following theorem:

Theorem 8.29. *The problem of whether a given SMCFG terminates almost surely is in PSPACE.*

The decidability is established through the following lemmas:

Lemma 8.30. *For a given SMCFG $\mathcal{G}$, one may construct in polynomial time a* non-deleting *SMCFG $\mathcal{G}'$ with the same termination probability.*

The proof of this lemma directly adopts the standard methods developed for MCFG [100, 101].

We then formally introduce the concept of a *simulation SCFG* A for a given SMCFG $\mathcal{G}$: there is a rule in A:

$$F \xrightarrow{p} F_1 \dots F_n$$

if and only if the rules for F with $F_1, \dots, F_n$ on the RHS in $\mathcal{G}$ are:

$$\begin{aligned} &F(\dots) \xleftarrow{p_1} F_1(\dots), \dots, F_n(\dots) \\ &\dots \\ &F(\dots) \xleftarrow{p_n} F_1(\dots), \dots, F_n(\dots) \end{aligned}$$

with $p = \sum_{i=1}^{n} p_i$.

Lemma 8.31. *For a given non-deleting SMCFG $\mathcal{G}$, its termination probability is the same as its simulation SCFG A.*

Proof. It suffices to prove instead the property that for any term of $\mathcal{G}$:

$$F(\alpha_1, \dots, \alpha_{N(F)}) \leftarrow F_1(\dots), \dots, F_m(\dots)$$

if the rule is *non-deleting*, its termination probability is the same as the term $F_1 \dots F_m$ of A.

This can be proved by induction on the length of the induction sequence. The base case, when the term is terminating, is trivial—with termination probability 1. In the inductive case, one may prove that since any rule of F_1 is non-deleting and the term itself is non-deleting, by induction for one more step, the sequence will remain non-deleting. Thus, the property follows directly from the induction hypothesis. □

Thus, for a non-deleting SMCFG, its termination probability can be converted to that of its simulation SCFG, which is of linear complexity. Then, the problem is solvable in PSPACE using established methods for SCFG [109].

This process can be straightforwardly applied to the *stochastic PMCFG* (SPMCFG), implying that the AST problem of SPMCFG is equivalent in hardness to that of SMCFG and SCFG, which is equivalent to pBPA and single-exit RMC [109]. Hence, the termination problem of rPTSA (and PAHORS) are much harder compared to that of SMCFG.

I learned very early the difference between knowing the name of something and knowing something. One's knowledge of the world begins with direct experiences. To experiment is to touch, to manipulate, to learn how the world works by engaging with it directly.

— Richard P. Feynman

9 PTSV

Contents

We implemented the algorithms for termination and expectation checks for both rPTSA and PAHORS. This implementation features the following contributions:

- It includes a unified and efficient equation system generation procedure that is shared between termination probability and expected termination time. The procedure significantly trims the space of patterns to explore, fundamentally

making PTSV feasible for evaluating empirical examples.

- Additionally, it features numerous polynomial optimisations. Combined with the equation system generation process, this enables PTSV to perform efficiently (usually within 1 second) in the experiments despite the EXPSPACE nature of the overall algorithm.

- We have comprehensively tested PTSV and confirmed its efficacy on a wide range of examples collected from the literature for both automata and recursion schemes. We have also devised several families of test cases to further demonstrate the strengths and limitations of PTSV.

- As an additional note, it is also the first implementation of PAHORS to rPTSA translation, building upon the algorithmic procedure proposed in [102].

Our tool is available at: `https://github.com/ssyram/PTSV`

The subsequent sections of this chapter begin with an overview of the PTSV's functionalities and workflow. Following this, we detail the usage of our tool, covering the installation steps and providing a manual for input format and output description. We then explore the implementation specifics, particularly focusing on the core procedure for efficiently generating the unified equation system, the key enabling the tool to perform practically. The document proceeds with a comprehensive set of experiments, which include case studies on automata and recursion schemes. Lastly, we discuss related tools and analyse their relationship with PTSV.

9.1 PTSV Overview

The Probabilistic Tree-Stack Verifier (PTSV) is a tool developed using the F# programming language and manipulates rational numbers using the Rationals .NET library. It efficiently addresses both qualitative and approximation problems, specifically calculating the termination probability and expected termination time for: rPTSA, PAHORS, pPDA, and pBPA models.

The PTSV workflow, illustrated in Figure 9.1, begins with accepting a file that describes a probabilistic model in one of the four supported formats. Depending on the model type, the workflow follows different paths:

For a PAHORS model input, PTSV initially converts it into an equivalent rPTSA model. The conversion process, detailed in Section 8.2.3, ensures that the values for the analysed properties, such as termination probability or expected termination time, are preserved correctly. The transformed model is then processed using the standard rPTSA workflow.

If the model is already in rPTSA format, it directly proceeds to the equation system generation phase without further modifications.

For pPDA or pBPA models, PTSV provides two approaches. One approach involves converting the pPDA to a 1-PTSA, as outlined in Section 8.1.2, and then analysing it using the standard rPTSA process. Alternatively, PTSV can generate

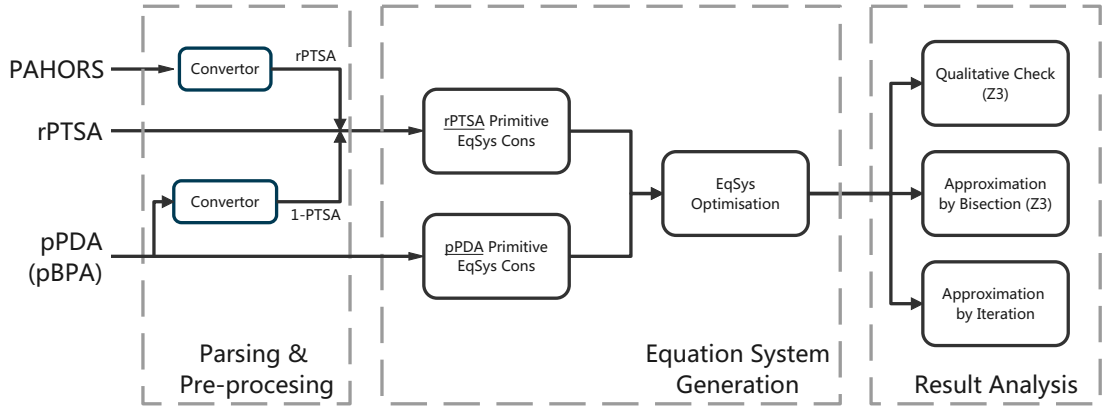

Figure 9.1: PTSV Overview

an equation system directly for the pPDA, following the established methodologies as referenced in the literature, such as [87, 110].

In the subsequent phase, a *primitive* equation system is constructed for either rPTSA or pPDA models. This system is then optimised to facilitate more efficient analysis.

Finally, PTSV offers three techniques for processing the optimised equation system and obtaining the final results. The first technique involves qualitative analysis for addressing AST and PAST problems using the Z3 SMT solver to decide the *existential theory of the reals (EToR)* inquiries through the "QF_NRA" solver [112]. The second technique uses bisection to approximate values, iteratively querying Z3 until the desired precision is achieved. The third technique employs iterative approximation through Kleene iteration to find the solution.

9.2 Usage Mannual

9.2.1 Running PTSV

Access the source code of PTSV and download it at `https://github.com/ssyram/PTSV`. Then, there are generally two ways to run PTSV.

Build from Docker The easiest way to run the project is running from Docker.

1. Install Docker (`https://docs.docker.com/get-docker/`) on local machine.
2. Change working directory to the project directory.
3. Execute the command to create a docker image:

   ```
   docker build . -t ptsv
   ```

4. Run the image built (with `docker run -it ptsv` or `docker run -dit ptsv`).
5. Run the following command to execute the tool:

```
./PTSV filesToTest
```

6. Or, run the `RunAll.sh` file to test all the examples to be introduced below, stored in the built directory `paper examples`:

```
./RunAll.sh
```

Build from `dotnet` One may also run by compiling the codes by `dotnet`.

1. Install dotnet-sdk (`https://dotnet.microsoft.com/en-us/download`) (recommended version is 6.0.403).
2. Change directory to the root of this project (with this `README.md`).
3. Run the following command to build the project:

```
dotnet build
```

4. Run the built `.dll` file. (usually in `./PTSV/bin/Debug/net6.0/`, so: `cd ./PTSV/bin/Debug/net6.0/` and then `dotnet PTSV.dll filesToTest`).
5. Or, run the whole project directly:

```
dotnet run --project PTSV filesToTest
```

Running PTSV Then, run PTSV by either `./PTSV files [options]` or `dotnet PTSV.dll files [options]`. Running options will be detailed below.

9.2.2 Input Format

A file to be run by PTSV consists of two parts:

1. (optional) some `let` value bindings defining variables to be used below for expressing probabilistic expressions;
2. a probabilistic model description, allowing formats of rPTSA, PAHORS, pPDA and pBPA. Each file should contain exactly one model description.

Additionally, line comments are allowed in a C-language-style – namely, a comment is prepended by `//`.

A `let` binding is of the form:

```
let <name> ':=' <expr>
```

An expression `<expr>` consists of rational number arithmetic computation. For example, define three probability values:

```
let pA := 1/4
let pB := 1/2
let pC := 1 - pA - pB
```

The model description varies as the type of the model changes. For each type of models, the general principle is a declaration of supportive information (types, initial state, etc.) followed by the model rules. We then introduce each of them one by one. Each part is a block surrounded by a specially marked environment of the following form:

```
%BEGIN <model> <block-desc>
...
%END <model> <block-desc>
```

PAHORS

The model description of a PAHORS model is of the following. Recall the definition of PAHORS in Definition 8.5. The model definition starts with a block declaring the PAHORS types, namely the definition of $\mathcal{N}$. Implicitly, we assume the first symbol to declare be the starting non-terminal S, which should have type o. The linear type operator $\multimap$ is writted as either `-o` or `->`, as the base type is written `o`, the latter one should be more explicit.

After the type declaration, it follows the definition of grammar rules, namely $\mathcal{R}$ of PAHORS.

```
%BEGIN PAHORS type
<name> : <type>[.]
%END PAHORS type

%BEGIN PAHORS grammar
<name> <args list> [‘(’ <prob expr> ‘)’] := <term>.
%END PAHORS grammar
```

Here, the `<prob expr>` is a probability expression that allows the arithmetic operations, constant numbers and variables declared above.

Additionally, we present some more notes on the format of `<term>`.

- Termination symbol: `\top`, of type `o`.
- Divergence symbol: `\Omega`, of type `o`, this is for explicit divergence.
- Projection symbol: `\pi(<number>)`, where `<number>` counts from `1`.
- Pairing (Grouping) symbol: `‘<’ <terms separated by ‘,’> ‘>’`.
- The `<term>` supports head-normal application terms of lambda-calculus, projections, and pairing (grouping).

Example 9.1. Let us consider another example translated from the “ListEven2” example in [84, 85]. The example is of order-2 and is depicted below.

```
let p1 := 1/2
let p2 := 1/2

%BEGIN PAHORS type
S : o
ListEven : (o & o -> o) -> o -> o
ListOdd : (o & o -> o) -> o -> o
Selector : o & o -> o
%END PAHORS type

%BEGIN PAHORS grammar
S := ListEven Selector \top.
ListEven x c (1/2):= c.
ListEven x c (1/2):=
  x <ListOdd Selector c, ListEven Selector c>.
ListOdd x c (1/2):=
  x <ListOdd Selector c, ListEven Selector c>.
Selector x (p1):= \pi(1) x.
Selector x (p2):= \pi(2) x.
%END PAHORS grammar
```

Example 9.2. Consider another PAHORS with non-terminals listed below:

$$S : o \qquad B : (o \multimap o) \multimap o \multimap o \qquad C : o \multimap o$$
$$A : ((o \multimap o) \multimap o \multimap o) \multimap o \qquad EndB : (o \multimap o) \multimap o \multimap o \qquad Id : o \multimap o$$
$$WrapB : ((o \multimap o) \multimap o \multimap o) \multimap (o \multimap o) \multimap o \multimap o$$

and the following reduction rules:

$$S \xrightarrow{p_A} A\ EndB \qquad B\ y\ z \xrightarrow{p_B} y\ (C\ z) \qquad EndB\ y\ z \xrightarrow{p_C} y\ z$$
$$A\ x \xrightarrow{p_A} A\ (WrapB\ x) \qquad WrapB\ x\ y\ z \to B\ (x\ y)\ z \qquad C\ z \xrightarrow{p_C} z$$
$$A\ x \xrightarrow{p_B} x\ Id\ \top \qquad EndB\ y\ z \xrightarrow{p_A} EndB\ y\ z \qquad Id\ z \to z$$

The scheme can be viewed as a random printer over $\{A, B, C\}^*$, where at each stage A, B, C can be produced with probabilities p_A, p_B, p_C respectively. It is designed to have the same termination probability as the probability of printing a string of the form $A^m B^m A^n C^m$ $(m > 0, n \geq 0)$. Indeed, note that terminating sequences are of the shape:

$$S \xrightarrow{p_A^m} A\ (WrapB^n\ EndB) \xrightarrow{p_B^m} EndB\ Id\ (C^m\ \top) \xrightarrow{p_A^n} EndB\ Id\ (C^m\ \top) \xrightarrow{p_C^m} \top$$

where $F^i\ x$ denotes the i-fold application of F to x.

This is an order-3 PAHORS and can be expressed in PTSV, as illustrated below.

```
let pA := 0.5
let pB := 0.25
let pC := 0.25
```

Figure 9.2: Example of PAHORS in PTSV

```
%BEGIN PAHORS type
S : o
A : ((o -> o) -> o -> o) -> o
B : (o -> o) -> o -> o
WrapB : ((o -> o) -> o -> o) -> (o -> o) -> o -> o
EndB : (o -> o) -> o -> o
C : o -> o
Id : o -> o
%END PAHORS type

%BEGIN PAHORS grammar
S (pA):= A EndB.  // \Omega is not shown
A x (pA):= A (WrapB x).
A x (pB):= x Id \top.
B y z (pB):= y (C z).
WrapB x y z := B (x y) z.
EndB y z (pA):= EndB y z.
EndB y z (pC):= y z.
C z (pC):= z.
Id z := z.
%END PAHORS grammar
```

Note that the definition itself does not conform to the probability 1 constraint that the probabilities across all rules for a non-terminal sum up to 1. For example, we only define a single rule for `S` with probability `pA`. This is for convenience, and the remainder of the probability is assumed to transit to the divergence `\Omega`.

rPTSA

It is a config followed by the rPTSA rules definition. The configuration is essentially the declaration of the restriction number k (`restriction` below), the initial control state q_{init} (`q0` below), the initial local memory m_{init} (`m0` below). Additionally, symbol for stack bottom $\perp$ is also customisable and it is declared by assigning `gamma0` below. After that, we define the rules for the rPTSA.

Note that it is the responsibility of the user to put in a correct restriction number. The equation system generation procedure explores only the patterns conforming to the restriction number.

```
%BEGIN rPTSA config
restriction := <number>
q0 := <q>
m0 := <m>
gamma0 := <gamma>
%END rPTSA config
```

```
%BEGIN rPTSA rules
<rules>
%END rPTSA rules
```

Here, an rPTSA rule is of the following format:

```
(<q>, <m>, <g>, <prob expr>, <op>)[.]
```

That is to say, we write the rule $(q, m, g) \xrightarrow{p} op$ as a 5-tuple (q, m, g, p, op).

Here, the operations `<op>` is given by:

```
<q>
(<q>, <m>, down)
(<q>, <m>, up <gamma>)
\top
\Omega
```

The first one is a horizontal transition, followed by the *down* and *up* operations. Additionally, we also employ the explicit instant termination operation `\top` for simplicity, besides the implicit termination given by a *down* rule from $\perp$ (`gamma0`); and also the explicit divergence operation `\Omega`.

Example 9.3. Consider the example of a random printer Example 9.2 that randomly prints A, B and C and asks the probability of printing a string in the form of $A^nB^nA^mC^n$. Now we define it in the rPTSA scenario, whose definition in PTSV is depicted below.

```
let pA := 0.7
let pB := 0.2
let pC := 1 - pA - pB

%BEGIN rPTSA config
restriction := 2
q0 := q1
m0 := m1
gamma0 := g0
%END rPTSA config

%BEGIN rPTSA rules
(q1, m1, g0, pA, (q1, m1, up g1))
(q1, m1, g1, pA, (q1, m1, up g1))
(q1, m1, g1, pB, (q2, m3, down))
(q2, m1, g1, pB, (q2, m2, down))
(q2, m1, g0, pA, q2)
(q2, m1, g0, pC, (q1, m1, up g1))
(q1, m2, g1, pC, (q1, m1, up g1))
(q1, m3, g1, 1, \top)
%END rPTSA rules
```

Fundamentally, it commences by generating A repeatedly with probability `pA` for a specified number of iterations, denoted as n, resulting in a stack height of $n+1$ while maintaining the control state `q1` and keeping the local memory at `m1`.

Alternatively, at each iteration, apart from continuing the generation of A, it might decide to initiate the generation of B. Initially, this causes the local memory to change to `m3`, signifying the end of generation. The process of generating B also continues for n steps, descending back to the base, during which the local memory transitions to `m2` on the second visit to generate C.

Upon reaching the bottom, while still in control state `q2`, it perpetuates looping at the base with a horizontal transition, potentially generating an arbitrary number of As.

Moreover, it might begin generating C by ascending with `q1` once again. In this context, during the *up* visits, instead of `m1` for the initial visit, it now holds `m2` and `q1`, distinguishing it from the initial generation. The generation of C concludes upon reaching a local memory of `m3`.

Additionally, note that, akin to the PAHORS case, the rules for some states might be incomplete (the probabilities across rules for them may not sum to 1) or even absent. For these missing rules, one may presume a transition to `\Omega`.

pPDA

The model of pPDA also consists of config and rules. The configuration simply requires the definition of `q0` and `gamma0`, abbreivating `m0` and `restriction` from that of rPTSA.

```
%BEGIN pPDA config
q0 := <q>
gamma0 := <gamma>
%END pPDA config

%BEGIN pPDA rules
<rules>
%END pPDA rules
```

There are two kinds of possible styles of rules, the first is the explicit style, which mimics the rPTSA rules, as defined in Definition 8.2; the second is the rewriting rule style, as Equation (8.1).

Explicit style In the explicit style, a pPDA rule follows this format:

```
(<q>, <gamma>, <prob expr>, <op>)[.]
```

This rule can be interpreted as: $(q, \gamma) \xrightarrow{p} op$ where q represents the current state, γ represents the stack symbol, p is the probability expression, and op represents the operation.

The possible operations `<op>` are:

```
<q>
pop
push <gamma>
```

Here, `<q>` signifies a horizontal transition to state q, `pop` signifies a stack pop operation, and `push <gamma>` signifies a stack push operation with symbol γ.

Rewriting rule style In the rewriting rule style, a pPDA rule is expressed as:

```
<q> <gamma> [‘(’ <prob expr> ‘)’] ‘->’ <q> <gamma>* ‘.’
```

This notation represents the transition rule where `<q>` is the current state, `<gamma>` is the stack symbol (could also be $\perp$), `<prob expr>` is the probability expression, and the right-hand side indicates the resulting state and stack symbols after the transition.

pBPA

As a more restricted version of pPDA, PTSV also accepts the definition of *Probabilistic Basic-Process Algebra* (pBPA) [110, 114], which is formally a single-control-state pPDA. Taking advantage of the characteristics, a rule of it would be like:

$$X \xrightarrow{p} X_1 \; X_2 \; \ldots \; X_n$$

In PTSV, the pBPA model exclusively supports rewriting rules. The configuration only necessitates defining the symbol for the stack bottom $\perp$ (`gamma0` below).

```
%BEGIN pBPA config
gamma0 := <ident>.
%END pBPA config

%BEGIN pBPA rules
<rules>
%END pBPA rules
```

A rewriting rule for pBPA follows this format:

```
<gamma> [‘(’ <prob expr> ‘)’] -> <gamma>*.
```

In this format, `<gamma>` ranges over the stack symbols, `<prob expr>` is the numeric expression indicating the probability, and the right-hand side specifies the sequence of stack symbols resulting from the application of the rule.

9.2.3 Output Analysis

A typical output example file is displayed in Figure 9.3[1]. The lines provide detailed information for various stages of the computation, besides the pure results for the termination problems. Here's a detailed explanation:

[1]We ignored the printing in-progress, which could be reached by the silent mode enabled by `-s` for simplicity.

```
======================== Running file: "example_file" ========================
read format: 8-PTSA
parse time: 0.1204s
termination probability equation system primitive scale: 17
termination probability equation system primitive construction time: 0.1282s
termination probability equation system simplified scale: 8
termination probability equation system simplification time: 0.0778s
termination probability equation system total construction time: 0.2080s
termination probability approximation value (iteration): 0.9986
termination probability approximation (iteration) times: 1407
termination probability approximation time (iteration): 0.4871s
termination probability is Almost Sure Termination (AST): True
termination probability Almost Sure Termination (AST) time: 0.5267s
expected termination time equation system primitive scale: 34
expected termination time equation system primitive construction time: 0.1577s
expected termination time equation system simplification time: 0.0650s
expected termination time equation system simplified scale: 17
expected termination time equation system total construction time: 0.2254s
expected termination time has value: False
expected termination time is Positive Almost Sure Termination (PAST): False
expected termination time qualitative time: 0.0048s
expected termination time approximation value (iteration): No Finite Value
conditional expected termination time approximation value (iteration): No Finite Value
expected termination time approximation time (iteration): 0.0000s
expected termination time approximation (iteration) times: 0
total time: 1.6421s
========================== End file: "example_file" ==========================
```

Figure 9.3: PTSV Output Example

- **read format**: This specifies the format of the probabilistic model, denoted as 8-PTSA here, indicating a PTSA subjected to an 8-restriction.
- **parse time**: This denotes the time required to parse the input file, which in this instance is 0.1204 seconds.
- **termination probability equation system primitive scale**: Indicates the scale (number of explored patterns) of the primitive equation system constructed for calculating the termination probability, set to 17 here.

  Recall from Figure 9.1, the construction is achieved through an optimised equation system generation process, a key contribution of our implementation, detailed below. Despite the potential exponential growth of the scale equation system concerning k (8 in this case), the construction only explores 17 patterns, significantly reducing computation time. After construction, the system undergoes simplification using various polynomial simplification techniques for further analysis. Thus, we refer to the equation system directly resulting from

the construction procedure, before undergoing simplification, as the *primitive* equation system.

- **termination probability equation system primitive construction time**: The time taken to construct the primitive system (of scale 17 here), which is 0.1282 seconds.
- **termination probability equation system simplified scale**: The scale of the equation system after simplification of the primitive system, reduced to a size of 8, approximately half of the original. Simplification can occasionally reduce the scale of the equation systems by up to 100 times or more.
- **termination probability equation system simplification time**: The time required to simplify the primitive system, which is 0.0778 seconds.
- **termination probability equation system total construction time**: The cumulative time for constructing the termination probability equation system, encompassing both the construction of the primitive system and its simplification, totalling 0.2080 seconds.

Next, the output provides information on the termination probability approximation and whether it is almost surely terminating:

- **termination probability approximation value (iteration)**: The approximated value of the termination probability, given as 0.9986. This value is derived through an iterative approximation process. Alternatively, the approximation can be achieved using the bisection method with the Z3 SMT solver, in which case the reporting will use "bisection" instead of "iteration". The way of configuration will be discussed below.
- **termination probability approximation (iteration) times**: The number of iterations taken, which is 1407. This represents the iterative steps needed to refine the probability estimate. If using the bisection method, there will be no this field, as the times required for bisection to work given a specific accuracy is fixed.
- **termination probability approximation time (iteration)**: The time taken for the approximation process, which is 0.4871 seconds. This duration reflects the total time consumed by all iterations. When employing the bisection method, this time will account for the bisection process duration using the Z3 SMT solver.
- **termination probability is Almost Sure Termination (AST)**: Indicates whether the termination is almost sure, which is **True** in this example. This determination is made using the Z3 SMT solver to perform the AST inquiry.
- **termination probability Almost Sure Termination (AST) time**: The time taken to decide AST, which is 0.5267 seconds in this instance. This duration reflects the process of querying AST using the Z3 SMT solver.

Following this, the output details the expected termination time equation system:

- **expected termination time equation system primitive scale**: Likewise, there is a report concerning the scale of the primitive equation system constructed for the expected termination time, which is 34 in this example.

  Recall that in the equation system for expected termination time, it involves also the equation system for the termination probability. To speed up the process, if the equation system for termination probability has been constructed, we will directly reuse the system and the process of exploring the patterns (to be detailed below when introducing the equation system construction procedure).

- **expected termination time equation system primitive construction time**: The time taken to construct this primitive system, which is 0.1577 seconds. Thanks to the reuse procedure by utilising the data from the construction for termination probability, the construction for expected termination time is still efficient.

- **expected termination time equation system simplification time**: The time taken to simplify this system, which is 0.0650 seconds in this example. The simplification procedure is the same as for the termination probabilities.

- **expected termination time equation system simplified scale**: The scale after simplification, reduced to 17, still cutting half in this example.

- **expected termination time equation system total construction time**: The total construction time for the expected termination time equation system, which is 0.2254 seconds. Like in the case of termination probability, the time includes the construction and simplification of the equation system.

The subsequent part concerns the results of the expected termination time.

- **expected termination time has value**: Recall that the equation system for the expected termination time might not yield a finite value, implying the result could be ∞. Consequently, prior to any further analysis, one must determine if a (finite) solution exists for this system. This issue is encoded as an SMT (EToR) query to be processed by Z3.

  Note that this property is sensitive; even minor value discrepancies, caused by, e.g., floating point representation, could lead to an incorrect conclusion. Thanks to the exact numeric representation using rational numbers in PTSV, we are safeguarded against the risk of potentially producing an erroneous answer. In this instance, the constructed equation system does not have a (finite) value.

- **expected termination time is Positive Almost Sure Termination (PAST)**: This indicates whether the expected termination time corresponds

to positive almost sure termination, which is **False**. This result combines the determination of whether there is a finite value for the equation system and whether the probabilistic model is AST.

- **expected termination time qualitative time**: The duration required to ascertain the qualitative properties (solved by Z3), which is 0.0048 seconds here.
- **expected termination time approximation value (iteration)**: The expected termination time $ett(-)$ value derived from the iteration process, indicated as **No Finite Value** in this example, signifying the expected termination time is infinite. In PTSV, there are two approaches to handle the generated expected termination time: iteration and bisection, like in the termination probability case above.
- **conditional expected termination time approximation value (iteration)**: The approximated value for conditional expected termination time, which is the expected termination time divided by the termination probability, calculated as **No Finite Value** since the expected termination time is infinite.
- **expected termination time approximation time (iteration)**: The time taken for the approximation process, recorded as 0.0000 seconds in this example – since the "no finite value" result has been obtained, no iteration occurs.
- **expected termination time approximation (iteration) times**: The number of iterations executed, which is 0 – since there is no finite value, no iteration is performed.

Finally, the **total time** for the entire process is reported as 1.6421 seconds, which is efficient enough for daily analysis. Indeed, this is also the typical (or slightly longer) running time across examples gathered from the literature, as can be seen below in the experimental part.

9.2.4 CLI Options

By understanding the entirety of PTSV's input and output, we are now prepared to discuss additional functionalities provided by the tool through the following options.

PTSV offers two primary modes of configuration:

- `-tp`: Activates termination probability analysis mode.
- `-ett`: Activates expected termination time analysis mode.

Once a mode is activated, the user can use the following commands to refine the user's analysis:

- `-approx`: Initiates value approximation.

- `-qual`: Computes qualitative results (AST/PAST).
- `-iter`: Uses iterative methods to approximate values.
- `-bisec`: Uses bisection methods (via Z3) to approximate values.

Note that once a mode is selected, all execution options for this mode are cleared and will wait for the commands to enable functionalities. For example, if the user wants to analyse termination probability and approximate its value iteratively but no need for AST analysis and no expected termination time analysis, it would involve:

```
./PTSV files -tp -approx -iter -ett
```

By default, if no options are specified, PTSV execution is equivalent to the following command:

```
./PTSV files -tp -qual -approx -iter -ett -qual -approx -iter
```

In addition to the primary modes and their specific commands, PTSV provides several options that can be used independently of the mode. These options offer additional control and customization for various aspects of the tool's functionality:

- `-d` or `-debug`: Print debugging information.
- `-min_diff <number>`: Specify the iteration minimum difference as the iteration convergence criterion (default: `1e-6`).
- `-accuracy <number>`: Specify bisection accuracy (default: `1e-6`).
- `-max_iter_round <number>`: Specify the maximum times of iteration (default: unlimited).
- `-t <number>`: Set timeout in milliseconds.
- `[-D] <name>=<number>`: Specify bindings as additional (prepending) `let` expressions.
- `-conv`: Convert pPDA/pBPA to rPTSA before solving, by default the equation system generation procedure for pPDA and pBPA is directly that for pPDA as in the literature [87, 110].
- `-s`: Enable silent mode, so only the result is printed.
- `-a`: Test all information, equivalent to:

  ```
  -tp -qual -approx -iter -bisec -ett -qual -approx -iter -bisec
  ```

- `-stack <number of bytes>`: Sets the stack size to prevent stack overflow issues. While stack overflow problems rarely occur in PTSV, they can arise in unusual cases. When this happens, the program will report an error and terminate execution without producing a result. Use this option to address such edge cases.

- `-report_stuck`: Report reachable status that may get stuck (no rule defined).

To further illustrate typical usage scenarios, consider the following examples:

Example 9.4. To compute both termination probability and expected termination time using iteration in silent mode:

```
./PTSV files -tp -approx -iter -ett -approx -iter -s
```

Example 9.5. To compute termination probability using bisection with a specific accuracy:

```
./PTSV files -tp -approx -bisec -accuracy 1e-5
```

9.3 Implementation Details

In this section, we delve deeper into the implementation details of PTSV, providing an overview of the broad optimisations and detailing the key ones that enable PTSV to efficiently handle a wide range of examples, usually within a second.

9.3.1 Implementation Overview

The primary challenge in implementing our method to deliver timely results is generating the system of equations for rPTSA as quickly as possible while simplifying the generated system as much as possible.

Note that the equation systems (e.g., Equation (5.9)) are of EXPSPACE complexity. Therefore, optimisations that minimise the traversal of patterns and summands on the RHS are crucial for developing a practical software tool. Furthermore, the scale of the equation system significantly impacts both the iterative approximation and the quantitative enquiries passed to the Z3 SMT solver for EToR verification.

To address this, we implemented a two-phase optimisation strategy in PTSV to efficiently generate minimal-scale equation systems for further approximation or qualitative analysis.

The first set of optimisations in PTSV occurs during the exploration of patterns, specifically in constructing the primitive equation system. The main goal in this phase is to detect only the patterns relevant to the solution and to eliminate spurious interactions.

More specifically, we focus on patterns that contribute to the bottom patterns $Trips^{\perp}_{[q_{init},q]}$ for some q, as these are sufficient to analyse the target property (*tp* or *ett*). A key aspect of this optimisation is the on-the-fly detection of spurious interactions, or those patterns $Trips^{\gamma}_{D} = \emptyset$, to avoid unnecessary exploration of redundant patterns. This optimisation of exploring *relevant* patterns, which are inherently exponential in scale, is essential for efficiently generating the equation system and will be discussed in detail below.

The second set of optimisations aims to simplify the generated equations and reduce their scales.

- This includes constant detection and propagation (through bounded iteration), and variable elimination in straightforward cases (e.g., when one variable is a linear factor of another or when dealing with a simple linear equation in one variable).

- We also implemented routines to simplify arithmetic expressions on the RHS of the equations, bringing them to a canonical form. When two variables with the same RHS are encountered, they are identified.

- Similarly, when we find RHSs that are identical functions of the associated variable, we eliminate one of them.

Overall, as confirmed by experimental data, these optimisations significantly reduce the number of equations, and in some cases, they alone can produce the final result. This typically occurs when the equations assume a simple linear form following earlier simplifications.

Depending on the verification task, the generated system of equations will be used either as part of a Z3 query or evaluated to support iterative tasks. Hence, these optimisations, as will be confirmed below, significantly contribute to the overall effectiveness of the tool across the collected examples.

9.3.2 Horizontal Elimination

As may have been noticed, in the input format, for the convenience of definition, we allow the *horizontal transition* that purely changes the control state without moving *up* or *down*, the semantics of which are described in Remark 3.5 above. Yet, to build the equation system, we should first convert back to our current definition for rPTSA by eliminating the horizontal transitions.

The construction that eliminates horizontal transitions replaces each of them with an up_∇ ($\nabla \notin \Gamma$) and records information about the current state, memory, and current stack symbol using new states of the form $\ulcorner(q, m, g)\urcorner$. This continues until a non-horizontal transition, *op* (say), is to be fired, in which case the automaton will return to the original node via a cascade of *down* operations that remember – in new state $\ulcorner op \urcorner$ – what non-horizontal transition should be executed, once we reach the original node. Due to the restriction condition, the number of up_∇ operations executed in a single node will also be restricted.

More formally, we add ∇ to Γ and modify the rules of $\mathcal{A}$ as follows.

- Any transition of the form $(q, m, g) \xrightarrow{p} q'$ is *replaced* by the following two rules:

$$(q, m, g) \xrightarrow{p} (\ulcorner(q', m, g)\urcorner, m, up_\nabla)$$
$$(\ulcorner(q, m, g)\urcorner, m_{init}, \nabla) \xrightarrow{p} (\ulcorner(q', m, g)\urcorner, m, up_\nabla)$$

- For any transition $(q, m, g) \xrightarrow{p} op$, where *op* is non-horizontal, *add* the

following three rules:

$$\begin{aligned}(\ulcorner(q, m, g)\urcorner, m_{init}, \nabla) &\xrightarrow{p} (\ulcorner op\urcorner, m_{init}, down)\\ (\ulcorner op\urcorner, m_{init}, \nabla) &\to (\ulcorner op\urcorner, m_{init}, down)\\ (\ulcorner op\urcorner, m, g) &\to op\end{aligned}$$

Obviously, the modified rPTSA will have exactly the same termination probability as the original one. Besides, one may simply assign value 0 in the reward function for the latter two rules in the second point above to accurately maintain the same value as in the computation of expected total accumulated reward.

From a more theoretical aspect, although not relevant to PTSV, as for coping with model checking, one may utilise the standard techniques to tackle the additional dummy configurations (with $\ulcorner op\urcorner$ as the current state). For example, in DRA, one interprets the dummy configurations to a new letter $\bullet$, which will never change the DRA state when read. While in LTL / PCTL(*), one also interprets the dummy configurations to a distinct atomic formula, say, $\bullet$, and then replaces all occurrences of atomic sub-formulas, say p, in the given formula by $\bullet \mathcal{U} \mathsf{p}$.

9.3.3 The Equation System Generation Procedure

The generation of equation systems is fundamental to PTSV. Given that the algorithm runs in exponential space, efficient generation is crucial for effective analysis. This section presents a unified and efficient framework for equation generation that supports both termination probability and expected termination time with minimal adaptation. This approach significantly reduces the space of patterns to explore, allowing PTSV to handle a wide range of examples, which will be introduced subsequently.

Theoretical Foundation

The key concept for generating the equation system in a unified manner is to investigate the process of constructing the *pre-paths patterns*, as illustrated in Equation (4.21). This approach provides the flexibility to manage both termination probability and expected termination time by applying the $wt(-)$ and/or $mht(-)$ functions to the decomposition results in an *on-the-fly* manner.

Furthermore, based on the decomposition equation, the most intuitive implementation would involve adopting a propose-and-test loop for the upward interface assigning functions $\mathbb{D}$. However, from an efficiency standpoint, it is evident that despite the potentially vast number of $\mathbb{D}$, only a small subset of these actually contribute to the final result – those where the corresponding $SynPaths^{\mathbb{D}}_{(g,D)}$ is not empty for a given g and D. Thus, we instead explore these effective $\mathbb{D}$ also in an *on-the-fly* manner by need. The key insight is to investigate through the synchronisation paths. This entails converting the original decomposition equation for pre-path patterns into the following form, which serves as the theoretical foundation for our

equation system construction procedure. Formally, for a pattern $v := \mathit{TripsOrUp}_D^{\mathscr{q}}$ [2], let $\mathit{SynPaths}_{(\mathscr{q},D)}$ be defined as the union of *all* $\mathit{SynPaths}^{\mathbb{D}}_{(\mathscr{q},D)}$ across all possible $\mathbb{D} \sim : v$, then we have the following:

$$\mathsf{path}(v) = \biguplus_{\pi \in \mathit{SynPaths}_{(\mathscr{q},D)}} \{\mathsf{Sync}(\pi, \{\gamma \mapsto \vec{\phi}_\gamma \mid \gamma \in \Gamma\}) \mid \forall \gamma \in \Gamma . \vec{\phi}_\gamma \in \mathsf{path}(\mathit{TripsOrUp}^{\gamma}_{\pi\restriction_\gamma})\} \tag{9.6}$$

The Framework

Next, we present the actual algorithm. For simplicity and clarity, we first introduce a general framework that outlines the core functionalities, followed by an explanation of the key optimisations applied to this framework to further expedite the construction process. Since only the *Trips* patterns are relevant to termination analysis, we focus on constructing the right-hand side for a fixed pattern $\mathit{Trips}^{\gamma}_D$. We refer to this construction procedure as $\mathsf{build}(\mathit{Trips}^{\gamma}_D)$.

We commence by defining the notion of *step information* $\iota = (\mathscr{q}, \mathscr{m}, \mathcal{D}, \eta)$, where:

- $\mathscr{q} \in \mathcal{Q}$ represents the current control state;
- $\mathscr{m} \in \mathcal{M}$ signifies the current local memory;
- $\mathcal{D} : \{\Downarrow\} \uplus \Gamma \to \mathcal{Q}^*$ serves as the exploring map, maintaining both the upward and downward interaction records; and,
- η embodies the *accumulated information*, the structure of which is contingent upon the equation system's intended purpose. Notably, for termination probability, η equals p, the cumulative weight of the synchronisation path; while for expected termination time, η is a pair (p, ℓ), recording both the cumulative weights p and the count of transitions ℓ (i.e., the accumulated rewards from the perspective of the expected total accumulated rewards).

Subsequently, given a pattern $v = \mathit{Trips}^{\mathscr{q}}_{\mathscr{q}::D}$, $\mathsf{build}(v)$ is succinctly defined as $\mathsf{traverse}_{\mathscr{g}}(\iota^{v}_{\mathit{init}})$, where $\iota^{v}_{\mathit{init}}$ is:

$$(\mathscr{q}, \mathscr{m}_{\mathit{init}}, \{\Downarrow \mapsto D\} \uplus \left\{\gamma \mapsto \varepsilon \,\middle|\, \gamma \in \Gamma\right\}, \eta_{\mathit{init}})$$

and the initial value of η_{init} varies based on the requirement – for termination probability, $\eta_{\mathit{init}} = 1$; for expected termination time, $\eta_{\mathit{init}} = (1, 0)$.

The function $\mathsf{traverse}_{\mathscr{g}}(\iota)$ then explores potential synchronisation paths in $\mathscr{g}$ as described by ι. Formally, $\mathsf{traverse}_{\mathscr{g}}(\iota)$ for $\iota = (\mathscr{q}, \mathscr{m}, \mathcal{D}, \boxed{\eta})$ navigates and generates corresponding components for all operations op in $\Delta(\mathscr{q}, \mathscr{m}, \mathscr{g})$ with a probability $p > 0$. To illustrate in an ML-like pseudocode:

```
1 let mutable rhs = 0 in
2 for δ ∈ Δ(q, m, g) do
3   let ι' = ι { η = updateAcc(η, δ) } in
```

[2] Consider also the decomposition of bottom patterns, as Equation (5.8).

```
4    rhs <- rhs + genFrom(ι′, op^(δ))
5  done;
6  return rhs
```

In line 1, an empty RHS is initialised to accumulate the RHS terms from each rule's exploration. Line 2 extracts each rule δ for status $(\mathscr{q}, \mathscr{m}, \mathscr{g})$ from Δ. Line 3 updates the step information ι' by modifying the accumulated information η, using F#-style syntax. The update by the original accumulated information η and the rule δ, that $\mathsf{updateAcc}(\eta, \delta)$, is also purpose-dependent – for termination probability, it is $\eta \cdot p$; for expected termination time, it results in a new pair $(p \cdot p', \ell')$, where $(p', \ell) := \eta$ and ℓ' adds to ℓ with the reward for executing δ. Finally, line 4 appends the RHS items obtained from executing op with ι' – the $\mathsf{genFrom}(\iota', op)$ function is determined by a case analysis of op.

- When $op = (\mathscr{q}', \mathscr{m}', \mathit{down})$ is a *down* operation, we must check the value of $\mathcal{D}(\Downarrow)$ to determine if the current path fulfils the requirement.

  - If $\mathcal{D} = [\mathscr{q}']$ comprises solely the control state $\mathscr{q}'$ in op, it indicates that the synchronisation path has been *completely* explored. Therefore, the current accumulated information η' of ι', combined with the upward interface in $\mathcal{D}$ for every $\gamma \in \Gamma$, can *potentially* correspond to non-trivial (non-0) term(s) for the pattern v. The specific process varies according to the intended purpose.

    In the context of termination probability, the term $\eta' \cdot \prod_{\gamma \in \Gamma} wt(\mathit{Trips}^{\gamma}_{\mathcal{D}(\gamma)})$ is returned. It is noteworthy that η' essentially represents $hp(\pi)$ for the current explored path π. On the other hand, for computing the expected termination time, the process yields a more intricate term:

    $$p \cdot \ell \cdot \prod_{\gamma \in \Gamma} wt(\mathit{Trips}^{\gamma}_{\mathcal{D}(\gamma)}) + \sum_{\gamma \in \Gamma} p \cdot ett(\mathit{Trips}^{\gamma}_{\mathcal{D}(\gamma)}) \cdot \prod_{\gamma' \in \Gamma \setminus \{\gamma\}} wt(\mathit{Trips}^{\gamma'}_{\mathcal{D}(\gamma')})$$

    where $(p, \ell) := \eta'$. Notably, $p \cdot \ell$ corresponds to the mean hitting time of the currently explored synchronisation path.

    Notably, the term "*potentially*" is used above as it is contingent on the non-triviality of the series of patterns $\mathit{Trips}^{\gamma}_{\mathcal{D}(\gamma)}$ for each $\gamma \in \Gamma$ with $\mathcal{D}(\gamma) \neq \varepsilon$.

  - If $\mathcal{D} = \mathscr{q}' :: \mathscr{q}'' :: D'$ for some D', this implies there are additional elements to explore, and the synchronisation path is not yet complete. Consequently, it returns the result of $\mathsf{traverse}(\iota'')$, where ι'' is $(\mathscr{q}'', \mathscr{m}', \mathcal{D}[\Downarrow \mapsto D'], \eta')$ and η' is the accumulated information of ι'.

  - In other cases, it implies that the current path does not align with the interaction interface. Therefore, an empty sequence 0 is returned as the result, ensuring that this synchronisation path does not contribute to the RHS of v.

- When $op = (\mathscr{q}', m', up_\gamma)$ represents an *up* operation, it becomes necessary to update $\mathcal{D}$ and continue the exploration, as the synchronisation path is still incomplete.

  In this case, we should first see if the upward exploration may continue by confirming that ($|\mathcal{D}(\gamma)| \leq 2k-2$) which conforms to the k restriction.

  Then, the challenge lies in determining how to further explore the synchronisation paths – more specifically, what are the potential states returning to this node from the upward direction γ? To address this, we need to maintain a list of feasible states descending from γ, with the goal of making this list as concise as possible for efficiency.

  By analysing the models, we developed additional techniques to streamline this list, with a notable approach involving exploiting the nature of linear game semantics to maintain a precise relationship when the model is transformed from PAHORS.

  Therefore, given the predicted descending states $\mathscr{q}''$ from γ, the process yields the *aggregate* of all the results of $\mathsf{traverse}(\iota'')$, where ι'' is: $\Big(\mathscr{q}'', m', \mathcal{D}\big[\gamma \mapsto \mathcal{D}(\gamma) +\!\!+ [\mathscr{q}', \mathscr{q}'']\big], \eta'\Big)$ and η' represents the accumulated information of ι'.

Key Optimisations

We have implemented further optimisations to the above synchronisation-paths-guided pattern exploration framework. These enhancements incorporate a DFS-based traversal strategy, featuring two key optimisations to expedite the process significantly: (1) treating found structurally void patterns as exceptions to avoid redundant exploration; and, (2) implementing void *prefix* buffering.

We start by defining the notion of *structurally void* patterns $Trips^{\mathscr{q}}_D$ as those for which no pre-run is described by the pattern $Trips^{\mathscr{q}}_D$. However, we are not able to find all of them in the DFS procedure due to circular dependency – that the RHS of a pattern depends on itself either directly or indirectly through other patterns.

Consequently, the function build not only constructs the equation system for the given pattern but also contructs for all patterns potentially contributing to it. Formally, during these processes, we additionally maintain three structures, namely, (1) a *partial* function r, representing the *resulting equation system*, which stores the constructed equation system and is initially set to $\emptyset$; (2) a function $\boxdot : \{\bot\} \uplus \Gamma \to 2^{\mathfrak{Q}^*}$, which stores the void prefixes of each direction such that, intuitively, for all $D \in \boxdot(\mathscr{q})$, the pattern $Trips^{\mathscr{q}}_D$ is structurally void, which is initialised to $\Big\{\mathscr{q} \mapsto \emptyset \,\Big|\, \mathscr{q} \in \{\bot\} \uplus \Gamma\Big\}$; and, (3) a DFS auxiliary structure P that records the status of the patterns under exploration, which is formally a *partial function* from patterns to a Boolean value indicating whether the pattern is structurally void. These structures are dynamically updated throughout the process to maintain accurate information.

For a pattern $v = Trips^{\mathscr{q}}_D$, the function $\mathsf{build}(v)$ then returns a Boolean value indicating whether the constructed value is *found to be* structurally void (may

not necessarily discover all due to circular dependency), and is illustrated in the following pseudocode:

```
1  if v ∈ dom(P) then return false
2  else if ∃D' ∈ ⊡(𝒒).D' ⊑ D then return true
3  else if v ∈ dom(r) then return false
4  else P <- P[v ↦ true];
5       let rhs = traverse(ι^v_init) in
6       r <- r ⊎ {v ↦ rhs};
7       let isVoid = P(v) in
8       (if isVoid then ⊡ <- ⊡[𝒒 ↦ ⊡(𝒒) ⊎ D]);
9       P <- P \ {v};
10      return isVoid  (* returns the value *)
```

In the code, lines 1, 2, and 3 initially check if the pattern is under exploration, has been identified as void, or has already been recorded in the result. Particularly, if the pattern is under construction, it suggests a circular dependency and whether it is structurally void could not be judged during the construction phase, and will be tackled by the simplification after the primitive system is constructed. For conservative purpose, we here presume it as not structurally void. From line 4 onwards, the program initiates exploration of the pattern. It first labels the pattern as structurally void in the path set in line 4, the status of which subjects to change once a potential synchronisation path is found, as illustrated below. Subsequently, in line 5, it performs the construction via traverse, records the result in line 6, and updates the void prefix map, including removing the pattern from the DFS auxiliary structure, starting from line 7.

It is important to note that a `rhs` value of 0 does not necessarily indicate structural voidness. For example, in the context of expected termination time, if the transitions in the entire transition system are all rewarded 0, the pattern's RHS value is 0 but may not be structurally void.

The traverse code remains unchanged, and modifications occur in the step $\mathsf{genFrom}(\iota', op)$. We now describe these changes.

When op is $(\mathscr{q}', -, down)$ and $\mathcal{D}$ in ι' contains a singleton $\{\mathscr{q}'\}$ in $\mathcal{D}(\Downarrow)$, before constructing and returning the value, we first verify if for every γ with $\mathcal{D}(\gamma) \neq \varepsilon$, $\mathsf{build}(\mathit{Trips}^{\gamma}_{\mathcal{D}(\gamma)})$ is uniformly false. Upon encountering a pattern $v = \mathit{Trips}^{\gamma}_{\mathcal{D}(\gamma)}$ with $\mathsf{build}(v) = \mathsf{true}$, we *throw* an exception containing the information v – which directs us back to the required interaction point. Additionally, when the check is passed, and a final value is generated, we shall also update P accordingly: $P \leftarrow P[v \mapsto \mathsf{false}]$, for the currently exploring pattern v.

Moreover, when op is $(\mathscr{q}', -, up_\gamma)$, for each predicted descending state $\mathscr{q}''$, we initially determine if $\mathcal{D}(\gamma) +\!\!+ [\mathscr{q}', \mathscr{q}'']$ is within $\boxdot(\gamma)$, indicating the structural voidness of the interaction, in which case, we cease further exploration from $\mathscr{q}''$ and use 0 as the result. Alternatively, when further exploration is warranted, we encapsulate $\mathsf{traverse}(\iota'')$ within a try-and-catch block that intercepts exceptions when the message is precisely the pattern $v = \mathit{Trips}^{\gamma}_{\mathcal{D}(\gamma) +\!\!+ [\mathscr{q}', \mathscr{q}'']}$ – that is, the pattern v is identified as structurally void. In this case, we terminate the exploration for additional paths from this point.

Final result Following the discussion above, to verify the value of the termination probability or the expected termination time, we merely need to construct the equation system for $Trips^{\perp}_{[q_{init},q]}$ across all q and their potentially contributing patterns. This can be achieved by returning r after executing $\mathsf{build}(Trips^{\perp}_{[q_{init},q]})$ for all $q \in \mathcal{Q}$.

Actual codes The preceding description elucidates the entire equation system generation process in a fairly thorough manner. Nonetheless, several nuanced technical aspects remain unexamined. An elegant rendition of the complete procedure, implemented in Haskell leveraging various Monad structures, is available in:
`https://github.com/ssyram/rtsa-mcfg/blob/main/src/EqSysBuild.hs`

9.4 Experimental Overview

Next, we delve into the empirical evaluation of PTSV. We have conducted a comprehensive series of experiments to confirm the effectiveness of PTSV, including a set of examples collected from the literature, which demonstrate its capability in handling existing cases within the domain. Additionally, we devised several series of parametrised examples, which can be easily scaled by adjusting parameters to further test the tool's limits.

These examples are categorised by their specific type – whether it is an automaton (rPTSA / pPDA) or a recursion scheme (PHORS / PAHORS) – and are introduced below in Section 9.5 and Section 9.6, respectively. These sections present case studies where the systems of equations discussed earlier are utilised to approximate *tp* and *ett*, and to decide AST and PAST.

Experimental Environment The experiments were conducted on a MacBook Pro (16-inch, 2019, Intel i7-9750H, 2.6 GHz, 32GB RAM). Each example was executed with a 500-second time-out.

9.4.1 Research Questions

Before delving into the details of the data, we pose several general questions that we aim to address through these experiments.

General Efficiency Is the tool efficient enough to effectively handle examples from the literature, including the conversion procedures and the overall solution time?

Equation Building Efficiency Given the exponential nature of the equation system's scale, particularly when dealing with rPTSA with larger restriction numbers or PAHORS with higher orders, is the tool scalable enough to manage these cases? Additionally, what are its potential limits?

Impact of Optimisations How do the optimisations presented contribute to the overall efficiency?

Conversion Efficiency To the best of our knowledge, this is the first implementation of the PAHORS to rPTSA conversion (which effectively includes the first implementation of the MAHORS to rTSA conversion) and the pPDA to 1-PTSA conversion. How efficient are these conversions? And, are they suitable for more practical applications?

Methodological Comparison We implemented various methods for obtaining results – through bisection and iteration – how do these approaches compare in terms of results and efficiency? Additionally, as we implemented different methods for resolving pPDA, what are the differences between the conversion-based and direct methods?

Potential Issues What are the major factors contributing to the running time, or even leading to a time-out?

9.4.2 Data Collection

For each example, we report two tables: one with data related to the termination probability (*tp*) and another for the expected termination time (*ett*).

The data collected represents the full capabilities of PTSV, including the determination of qualitative properties and the approximation of actual values via both bisection and iteration for *tp* and *ett*, which can be utilised using the `-a` CLI option.

To clarify, we introduce the meanings of the column headings, which are essentially symbolic representations of the output results for easier display in the tables.

tp **tables**

- AST? indicates whether the instance is almost-sure terminating, corresponding to "termination probability is Almost Sure Termination (AST)".
- $\approx_{\text{bisec}}$ provides the result of approximating termination probabilities using bisection, corresponding to "termination probability approximation value (bisection)".
- $\approx_{\text{iter}}$ offers the result of approximating termination probabilities using naive iteration, corresponding to "termination probability approximation value (iteration)".
- $t_{=}$ represents the total construction and optimisation time for the equation system, corresponding to "termination probability equation system total construction time".
- t_{AST} is the time taken by Z3 to answer the AST query, corresponding to "termination probability Almost Sure Termination (AST) time".
- t_{bisec} is the total time for approximation by bisection, including Z3 query time, corresponding to "termination probability approximation time (bisection)".

- t_{iter} is the time taken for iteration, corresponding to "termination probability approximation time (iteration)".

All times are reported in seconds.

- $\#^1_=$ is the number of equations after the first optimisation, corresponding to "termination probability equation system primitive scale".
- $\#^2_=$ is the number of equations after the second optimisation, corresponding to "termination probability equation system simplified scale".
- $\#_{\text{iter}}$ is the number of iterations performed, corresponding to "termination probability approximation (iteration) times".

ett **tables**

- $< \infty$? indicates if the second equation has a fixpoint, corresponding to "expected termination time has value".
- PAST? determines if the instance is positive almost-sure terminating, combining AST? and $< \infty$?, corresponding to "expected termination time is Positive Almost Sure Termination (PAST)".
- All other details ($\approx_{\text{bisec}}$, $\approx_{\text{iter}}$, $t_=$, t_{bisec}, t_{iter}, $\#^1_=$, $\#^2_=$) are analogous to those for termination probabilities but refer to $ett(-)$.
- $t_{<\infty}$ is the time taken by Z3 to determine if $ett(-) < \infty$, corresponding to "expected termination time qualitative time".

A typical example of the experimental data collection is as follows:

Example 9.7. Consider the example of random printer of rPTSA, i.e., Example 9.3. Its result is as below:

tp

Name	AST?	$\approx_{\text{bisec}}$	$\approx_{\text{iter}}$	$t_=$	t_{AST}	t_{bisec}	t_{iter}	$\#_{=1}$	$\#_{=2}$	$\#_{\text{iter}}$
Example 9.3	False	0.0142	0.0142	0.3078	0.0037	0.0000	0.0013	3	1	0

ett

Name	$< \infty$?	PAST?	$\approx_{\text{bisec}}$	$\approx_{\text{iter}}$	$t_=$	$t_{<\infty}$	t_{bisec}	t_{iter}	$\#_{=1}$	$\#_{=2}$	$\#_{\text{iter}}$
Example 9.3	True	False	0.0574	0.0574	0.1318	0.0631	0.0761	0.0123	6	1	2

9.5 Case Studies: Automata

To our knowledge, PTSV is the first tool dedicated to rPTSA. We are not aware of any benchmarks developed specifically for rPTSA, so here we propose several families of examples that rely crucially on tree stacks and cannot be captured with probabilistic pushdown automata. All the examples can be found at `https://github.com/ssyram/PTSV-Examples`.

In particular, we consider several families of relatively complicated examples. They illustrate that our methods can be used to verify quantitative properties of common randomly generated data structures, such as trees and queues.

9.5.1 Tree Shapes

Our first group of case studies is inspired by the language

$$B_m = \{\underbrace{w\#\cdots\#w}_{m} \mid w \in D,\, n \in \mathbb{N}\},$$

where $D ::= \epsilon \mid [\, D\,]\, D$ is the Dyck language. Note that the language relies on copying a well-bracketed word $w \in D$ a bounded number of times and, for $m > 1$, it cannot be context-free. It is also known that B_3 is not even indexed [**ES76**], i.e. it goes beyond order-2 pushdown automata ("stacks of stacks").

Instead of $w \in D$, we shall use binary trees, which could be viewed as parse trees for D, and consider rPTSA that generate random binary trees as a tree stack. This can be done by depth-first search: the rPTSA chooses at each stage (with probability $\frac{1}{2}$) whether the current node should be a binary node or a leaf. In the former case, the rPTSA performs up_{left} and, on return, continues with up_{right}. If the automaton chooses to create a leaf, it performs *down* and returns to its parent.

Random Paths (RP)

In the first case study, after generating the tree, the automaton will select a random path from the root to a leaf $m-1$ times by choosing fairly (with probability $\frac{1}{2}$) whether to go left or right. Once it reaches a leaf, it will (deterministically) backtrack to the root node to choose another random path, or to terminate if the run has already produced $(m-1)$ paths. We report the results for several values of m below. This example is confirmed as AST, but not PAST. It is interesting to note that the number of equations grows linearly and remains linear after the second round of optimisations, which halves the number of equations. Ultimately, equation generation at $m = 13$ becomes a bottleneck under our time limit of 500s. As *ett* is unbounded in this case, we do not report t_{bisec}, t_{iter}. The results are shown in Table 9.1.

Repeated Traversals (RT)

In the second case study, we will also generate random binary tree shapes, but will consider various probabilities p of generating a branching node: $p \in \{\frac{1}{2}, \frac{1}{3}, \frac{2}{3}\}$. In each case, following tree generation, we will perform $m-1$ full depth-first traversals of the generated tree. Before a traversal begins, the automaton will choose fairly (with probability $\frac{1}{2}$) whether to peform a left-to-right DFS or right-to-left DFS. The presence of this choice turns out to induce an exponential blow-up in the number of equations generated during the first stage ($\#^1_=$), but after further optimisations ($\#^2_=$) we will be left with 1 equation for *tp* and 3 for *ett*. Regardless of p, this allows us to generate the equations for $m \leq 13$ in the given time limit. If we removed the choice, i.e. performed left-to-right DFS each time, even the values of $\#^1_=$ would be constant, which would allow us to handle all values of m up to 1000 within the same time limit. Examples from this study are AST for $p = \frac{1}{2}, \frac{1}{3}$ and PAST for $p = \frac{1}{3}$. The generated data are available in Table 9.2.

tp

m	AST?	$\approx_{\text{bisec}}$	$\approx_{\text{iter}}$	$t_=$	t_{AST}	t_{bisec}	t_{iter}	$\#^1_=$	$\#^2_=$	$\#_{\text{iter}}$
1	True	1.0000	0.9986	0.1252	0.0139	0.0746	0.1021	3	1	1407
3	True	1.0000	0.9986	0.0019	0.0102	0.0572	0.2079	7	3	1407
5	True	1.0000	0.9986	0.0066	0.0053	0.0513	0.2663	11	5	1407
7	True	1.0000	0.9986	0.0233	0.0051	0.0534	0.3742	15	7	1407
9	True	1.0000	0.9986	0.1469	0.0063	0.0618	0.4971	19	9	1407
11	True	1.0000	0.9986	1.9686	0.0072	0.0679	0.6320	23	11	1407
13	True	1.0000	0.9986	56.1449	0.0069	0.0940	2.1178	27	13	1407

ett

m	$< \infty$?	PAST?	$\approx_{\text{bisec}}$	$\approx_{\text{iter}}$	$t_=$	$t_{<\infty}$	t_{bisec}	t_{iter}	$\#^1_=$	$\#^2_=$	$\#_{\text{iter}}$
1	False	False	-	-	0.0359	0.0099	-	-	6	3	-
3	False	False	-	-	0.0137	0.0067	-	-	14	7	-
5	False	False	-	-	0.0393	0.0022	-	-	22	11	-
7	False	False	-	-	0.1403	0.0034	-	-	30	15	-
9	False	False	-	-	0.6541	0.0038	-	-	38	19	-
11	False	False	-	-	6.5217	0.0028	-	-	46	23	-
13	False	False	-	-	159.4223	0.0053	-	-	54	27	-

Table 9.1: Result Tables for Random Paths

9.5.2 Queues

This example is inspired by the language $C_m = \{\underbrace{w \cdots w}_{m} \mid w \in \{a,b\}^*\}$, which is well known not to be context-free for $m > 1$. We will consider an rPTSA that generates a random finite list (queue) consisting of two types of customers: a and b. Technically, the rPTSA will choose a or b or 'end of queue' (with probabilities $p_A = \frac{1}{2}$, $p_B = \frac{1}{3}$ and $p_C = \frac{1}{6}$ respectively) while growing the tree stack upwards. The local memory will be used to remember the kind of customer. When queue formation is over, the automaton will move downwards to the root node. It will then make $m-1$ passes up and down the tree stack to "serve" the customers in the queue. Note that the automaton will be m-restricted and the tree stack will be a single branch of unbounded length.

Before each pass begins, one type of customers will be designated (at random) as angry. We assume that the probabilities involved are $\frac{1}{4}$ for a and $\frac{3}{4}$ for b. During the ith pass ($1 \leq i < m$), serving an angry customer can trigger immediate termination with probability $p_i = \frac{1}{i+1}$ i.e. service will continue with probability $1 - p_i$. Note that this means that the longer the service lasts the less likely an angry customer is to trigger termination (e.g. because servers get used to angry behaviour). When one kind of customer is viewed as angry, the other is considered nice: after meeting a nice customer the server will move on to the next customer (if one exists) with probability 1.

The corresponding statistics are available below. The number of equations grows

tp

p	m	AST?	$\approx_{\text{bisec}}$	$\approx_{\text{iter}}$	$t_=$	t_{AST}	t_{bisec}	t_{iter}	$\#_=^1$	$\#_=^2$	$\#_{\text{iter}}$
1/2	1	True	1.0000	0.9986	0.0007	0.0051	0.0435	0.2595	3	1	1407
	3	True	1.0000	0.9986	0.0022	0.0065	0.0600	0.4524	9	1	1407
	5	True	1.0000	0.9986	0.0090	0.0061	0.0566	0.3500	33	1	1407
	7	True	1.0000	0.9986	0.1584	0.0061	0.0552	0.3075	129	1	1407
	9	True	1.0000	0.9986	0.8906	0.0055	0.0545	0.3523	513	1	1407
	11	True	1.0000	0.9986	6.0025	0.0063	0.0582	0.3405	2049	1	1407
	13	True	1.0000	0.9986	62.9373	0.0058	0.0529	0.3146	8193	1	1407
1/3	1	True	1.0000	1.0000	0.0008	0.0069	0.0923	0.0017	3	1	30
	3	True	1.0000	1.0000	0.0029	0.0070	0.0676	0.0022	9	1	30
	5	True	1.0000	1.0000	0.0109	0.0070	0.0687	0.0018	33	1	30
	7	True	1.0000	1.0000	0.1626	0.0069	0.0679	0.0017	129	1	30
	9	True	1.0000	1.0000	0.8352	0.0069	0.0676	0.0024	513	1	30
	11	True	1.0000	1.0000	5.6161	0.0069	0.0661	0.0018	2049	1	30
	13	True	1.0000	1.0000	55.1997	0.0068	0.0657	0.0017	8193	1	30
2/3	1	False	0.5000	0.5000	0.0007	0.0000	0.0737	0.0016	3	1	28
	3	False	0.5000	0.5000	0.0023	0.0000	0.0801	0.0017	9	1	28
	5	False	0.5000	0.5000	0.0101	0.0000	0.0687	0.0017	33	1	28
	7	False	0.5000	0.5000	0.1628	0.0000	0.0672	0.0017	129	1	28
	9	False	0.5000	0.5000	0.9999	0.0000	0.0659	0.0019	513	1	28
	11	False	0.5000	0.5000	6.9816	0.0000	0.0678	0.0018	2049	1	28
	13	False	0.5000	0.5000	68.1158	0.0000	0.0657	0.0022	8193	1	28
	14	False	0.5000	0.5000	161.5296	0.0000	0.0667	0.0016	16385	1	28

ett

p	m	$< \infty$?	PAST?	$\approx_{\text{bisec}}$	$\approx_{\text{iter}}$	$t_=$	$t_{<\infty}$	t_{bisec}	t_{iter}	$\#_=^1$	$\#_=^2$	$\#_{\text{iter}}$
1/2	1	False	False	-	-	0.0015	0.0020	-	-	6	3	-
	3	False	False	-	-	0.0362	0.0029	-	-	18	3	-
	5	False	False	-	-	0.1194	0.0025	-	-	66	3	-
	7	False	False	-	-	0.3789	0.0038	-	-	258	3	-
	9	False	False	-	-	1.3698	0.0026	-	-	1026	3	-
	11	False	False	-	-	7.9520	0.0027	-	-	4098	3	-
	13	False	False	-	-	74.4881	0.0024	-	-	16386	3	-
1/3	1	True	True	7.0000	7.0000	0.0395	0.0031	0.0954	0.0407	6	3	45
	3	True	True	19.0000	19.0000	0.0193	0.0028	0.1113	0.0400	18	3	48
	5	True	True	31.0000	31.0000	0.0841	0.0031	0.1087	0.0390	66	3	50
	7	True	True	43.0000	43.0000	0.3007	0.0034	0.1176	0.0423	258	3	50
	9	True	True	55.0000	55.0000	1.5330	0.0028	0.1141	0.0442	1026	3	51
	11	True	True	67.0000	67.0000	7.3116	0.0028	0.1590	0.0086	4098	3	52
	13	True	True	79.0000	79.0000	81.5300	0.0029	0.1288	0.0400	16386	3	52
2/3	1	True	False	3.5000	3.5000	0.0237	0.0031	0.1090	0.0425	6	3	44
	3	True	False	9.5000	9.5000	0.0234	0.0029	0.1317	0.0453	18	3	46
	5	True	False	15.5000	15.5000	0.0855	0.0029	0.1035	0.0427	66	3	48
	7	True	False	21.5000	21.5000	0.3243	0.0030	0.1275	0.0434	258	3	49
	9	True	False	27.5000	27.5000	1.5422	0.0029	0.1124	0.0431	1026	3	49
	11	True	False	33.5000	33.5000	9.4813	0.0030	0.1393	0.0447	4098	3	50
	13	True	False	39.5000	39.5000	84.3588	0.0028	0.1520	0.0083	16386	3	50
	14	True	False	42.5000	42.5000	351.0406	0.0029	0.1196	0.0432	32770	3	50

Table 9.2: Result tables for Repeated Traversals

exponentially in this case during the first phase ($\#^1_=$), but is reduced to 1 after optimisation. This allows us handle $m \leq 11$ given our time allowance. When the ability to randomly designate angry customers is removed (i.e. a is assumed to be angry and b is nice), $\#^1_=$ grows linearly, which makes it possible to handle $m \leq 200$.

tp

m	AST?	$\approx_{\text{bisec}}$	$\approx_{\text{iter}}$	$t_=$	t_{AST}	t_{bisec}	t_{iter}	$\#^1_=$	$\#^2_=$	$\#_{\text{iter}}$
1	True	1.0000	1.0000	0.1211	0.0016	0.0000	0.0013	2	1	0
3	True	1.0000	1.0000	0.0033	0.0000	0.0000	0.0000	15	1	0
5	True	1.0000	1.0000	0.0211	0.0000	0.0000	0.0000	63	1	0
7	True	1.0000	1.0000	0.1155	0.0000	0.0000	0.0000	255	1	0
9	True	1.0000	1.0000	0.8969	0.0000	0.0000	0.0000	1023	1	0
11	True	1.0000	1.0000	17.5876	0.0000	0.0000	0.0000	4095	1	0

ett

m	$< \infty$?	PAST?	$\approx_{\text{bisec}}$	$\approx_{\text{iter}}$	$t_=$	$t_{<\infty}$	t_{bisec}	t_{iter}	$\#^1_=$	$\#^2_=$	$\#_{\text{iter}}$
1	True	True	13.0000	13.0000	0.0411	0.0299	0.0794	0.0078	4	1	2
3	True	True	19.2166	19.2166	0.0189	0.0083	0.0783	0.0001	30	1	2
5	True	True	22.0299	22.0299	0.0688	0.0019	0.0615	0.0002	126	1	2
7	True	True	24.2401	24.2401	0.5045	0.0025	0.0619	0.0003	510	1	2
9	True	True	26.1653	26.1653	4.1441	0.0031	0.0816	0.0003	2046	1	2
11	True	True	27.9195	27.9195	144.9703	0.0085	0.3420	0.0014	8190	1	2

9.5.3 Probabilistic Pushdown Automata

Although PTSV is dedicated to analysing PTSA, it also accepts probabilistic pushdown automata (pPDA) and probabilistic basic process algebra (pBPA) as inputs. It is known that pPDA can be translated into 1-restricted PTSA; however, the conversion is not immediate, as detailed above in Section 8.1.2.

Because the equation construction via translation to 1-PTSA differs from the classic direct method of deriving systems of equations for pPDA [110], we have implemented both approaches inside PTSV. Using pPDA examples from [105, 139], we then compared their performance, finding that the direct approach is usually (though not always) faster, and both methods can handle most examples within the allotted time.

pPDA examples from [139]

The work of [139] collected a large set of examples from the literature on pPDA, many of which are converted from actual programs using a standard translation. We used these examples in our experiments comparing the conversion-based method (converting pPDA to 1-PTSA) and the direct equation system generation method as in [87, 110]. The results, displayed in Table 9.3, include cases where some examples experienced time-outs with the Z3 SMT solver, marked as *t.o.*. Besides, with the conversion-based method, we added a new column t_{conv} denoting the conversion time from pPDA to 1-PTSA.

Upon observing the data, one may note that the conversion time is generally fast and even somewhat trivial. However, it is important to highlight that the equation systems generated for both methods differ, making them variably sensitive to optimisation procedures. For instance, the "virus" case demonstrates the conversion method's efficacy in reducing the equation system from 566 to 1 in 8 seconds, whereas the direct method resulted in an equation system leading to a time-out with Z3.

Additionally, although the conversion produces primitive equation systems with larger scales, the optimisation procedure is generally capable of reducing the generated system to a significantly smaller scale, often comparable with or even smaller than the one from the direct method. This showcases the overall capability of our optimisation procedure.

On the other hand, the different shapes of the generated equation systems themselves have a significant impact on the efficiency of the Z3 SMT solver, leading to time-outs in various cases. These results suggest that both methods have their respective strengths and potential for further optimisation.

"and-or-tree" Examples from [105] Further, we also confirmed the correctness of PTSV by testing it on the "and-or-tree" examples from [105], which has been introduced above as Example 2.3. The dataset is generally large; we have tested all the data, but we will present only a portion here for the sake of brevity. The results for both conversion-based and direct methods are reported in Table 9.4.

For this example, the Z3 solver times out for most of the examples involving the expected termination time equation system. Hence, we present only the iteration-based approximation method $\approx_{\text{iter}}$. Note also that the data in [105] concerns *conditional* expected termination time, which is the $cett(-)$ value, defined as the expected termination time divided by the termination probability. Therefore, we also present the column of $\approx_{\text{cond}}$ for the conditional expected termination value, obtained by dividing $\approx_{\text{iter}}$ in the *ett* table by $\approx_{\text{iter}}$ in the corresponding *tp* table.

As an additional note, one may observe that the conditional expected termination time value computed by PTSV is 2 more than the corresponding value reported in the original paper [105]. This discrepancy arises because the examples in the original work start with a stack that already contains an element. Initialising this stack configuration contributes an extra step at the beginning, involving a move *up* from the bottom node. Additionally, the termination process requires one extra step at the bottom of the stack.

9.6 Case Studies: Recursion Schemes

Another family of inputs our tool can handle are *probabilistic higher-order recursion schemes* (PHORS) [84], which can be viewed as abstractions of randomised functional programs. To be more specific, as discussed, the family of models supported by PTSV is the PHORS with affine constraint and product types.

To confirm the effectiveness of our construction, we implemented the PAHORS to rPTSA conversion based on the processes detailed above in Section 8.2.3. Then, the tool will process the converted rPTSA model by the developed algorithms.

tp

Method	Example	t_{conv}	AST?	$\approx_{\text{bisec}}$	$\approx_{\text{iter}}$	$t_{=}$	t_{AST}	t_{bisec}	t_{iter}	$\#_{=1}$	$\#_{=2}$	$\#_{\text{iter}}$
Conversion	escapeN 3	0.0043	*t.o.*	*t.o.*	0.6499	0.3382	*t.o.*	*t.o.*	0.0609	111	28	55
	rw-p	0.0009	True	1.0000	1.0000	0.2081	0.0153	0.0986	0.0076	5	1	25
	modN 3	0.0127	True	1.0000	1.0000	0.2706	0.0209	0.1552	0.0140	95	8	41
	golden	0.0034	True	1.0000	1.0000	0.2236	0.0162	0.0987	0.0217	11	2	148
	geom-offspring	0.0033	False	0.7895	0.7895	0.2537	0.0010	0.1099	0.0211	18	3	93
	gen-fun	0.0029	True	1.0000	1.0000	0.2409	0.0159	0.0977	0.0075	21	2	27
	virus	0.0203	False	0.1066	0.1066	8.3426	0.0010	0.0000	0.0006	566	1	0
	sequentialN 5	0.0083	*t.o.*	*t.o.*	1.0000	0.4674	*t.o.*	*t.o.*	0.0090	212	9	21
	and-or-tree-1	0.1846	False	0.4189	0.4189	0.8041	0.0016	0.3571	0.0637	99	14	47
	and-or-tree-0	0.1829	False	0.5811	0.5811	0.7098	0.0016	0.4888	0.0716	99	14	47
Direct	escapeN 3	-	False	0.6499	0.6499	0.2489	0.0010	0.3301	0.0126	4	3	39
	rw-p	-	True	1.0000	1.0000	0.2297	0.0150	0.1038	0.0073	2	1	25
	modN 3	-	True	1.0000	1.0000	0.2692	0.0157	0.1073	0.0086	6	3	47
	golden	-	True	1.0000	1.0000	0.2021	0.0156	0.1152	0.0176	2	1	87
	geom-offspring	-	False	0.7895	0.7895	0.2657	0.0010	0.1137	0.0150	4	2	51
	gen-fun	-	True	1.0000	1.0000	0.2538	0.0154	0.1155	0.0082	3	2	27
	virus	-	*t.o.*	*t.o.*	0.1066	0.0117	*t.o.*	*t.o.*	0.0070	7	6	19
	sequentialN 5	-	*t.o.*	*t.o.*	1.0000	0.0157	*t.o.*	*t.o.*	0.0187	15	5	36
	and-or-tree-1	-	False	0.4189	0.4189	0.0732	0.0000	0.1410	0.0100	16	4	46
	and-or-tree-0	-	False	0.5811	0.5811	0.0784	0.0000	0.1406	0.0110	16	4	46

ett

Method	Name	t_{conv}	$< \infty$?	PAST?	$\approx_{\text{bisec}}$	$\approx_{\text{iter}}$	$t_{=}$	$t_{<\infty}$	t_{bisec}	t_{iter}	$\#_{=1}$	$\#_{=2}$	$\#_{\text{iter}}$
Conversion	escapeN 3	0.0043	*t.o.*	*t.o.*	*t.o.*	1.9841	1.5385	*t.o.*	*t.o.*	0.2467	204	55	83
	rw-p	0.0009	True	True	2.5000	2.5000	0.0446	0.0100	0.1086	0.0057	10	2	34
	modN 3	0.0127	True	True	13.0000	13.0000	0.2626	0.0060	0.4155	0.0783	156	22	88
	golden	0.0034	True	True	10.0000	10.0000	0.0438	0.0100	0.1231	0.0656	22	4	276
	geom-offspring	0.0033	True	False	5.7548	5.7548	0.0840	0.0046	0.1560	0.0735	35	8	171
	gen-fun	0.0029	True	True	6.0000	6.0000	0.0830	0.0051	0.1253	0.0106	41	5	37
	virus	0.0203	True	False	0.4294	0.4294	74.6859	0.0225	0.0858	0.0087	1068	1	2
	sequentialN 5	0.0083	*t.o.*	*t.o.*	*t.o.*	27.2132	1.7427	*t.o.*	*t.o.*	0.1133	368	34	89
	and-or-tree-1	0.1846	*t.o.*	*t.o.*	*t.o.*	3.8419	1.2348	*t.o.*	*t.o.*	0.1313	173	29	73
	and-or-tree-0	0.1829	*t.o.*	*t.o.*	*t.o.*	5.3204	1.3729	*t.o.*	*t.o.*	0.1367	173	29	73
Direct	escapeN 3	-	True	False	1.9841	1.9841	0.0361	0.0139	21.5221	0.0193	8	6	57
	rw-p	-	True	True	2.5000	2.5000	0.0421	0.0107	0.1085	0.0071	4	2	34
	modN 3	-	True	True	13.0000	13.0000	0.0232	0.0106	0.1545	0.0280	12	7	81
	golden	-	True	True	10.0000	10.0000	0.0469	0.0123	0.1184	0.0408	4	2	163
	geom-offspring	-	True	False	5.7548	5.7548	0.0332	0.0074	0.1580	0.0442	8	5	102
	gen-fun	-	True	True	6.0000	6.0000	0.0233	0.0117	0.1284	0.0124	6	3	36
	virus	-	*t.o.*	*t.o.*	*t.o.*	0.4294	0.0555	*t.o.*	*t.o.*	0.0218	14	12	24
	sequentialN 5	-	*t.o.*	*t.o.*	*t.o.*	27.2132	0.1094	*t.o.*	*t.o.*	0.0678	30	9	54
	and-or-tree-1	-	True	False	3.8419	3.8419	0.0784	0.0158	0.4118	0.1070	32	13	89
	and-or-tree-0	-	True	False	5.3204	5.3204	0.1036	0.0226	0.5342	0.1071	32	13	89

Table 9.3: Experimental Results of Examples from [139]

Note that the values for *ett* below correctly reflect those of the original recursion schemes, by counting only the reduction steps in the converted rPTSA, as mentioned in the conversion. Furthermore, the reported value is equivalent to $\sum_{n=0}^{\infty} n \cdot \left(\sum_{\zeta \in Red_n^{\top}(\mathcal{G})} tp(\zeta)\right)$.

Example 9.8. When the random printer example, as shown in Example 9.2, is put into PTSV, it generates the following results.

tp

AST?	$\approx_{\text{bisec}}$	$\approx_{\text{iter}}$	t_{conv}	$t_{=}$	t_{AST}	t_{bisec}	t_{iter}	$\#_{=}^{1}$	$\#_{=}^{2}$	$\#_{\text{iter}}$
False	0.0645	0.0645	0.0040	0.0025	0.0000	0.0000	0.0000	42	1	0

tp

Method	Parameter	t_{conv}	AST?	$\approx_{\text{bisec}}$	$\approx_{\text{iter}}$	$t_=$	t_{AST}	t_{bisec}	t_{iter}	$\#_{=1}$	$\#_{=2}$	$\#_{\text{iter}}$
Conversion	$[a\|0](0.5, 0.4, 0.2, 0.2)$	0.1635	False	0.5000	0.5000	0.5356	0.0017	0.2601	0.0681	67	8	81
	$[a\|1](0.5, 0.4, 0.2, 0.2)$	0.0026	False	0.3000	0.3000	0.1080	0.0000	0.1993	0.0476	67	8	81
	$[a](0.5, 0.4, 0.2, 0.2)$	0.0026	False	0.8000	0.8000	0.1118	0.0000	0.1719	0.0505	72	9	82
	$[a\|0](0.5, 0.5, 0.1, 0.1)$	0.0047	False	0.5556	0.5556	0.1060	0.0000	0.1993	0.0527	67	8	92
	$[a\|1](0.5, 0.5, 0.1, 0.1)$	0.0022	False	0.3056	0.3056	0.0772	0.0000	0.1776	0.0502	67	8	92
	$[a](0.5, 0.5, 0.1, 0.1)$	0.0026	False	0.8611	0.8611	0.0975	0.0000	0.2443	0.0584	72	9	92
	$[a\|0](0.2, 0.4, 0.2, 0.2)$	0.0022	False	0.6957	0.6956	0.1063	0.0000	0.2018	0.0534	67	8	77
	$[a\|1](0.2, 0.4, 0.2, 0.2)$	0.0030	False	0.1148	0.1148	0.1023	0.0000	0.2763	0.0827	67	8	77
	$[a](0.2, 0.4, 0.2, 0.2)$	0.0030	False	0.8104	0.8104	0.1219	0.0000	0.2152	0.0486	72	9	77
	$[a\|0](0.4, 0.4, 0.2, 0.2)$	0.0026	False	0.5714	0.5714	0.1004	0.0001	0.2120	0.0515	67	8	81
	$[a\|1](0.4, 0.4, 0.2, 0.2)$	0.0027	False	0.2362	0.2362	0.1075	0.0000	0.2599	0.0636	67	8	81
	$[a](0.4, 0.4, 0.2, 0.2)$	0.0023	False	0.8076	0.8076	0.1144	0.0000	0.2323	0.0583	72	9	81
Direct	$[a\|0](0.5, 0.4, 0.2, 0.2)$	-	False	0.5000	0.5000	0.1241	0.0000	0.2013	0.0443	30	8	81
	$[a\|1](0.5, 0.4, 0.2, 0.2)$	-	False	0.3000	0.3000	0.0452	0.0000	0.2297	0.0517	30	8	81
	$[a](0.5, 0.4, 0.2, 0.2)$	-	False	0.8000	0.8000	0.0538	0.0000	0.2049	0.0480	30	9	82
	$[a\|0](0.5, 0.5, 0.1, 0.1)$	-	False	0.5556	0.5556	0.0476	0.0000	0.1982	0.0540	30	8	92
	$[a\|1](0.5, 0.5, 0.1, 0.1)$	-	False	0.3056	0.3056	0.0397	0.0000	0.2275	0.0555	30	8	92
	$[a](0.5, 0.5, 0.1, 0.1)$	-	False	0.8611	0.8611	0.0505	0.0000	0.2081	0.0591	30	9	92
	$[a\|0](0.2, 0.4, 0.2, 0.2)$	-	False	0.6957	0.6956	0.0491	0.0000	0.2123	0.0487	30	8	77
	$[a\|1](0.2, 0.4, 0.2, 0.2)$	-	False	0.1148	0.1148	0.0495	0.0000	0.3206	0.0531	30	8	77
	$[a](0.2, 0.4, 0.2, 0.2)$	-	False	0.8104	0.8104	0.0561	0.0000	0.2545	0.0530	30	9	77
	$[a\|0](0.4, 0.4, 0.2, 0.2)$	-	False	0.5714	0.5714	0.0365	0.0000	0.1852	0.0522	30	8	81
	$[a\|1](0.4, 0.4, 0.2, 0.2)$	-	False	0.2362	0.2362	0.0380	0.0000	0.2497	0.0514	30	8	81
	$[a](0.4, 0.4, 0.2, 0.2)$	-	False	0.8076	0.8076	0.0469	0.0000	0.2008	0.0537	30	9	81

ett

Method	Parameter	t_{conv}	$\approx_{\text{iter}}$	$\approx_{\text{cond}}$	$t_=$	t_{iter}	$\#_{=1}$	$\#_{=2}$	$\#_{\text{iter}}$
Conversion	$[a\|0](0.5, 0.4, 0.2, 0.2)$	0.1635	6.5000	13.0000	0.4796	0.1629	110	17	141
	$[a\|1](0.5, 0.4, 0.2, 0.2)$	0.0026	2.9000	9.6667	0.3459	0.1398	110	17	141
	$[a](0.5, 0.4, 0.2, 0.2)$	0.0026	9.4000	11.7500	0.3712	0.1406	116	17	142
	$[a\|0](0.5, 0.5, 0.1, 0.1)$	0.0047	7.4444	13.3988	0.3953	0.1662	110	17	164
	$[a\|1](0.5, 0.5, 0.1, 0.1)$	0.0022	2.2944	7.5079	0.3792	0.1752	110	17	164
	$[a](0.5, 0.5, 0.1, 0.1)$	0.0026	9.7389	11.3098	0.4756	0.2390	116	17	164
	$[a\|0](0.2, 0.4, 0.2, 0.2)$	0.0022	6.8364	9.8266	0.3969	0.1770	110	17	133
	$[a\|1](0.2, 0.4, 0.2, 0.2)$	0.0030	0.9488	8.2648	0.4233	0.1867	110	17	133
	$[a](0.2, 0.4, 0.2, 0.2)$	0.0030	7.7852	9.6066	0.4446	0.2031	116	17	133
	$[a\|0](0.4, 0.4, 0.2, 0.2)$	0.0026	6.8598	12.0053	0.4705	0.2901	110	17	140
	$[a\|1](0.4, 0.4, 0.2, 0.2)$	0.0027	2.1867	9.2578	0.4727	0.2044	110	17	140
	$[a](0.4, 0.4, 0.2, 0.2)$	0.0023	9.0465	11.2017	0.4633	0.2070	116	17	141
Direct	$[a\|0](0.5, 0.4, 0.2, 0.2)$	-	6.5000	13.0000	0.2255	0.1610	60	17	141
	$[a\|1](0.5, 0.4, 0.2, 0.2)$	-	2.9000	9.6667	0.2226	0.2620	60	17	141
	$[a](0.5, 0.4, 0.2, 0.2)$	-	9.4000	11.7500	0.2418	0.1816	60	17	142
	$[a\|0](0.5, 0.5, 0.1, 0.1)$	-	7.4444	13.3988	0.2206	0.2014	60	17	164
	$[a\|1](0.5, 0.5, 0.1, 0.1)$	-	2.2944	7.5079	0.2474	0.2239	60	17	164
	$[a](0.5, 0.5, 0.1, 0.1)$	-	9.7389	11.3098	0.2485	0.2045	60	17	164
	$[a\|0](0.2, 0.4, 0.2, 0.2)$	-	6.8364	9.8281	0.2346	0.1996	60	17	133
	$[a\|1](0.2, 0.4, 0.2, 0.2)$	-	0.9488	8.2648	0.2559	0.1897	60	17	133
	$[a](0.2, 0.4, 0.2, 0.2)$	-	7.7852	9.6066	0.2843	0.1857	60	17	133
	$[a\|0](0.4, 0.4, 0.2, 0.2)$	-	6.8598	12.0053	0.2511	0.2015	60	17	140
	$[a\|1](0.4, 0.4, 0.2, 0.2)$	-	2.1867	9.2578	0.3910	0.2487	60	17	140
	$[a](0.4, 0.4, 0.2, 0.2)$	-	9.0465	11.2017	0.2347	0.1816	60	17	141

Table 9.4: Results for "and-or-tree" Examples

tp

Example	AST?	$\approx_{\text{bisec}}$	$\approx_{\text{iter}}$	t_{conv}	$t_{=}$	t_{AST}	t_{bisec}	t_{iter}	$\#^1_{=}$	$\#^2_{=}$	$\#_{\text{iter}}$
Ex2.3	True	1.0000	1.0000	0.0907	0.1157	0.0038	0.0000	0.0037	2	1	0
Listgen	True	1.0000	1.0000	0.0005	0.0328	0.0000	0.0000	0.0000	10	1	0
Treegen	False	0.6180	0.6180	0.0005	0.0071	0.0001	0.0862	0.0033	14	2	21
ListEven	False	0.6667	0.6667	0.0009	0.0005	0.0000	0.0000	0.0000	8	1	0
ListEven2	False	0.7500	0.7500	0.0151	0.0038	0.0000	0.0566	0.0002	24	2	19
Ex5.4(0, 0)	False	0.0000	0.0000	0.0020	0.0018	0.0002	0.0000	0.0000	1	1	0
Ex5.4(0.3, 0.3)	False	0.3333	0.3333	0.0011	0.0044	0.0000	0.0589	0.0005	28	2	9
Ex5.4(0.5, 0.5)	True	1.0000	0.9990	0.0012	0.0036	0.0075	0.0377	0.1007	28	2	1991
TreeEven(0.5)	False	0.2929	0.2922	0.0007	0.0045	0.0000	0.0445	0.1828	20	2	1406
TreeEven(0.49)	False	0.2774	0.2774	0.0007	0.0032	0.0000	0.0516	0.0435	20	2	324
TreeEven(0.51)	False	0.2887	0.2887	0.0006	0.0033	0.0000	0.0500	0.0413	20	2	328
Discont(.01, .99)	True	1.0000	1.0000	0.0002	0.0004	0.0000	0.0000	0.0000	8	1	0
Example 4.9	False	0.5000	0.5000	0.0003	0.0005	0.0000	0.0000	0.0000	11	1	0

ett

Example	$< \infty$?	PAST?	$\approx_{\text{bisec}}$	$\approx_{\text{iter}}$	$t_{=}$	$t_{<\infty}$	t_{bisec}	t_{iter}	$\#^1_{=}$	$\#^2_{=}$	$\#_{\text{iter}}$
Ex2.3	True	True	1.0000	1.0000	0.0439	0.0321	0.0589	0.0074	4	1	2
Listgen	True	True	5.0000	5.0000	0.0134	0.0087	0.0686	0.0000	18	1	2
Treegen	True	False	2.0652	2.0652	0.0347	0.0041	0.0968	0.0104	25	5	31
ListEven	True	False	1.7778	1.7778	0.0021	0.0094	0.0609	0.0003	15	3	21
ListEven2	True	False	3.0000	3.0000	0.0174	0.0088	0.0695	0.0007	39	5	27
Ex5.4(0, 0)	True	False	0.0000	0.0000	0.0017	0.0071	0.1115	0.0070	2	1	1
Ex5.4(0.3, 0.3)	True	False	1.5833	1.5833	0.0136	0.0071	0.0671	0.0021	42	5	12
Ex5.4(0.5, 0.5)	False	False	-	-	0.0125	0.0023	-	-	42	5	-
TreeEven(0.5)	False	False	-	-	0.0193	0.0022	-	-	33	6	-
TreeEven(0.49)	True	False	44.6515	44.6515	0.0197	0.0024	0.3961	0.3168	33	6	879
TreeEven(0.51)	True	False	46.4740	46.4740	0.0222	0.0023	0.3927	0.3190	33	6	883
Discont(.01, .99)	True	True	201.00	201.00	0.0007	0.0021	0.0551	0.0000	14	1	2
Example 4.9	True	False	2.0000	2.0000	0.0005	0.0015	0.0472	0.0000	16	1	2

Table 9.5: Results for PAHORS

ett

$< \infty$?	PAST?	$\approx_{\text{bisec}}$	$\approx_{\text{iter}}$	$t_{=}$	$t_{<\infty}$	t_{bisec}	t_{iter}	$\#^1_{=}$	$\#^2_{=}$	$\#_{\text{iter}}$
True	False	0.3309	0.3309	0.0071	0.0087	0.0517	0.0000	70	1	2

To further evaluate the capability of PTSV in handling PAHORS as input, we tested our tool on two sets of benchmarks: one comprising examples collected from [84] and another consisting of several example families we developed.

Examples from [84]

To verify the accuracy of our method, we tested PTSV on a set of PAHORS examples primarily sourced from [84], which are derived from various randomised algorithms. The Table 9.5 presents our results on this front, which have been verified to be consistent with the results reported in the original paper [84]. The experimental results include both upper and lower bounds for *tp*. Meanwhile, we are the first to report the expectation results for these examples.

tp

m	AST?	$\approx_{\text{bisec}}$	$\approx_{\text{iter}}$	t_{conv}	$t_{=}$	t_{AST}	t_{bisec}	t_{iter}	$\#_{=1}$	$\#_{=2}$	$\#_{\text{iter}}$
1	True	1.0000	1.0000	0.1219	0.2198	0.0047	0.0000	0.0042	6	1	0
3	False	0.6180	0.6180	0.0004	0.0037	0.0002	0.0600	0.0023	14	2	21
5	False	0.5188	0.5188	0.0008	0.0282	0.0000	0.0540	0.0011	22	2	8
6	False	0.5087	0.5087	0.0010	0.0834	0.0000	0.0462	0.0006	26	2	6
7	False	0.5041	0.5041	0.0011	0.3370	0.0000	0.0478	0.0004	30	2	5
8	False	0.5020	0.5020	0.0025	1.4606	0.0000	0.0483	0.0005	34	2	5
9	False	0.5010	0.5010	0.0019	8.0699	0.0000	0.0703	0.0007	38	2	4

ett

m	$< \infty$?	PAST?	$\approx_{\text{bisec}}$	$\approx_{\text{iter}}$	$t_{=}$	$t_{<\infty}$	t_{bisec}	t_{iter}	$\#_{=1}$	$\#_{=2}$	$\#_{\text{iter}}$
1	True	True	3.0000	3.0000	0.0496	0.0580	0.0712	0.0078	11	1	2
3	True	False	2.0652	2.0652	0.0210	0.0070	0.0939	0.0082	25	5	31
5	True	False	1.1523	1.1523	0.2523	0.0028	0.1666	0.0047	39	5	11
6	True	False	1.0752	1.0752	0.7236	0.0022	0.4625	0.0021	46	5	9
7	True	False	1.0390	1.0390	2.8042	0.0027	1.7653	0.0023	53	5	8
8	True	False	1.0207	1.0207	15.8502	0.0025	8.7385	0.0024	60	5	7
9	True	False	1.0111	1.0111	92.3280	0.0026	55.5413	0.0021	67	5	6

Table 9.6: Results for NTreegen Series

While not all PAHORS in [84] initially appear additive, most can be converted to additive form by recognising that the arguments reused on their RHSs remain unchanged, allowing replacement by non-terminals that can be shared without restrictions. Furthermore, Table 9.5 includes Example 4.9 from [84], which could not be addressed in the original paper due to its order being 3. Likewise, the order-3 random printer example above (i.e., Example 9.2) cannot be handled by the original work as well.

NTreegen and VTreegen Examples

As can be observed from the data, none of the examples from [84] presented a substantial challenge for the tool. To further explore the limits of PTSV, we consider two additional series of examples – NTreegen and VTreegen – which produce rPTSA with high restriction parameters.

Example 9.9. The example series NTreegen(m) is defined by the rules: $S \to F\top$, $F\,x \xrightarrow{1/2} x$, $F\,x \xrightarrow{1/2} F^m\,x$, where $F : o \to o$ and $m \geq 1$. This translates into an m-restricted rPTSA.

Within the time constraints of our experiments, PTSV can manage NTreegen for $m \leq 9$. We observe a linear increase in the number of equations after the first optimisation phase, as shown in Table 9.6.

We also consider a variant of the aforementioned series, namely VTreegen.

Example 9.10. The series VTreegen(m) is defined by the rules: $S \to F\top$ and

tp

m	AST?	$\approx_{\text{bisec}}$	$\approx_{\text{iter}}$	t_{conv}	$t_{=}$	t_{AST}	t_{bisec}	t_{iter}	$\#^1_{=}$	$\#^2_{=}$	$\#_{\text{iter}}$
1	True	1.0000	1.0000	0.0002	0.0004	0.0000	0.0000	0.0000	6	1	0
2	True	1.0000	0.9988	0.0004	0.0047	0.0122	0.0525	0.3376	14	2	2440
3	False	0.4142	0.4142	0.0010	0.0559	0.0000	0.0573	0.0084	26	3	23
4	False	0.2757	0.2757	0.0028	3.4690	0.0000	0.0926	0.0134	42	4	13

ett

m	$< \infty$?	PAST?	$\approx_{\text{bisec}}$	$\approx_{\text{iter}}$	$t_{=}$	$t_{<\infty}$	t_{bisec}	t_{iter}	$\#^1_{=}$	$\#^2_{=}$	$\#_{\text{iter}}$
1	True	True	3.0000	3.0000	0.0016	0.0023	0.0565	0.0002	11	1	2
2	False	False	-	-	0.0646	0.0093	-	-	26	5	-
3	True	False	1.4142	1.4142	0.6747	0.0034	0.1933	0.0955	49	7	32
4	True	False	0.7151	0.7151	33.8611	0.0065	4.5098	0.1052	80	9	18

Table 9.7: Results for VTreegen Series

$F\,x \xrightarrow{1/m+1} F^i\,x$, where $F : o \to o$ and $0 \leq i \leq m$. This translates into an m-restricted rPTSA.

PTSV can handle VTreegen(m) for $m \leq 4$, and we observe exponential growth in the number of equations after the first optimisation phase. The results are depicted in Table 9.7.

9.7 Experimental Conclusions

Having presented the detailed experimental data, we are now prepared to conclude the experiment by addressing the research questions posed at the outset. The results obtained provide a comprehensive overview of the tool's performance across different aspects, allowing us to draw informed conclusions regarding its efficiency and potential limitations.

General Efficiency As observed from the data, PTSV operates at a reasonable speed across the examples, with only minimal time-outs. Almost all examples sourced from the literature are resolved within seconds, with only a few experiencing time-outs due to Z3. Furthermore, among those that did not time-out, only a small number of specially devised examples took longer than a minute to complete. This indicates that the tool can handle a wide range of inputs effectively. Thus, it can be reasonably argued that the tool demonstrates satisfactory general efficiency, making it suitable for practical applications in typical scenarios.

Equation Building Efficiency Despite the exponential complexity inherent in the equation system's scale, the data shows that building the equation systems was rarely a bottleneck across the examples from the literature, with most equation systems being constructed within a second. For certain specially designed examples with a larger number of restrictions or higher order, the building process takes longer. This is attributed to the traversal procedure, which may struggle with the

exponential number of possible synchronisation paths. Nevertheless, even in these cases, the delay is manageable and does not severely impact the overall performance. Overall, the building procedure can be considered robust and efficient, effectively managing the inherent complexity of the problem space.

Impact of Optimisations The experimental data suggests that the efficiency of both the overall procedure and the equation system building can be largely attributed to the optimisations introduced. In particular, it is evident that the overall scales of the equation systems built are often significantly smaller than the exponential scales indicated by the theoretical results. This reduction in scale is a foundational factor enabling the tool to run the examples effectively and is central to the efficiency of the equation system building process.

Moreover, the simplification procedure applied to the primitive equation system proves to be highly effective, reducing the number of equations by at least half, and in some extreme cases, by several orders of magnitude. In many instances, the simplification procedure alone suffices to reach the target value. Notably, this procedure is often crucial, as without it, Z3 would frequently time out. Thus, the optimisations not only enhance the tool's efficiency but are also vital in making the tool practical in performing these analyses.

Conversion Efficiency As expected, the data confirms that the conversion procedure from pPDA to 1-PTSA, given its original linear complexity, incurs negligible time, underscoring its efficiency. Similarly, the conversion from PAHORS to rPTSA is also highly efficient, with the process completing in very short times across the collected examples. In summary, the data affirms the robustness and efficiency of the conversion processes, indicating their suitability for handling practical cases. This efficiency in conversion ensures that the tool can seamlessly integrate different formalisms, broadening its applicability.

Methodological Comparison The data shows that, under the default settings, the results from iteration and bisection differ only slightly, mostly yielding the same value. However, as anticipated, due to the multiple calls to Z3, bisection tends to take longer and sometimes even results in time-outs, whereas iteration is more robust, never timing out across the examples. An empirical observation here is that Z3 is particularly sensitive to the "shape" of the given equation system, leading to observable differences in efficiency between the conversion-based method and the direct method for pPDA examples.

Specifically, these methods generate different equation systems for Z3, resulting in varying efficiency. As expected, the direct method is generally faster than the conversion-based method. Notably, this difference is also influenced by the additional representation complexity from the rewriting-rule-based representation (i.e., Equation (8.1)) to the operation-based representation (i.e., Definition 8.2), where the latter produces many more control states and more rules to manage.

Potential Issues In most cases, the primary issue leading to time-outs stems from Z3. However, some problems can only be solved with the assistance of Z3, such as AST, PAST, and bisection approximation. Fundamentally, Z3 is a crucial

component of the workflow, as it is essential to first confirm the existence of a solution to the *ett* equation system before computing the approximation, to avoid potential errors such as infinite iterations.

Additionally, another issue was found in certain devised examples, such as the repeated traversal (RT) series, where the current equation system building procedure, based on exploring synchronisation paths, is often hindered by the exponential number of paths. A potential future direction could involve developing a more sophisticated procedure for exploring and representing the set of synchronisation paths. Such advancements would not only improve the efficiency of the tool but also expand its capability to handle even more complex scenarios without compromising on performance.

9.8 Related Tools

PreMo [140] is a software tool for analysing probabilistic models based on recursive Markov chains, which are equivalent to pushdown systems. Theoretically, it also relies on fixed-point characterisations of *tp*, *ett*, albeit those for pushdown automata [110]. PreMo can generate the corresponding systems of equations and features optimised numerical algorithms for computing least fixed points, such as decomposed variants of the Newton's method [199]. In the future, we plan to take advantage of the numerical engine of PreMo to provide state-of-the-art support for iteration in our tool. A related recent development is PRAY [139], which can produce certificates for validating bounds on quantitative properties of probabilistic pushdown automata.

As concerns higher-order recursion schemes, the authors of [84] proposed and implemented a method for calculating sound lower and upper bounds for *tp* for probabilistic second-order schemes (while showing that the AST problem for such schemes is undecidable). As already mentioned, our setting (PAHORS) is incomparable with such schemes. However, most examples from [84] can still be expressed as PAHORS, and we report on them in the paper. Note that our tool can solve both AST and PAST for PAHORS, and approximate both *tp* and *ett* to arbitrary precision through bisection.

The AST and PAST problems have been actively researched for probabilistic imperative programs with possibly continuous distributions, drawing upon techniques from linear programming and martingale theory. Numerous tools have been developed in this strand, mainly for certifying AST or PAST, e.g. MGen [200], LexRSM [201], Absynth [202], KoAT [203], eco-imp [204], AMBER [205]. The last one is capable of both certifying and disproving (P)AST. In general, due to support for unbounded and continuous distributions, this setting is orthogonal to ours. Nevertheless, some integer-based examples with discrete distributions considered in the literature could in principle be handled by ad hoc translations to our setting using the tree stack to represent counters. Then, so long as the use of counters yields a restriction bound for the tree stack, PTSV could analyse them.

Every new beginning comes from some other beginning's end.

— Lucius Annaeus Seneca

10 Conclusion

We presented two equivalent probabilistic models with excessive expressiveness compared to the probabilistic recursive models (Recursive Markov Chains / Probabilistic Pushdown Automata), namely, Restricted Probabilistic Tree-Stack Automata (rPTSA) and Probabilistic Additive Higher-Order Recursion Schemes (PAHORS). These models address the undecidable challenges posed by the general PHORS hierarchy.

For the models, we demonstrated that the almost sure termination (AST) problem, and more generally the positive AST (PAST) problem for these models, are in EXPSPACE. Moreover, we investigated the model checking problems, including both linear-time and branching-time properties for these models, and provided constructive proofs of their decidability. To the best of our knowledge, we have identified the first class of non-trivial extensions to the probabilistic recursive models with a decidable AST problem, and we are the first to explore the expectation-based properties and model checking problems.

Additionally, we examined the restriction property of rPTSA, which is crucial for the decidability results. We established that although determining whether a given PTSA is restricted is undecidable, testing whether a PTSA is k-restricted for a given $k > 0$ is decidable.

Beyond the theoretical contributions, we implemented decision procedures for termination probability and expected termination time for both models in the tool PTSV. We validated the tool on a wide range of examples from the literature, confirming its effectiveness and efficiency despite the exponential complexity of the decision procedures.

In conclusion, we believe these results push the boundaries between decidable probabilistic models (ours and the recursive ones) and undecidable ones (PHORS), and hence also help delve deeper into understanding the world of infinite state Markov chains.

Future work could further explore: (1) the complexity lower bounds of the

decision procedures; (2) the possibilities of efficiently implementing the model checking procedures; and, (3) positioning more precisely the expressiveness of rPTSA and PAHORS in terms of the PHORS hierarchy – currently, we only know that PAHORS is more expressive than order-1 PHORS and is incomparable with order-2, but whether PAHORS is subsumed by order-3 is still open.

The first kind of intellectual and artistic personality belongs to the hedgehogs, the second to the foxes ...

— Sir Isaiah Berlin [206]

References

[1] Blaise Pascal. *pensées*. Dezobry et E. Magdeleine, 1852.

[2] Richard Von Mises. *Probability, statistics, and truth*. Courier Corporation, 1981.

[3] Peixin Wang et al. "Proving expected sensitivity of probabilistic programs with randomized variable-dependent termination time". In: *Proceedings of the ACM on Programming Languages* 4.POPL (2019), pp. 1–30.

[4] Peixin Wang et al. "Cost analysis of nondeterministic probabilistic programs". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 204–220.

[5] Karel De Leeuw et al. "Computability by probabilistic machines". In: *Automata studies* 34 (1956), pp. 183–198.

[6] SHAFI GOLDWASSER and SILVIO MICALI. "Probabilistic Encryption". In: *JOURNAL OF COMPUTER AND SYSTEM SCIENCES* 28 (1984), pp. 270–299.

[7] Ian Taylor. *Alan M. Turing: The Applications of Probability to Cryptography*. 2015. arXiv: `1505.04714 [math.HO]`.

[8] Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press, 2017.

[9] Rajeev Motwani and Prabhakar Raghavan. "Randomized algorithms". In: *ACM Computing Surveys (CSUR)* 28.1 (1996), pp. 33–37.

[10] M. O. Rabin. "Probabilistic algorithms". In: *Algorithms and complexity: new directions and results*. Academic Press, New York, 1976, pp. 21–39.

[11] Kurtland Chua et al. "Deep reinforcement learning in a handful of trials using probabilistic dynamics models". In: *Advances in neural information processing systems* 31 (2018).

[12] Yunpeng Pan, George I Boutselis, and Evangelos A Theodorou. "Efficient reinforcement learning via probabilistic trajectory optimization". In: *IEEE transactions on neural networks and learning systems* 29.11 (2018), pp. 5459–5474.

[13] Rushikesh Kamalapurkar, Joel A Rosenfeld, and Warren E Dixon. "Efficient model-based reinforcement learning for approximate online optimal control". In: *Automatica* 74 (2016), pp. 247–258.

[14] Juan Camilo Gamboa Higuera, David Meger, and Gregory Dudek. "Synthesizing neural network controllers with probabilistic model-based reinforcement learning". In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2018, pp. 2538–2544.

[15] Muffy Calder, Stephen Gilmore, and Jane Hillston. "Modelling the influence of RKIP on the ERK signalling pathway using the stochastic process algebra PEPA". In: *Transactions on computational systems biology VII*. Springer, 2006, pp. 1–23.

[16] Muffy Calder et al. "Analysis of signalling pathways using continuous time Markov chains". In: *Transactions on Computational Systems Biology VI*. Springer. 2006, pp. 44–67.

[17] M. Kwiatkowska, G. Norman, and D. Parker. "Using probabilistic model checking in systems biology". In: *ACM SIGMETRICS Performance Evaluation Review* 35.4 (2008), pp. 14–21.

[18] Matthew R Lakin et al. "Design and analysis of DNA strand displacement devices using probabilistic model checking". In: *Journal of the Royal Society Interface* 9.72 (2012), pp. 1470–1485.

[19] Frits Dannenberg et al. "DNA walker circuits: Computational potential, design, and verification". In: *DNA Computing and Molecular Programming: 19th International Conference, DNA 19, Tempe, AZ, USA, September 22-27, 2013. Proceedings 19*. Springer. 2013, pp. 31–45.

[20] Pietro Lio, Emanuela Merelli, and Nicola Paoletti. "Multiple verification in computational modeling of bone pathologies". In: *arXiv preprint arXiv:1109.1368* (2011).

[21] Gethin Norman et al. "Formal analysis and validation of continuous-time markov chain based system level power management strategies". In: *Seventh IEEE International High-Level Design Validation and Test Workshop, 2002.* IEEE. 2002, pp. 45–50.

[22] G. Norman et al. "Using Probabilistic Model Checking for Dynamic Power Management". In: *Proc. 3rd Workshop on Automated Verification of Critical Systems (AVoCS'03)*. Ed. by M. Leuschel, S. Gruner, and S. Lo Presti. Technical Report DSSE-TR-2003-2, University of Southampton. Apr. 2003, pp. 202–215.

[23] A. Susu et al. "Stochastic Modeling and Analysis for Environmentally Powered Wireless Sensor Nodes". In: *Proc. 6th International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOPT'08)*. 2008, pp. 11–20.

[24] Carol Mak, Fabian Zaiser, and Luke Ong. "Nonparametric Involutive Markov Chain Monte Carlo". In: *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*. Ed. by Kamalika Chaudhuri et al. Vol. 162. Proceedings of Machine Learning Research. PMLR, 2022, pp. 14802–14859. URL: `https://proceedings.mlr.press/v162/mak22a.html`.

[25] Sam Staton et al. "Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints". In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*. 2016, pp. 525–534.

[26] Raven Beutner, C.-H. Luke Ong, and Fabian Zaiser. "Guaranteed bounds for posterior inference in universal probabilistic programming". In: *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*. Ed. by Ranjit Jhala and Isil Dillig. ACM, 2022, pp. 536–551. URL: https://doi.org/10.1145/3519939.3523721.

[27] Fabian Zaiser, Andrzej S Murawski, and Luke Ong. "Exact Bayesian Inference on Discrete Models via Probability Generating Functions: A Probabilistic Programming Approach". In: *Thirty-seventh Conference on Neural Information Processing Systems*. 2023. URL: https://openreview.net/forum?id=FtNruwFEs3.

[28] Carol Mak et al. "Densities of Almost Surely Terminating Probabilistic Programs are Differentiable Almost Everywhere". In: *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*. Ed. by Nobuko Yoshida. Vol. 12648. Lecture Notes in Computer Science. Springer, 2021, pp. 432–461.

[29] Carol Mak, Fabian Zaiser, and Luke Ong. "Nonparametric Hamiltonian Monte Carlo". In: *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, 2021, pp. 7336–7347. URL: http://proceedings.mlr.press/v139/mak21a.html.

[30] Sam Staton. "Commutative semantics for probabilistic programming". In: *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings 26*. Springer. 2017, pp. 855–879.

[31] Noah D. Goodman et al. "Church: a language for generative models". In: *UAI 2008, Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, Helsinki, Finland, July 9-12, 2008*. AUAI Press, 2008, pp. 220–229.

[32] David Tolpin, Jan-Willem van de Meent, and Frank D. Wood. "Probabilistic Programming in Anglican". In: *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2015, Porto, Portugal, September 7-11, 2015, Proceedings, Part III*. Vol. 9286. Lecture Notes in Computer Science. Springer, 2015, pp. 308–311.

[33] Marco F. Cusumano-Towner et al. "Gen: a general-purpose probabilistic programming system with programmable inference". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. ACM, 2019, pp. 221–236.

[34] Eli Bingham et al. "Pyro: Deep Universal Probabilistic Programming". In: *J. Mach. Learn. Res.* 20 (2019), 28:1–28:6.

[35] Dustin Tran et al. "Edward: A library for probabilistic modeling, inference, and criticism". In: *CoRR* abs/1610.09787 (2016).

[36] Hong Ge, Kai Xu, and Zoubin Ghahramani. "Turing: A Language for Flexible Probabilistic Inference". In: *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*. Vol. 84. Proceedings of Machine Learning Research. PMLR, Sept. 2018, pp. 1682–1690.

[37] Adam Fabio. *KILLED BY A MACHINE: THE THERAC-25*. Web Page. 2015. URL: `https://hackaday.com/2015/10/26/killed-by-a-machine-the-therac-25/`.

[38] *N° 33-1996: Ariane 501 - Presentation of Inquiry Board report*. Web Page. URL: `https://www.esa.int/Newsroom/Press_Releases/Ariane_501_-_Presentation_of_Inquiry_Board_report`.

[39] Michael Carbin, Sasa Misailovic, and Martin C Rinard. "Verifying quantitative reliability for programs that execute on unreliable hardware". In: *ACM SIGPLAN Notices* 48.10 (2013), pp. 33–52.

[40] Steen Andreassen et al. "Using probabilistic and decision–theoretic methods in treatment and prognosis modeling". In: *Artificial Intelligence in medicine* 15.2 (1999), pp. 121–134.

[41] Jonathan G Richens, Ciarán M Lee, and Saurabh Johri. "Improving the accuracy of medical diagnosis with causal machine learning". In: *Nature communications* 11.1 (2020), p. 3923.

[42] Peixin Wang et al. "Static Posterior Inference of Bayesian Probabilistic Programming via Polynomial Solving". In: *Proceedings of the ACM on Programming Languages* 8.PLDI (2024), pp. 1361–1386.

[43] Koen Claessen and John Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP '00. New York, NY, USA: Association for Computing Machinery, 2000, pp. 268–279. URL: `https://doi.org/10.1145/351240.351266`.

[44] Agustín Mista, Alejandro Russo, and John Hughes. "Branching processes for quickcheck generators". In: *ACM SIGPLAN Notices* 53.7 (2018). Publisher: ACM New York, NY, USA, pp. 1–13.

[45] Benjamin Lucien Kaminski and Joost-Pieter Katoen. "On the hardness of almost–sure termination". In: *International Symposium on Mathematical Foundations of Computer Science*. Springer. 2015, pp. 307–318.

[46] Stephen Cole Kleene. "Recursive predicates and quantifiers". In: *Transactions of the American Mathematical Society* 53.1 (1943), pp. 41–73.

[47] Piergiorgio Odifreddi. *Classical recursion theory: The theory of functions and sets of natural numbers*. Elsevier, 1992.

[48] Holger Hermanns, Björn Wachter, and Lijun Zhang. "Probabilistic cegar". In: *International Conference on Computer Aided Verification*. Springer. 2008, pp. 162–175.

[49] Patrick Cousot and Michael Monerau. "Probabilistic abstract interpretation". In: *European Symposium on Programming*. Springer. 2012, pp. 169–193.

[50] Raven Beutner and Luke Ong. "On Probabilistic Termination of Functional Programs with Continuous Distributions". In: *Proceedings of the 42st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2021*. 2021.

[51] Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. "Operational versus weakest pre-expectation semantics for the probabilistic guarded command language". In: *Performance Evaluation* 73 (2014), pp. 110–132.

[52] Calvin Smith, Justin Hsu, and Aws Albarghouthi. "Trace abstraction modulo probability". In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–31.

[53] Di Wang, Jan Hoffmann, and Thomas Reps. "PMAF: an algebraic framework for static analysis of probabilistic programs". In: *ACM SIGPLAN Notices* 53.4 (2018), pp. 513–528.

[54] Jinyi Wang et al. "Quantitative analysis of assertion violations in probabilistic programs". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, pp. 1171–1186.

[55] Edmund Clarke et al. "Counterexample-guided abstraction refinement". In: *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings 12*. Springer. 2000, pp. 154–169.

[56] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. "Refinement of trace abstraction". In: *International Static Analysis Symposium*. Springer. 2009, pp. 69–85.

[57] Bat-Chen Rothenberg, Daniel Dietsch, and Matthias Heizmann. "Incremental verification using trace abstraction". In: *Static Analysis: 25th International Symposium, SAS 2018, Freiburg, Germany, August 29–31, 2018, Proceedings 25*. Springer. 2018, pp. 364–382.

[58] Gilles Barthe et al. "An assertion-based program logic for probabilistic programs". In: *European Symposium on Programming*. Springer, Cham. 2018, pp. 117–144.

[59] Jianhui Chen and Fei He. "Proving almost-sure termination by omega-regular decomposition". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2020, pp. 869–882.

[60] Alessandra Di Pierro and Herbert Wiklicky. "Probabilistic Abstract Interpretation: From Trace Semantics to DTMC's and Linear Regression". In: *Semantics, Logics, and Calculi*. Springer, 2016, pp. 111–139.

[61] Annabelle McIver, Carroll Morgan, and Charles Carroll Morgan. *Abstraction, refinement and proof for probabilistic systems*. Springer Science & Business Media, 2005.

[62] S. Abramsky and C. L. Hankin, eds. *Abstract Interpretation for Declarative Languages*. Ellis Horwood, 1987.

[63] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Proceedings of POPL'77*. 1991, pp. 238–252.

[64] Benjamin C Pierce et al. “Software foundations”. In: *Webpage: http://www. cis. upenn. edu/bcpierce/sf/current/index. html* (2010).

[65] Timon Gehr, Sasa Misailovic, and Martin Vechev. “PSI: Exact symbolic inference for probabilistic programs”. In: *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I 28*. Springer. 2016, pp. 62–83.

[66] Jacques Carette and Chung-chieh Shan. “Simplifying probabilistic programs using computer algebra”. In: *Practical Aspects of Declarative Languages: 18th International Symposium, PADL 2016, St. Petersburg, FL, USA, January 18-19, 2016. Proceedings 18*. Springer. 2016, pp. 135–152.

[67] Praveen Narayanan et al. “Probabilistic inference by program transformation in Hakaru (system description)”. In: *Functional and Logic Programming: 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings 13*. Springer. 2016, pp. 62–79.

[68] Marcel Moosbrugger et al. “The probabilistic termination tool amber”. In: *Formal Methods in System Design* 61.1 (2022), pp. 90–109.

[69] Wang Yu, Gerald B Sheblé, and Manuel António Matos. “Application of Markov chain models for short-term generation assets valuation”. In: *Probability in the Engineering and Informational Sciences* 20.1 (2006), pp. 127–141.

[70] M. Kwiatkowska, G. Norman, and D. Parker. “Probabilistic Verification of Herman’s Self-Stabilisation Algorithm”. In: *Formal Aspects of Computing* 24.4 (2012), pp. 661–670.

[71] M. Kwiatkowska, G. Norman, and R. Segala. “Automated Verification of a Randomized Distributed Consensus Protocol Using Cadence SMV and PRISM”. In: *Proc. 13th International Conference on Computer Aided Verification (CAV’01)*. Ed. by G. Berry, H. Comon, and A. Finkel. Vol. 2102. LNCS. Springer, 2001, pp. 194–206.

[72] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[73] M. Kwiatkowska, G. Norman, and D. Parker. “PRISM 4.0: Verification of Probabilistic Real-time Systems”. In: *Proc. 23rd International Conference on Computer Aided Verification (CAV’11)*. Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. LNCS. Springer, 2011, pp. 585–591.

[74] M. Kwiatkowska, G. Norman, and D. Parker. “Probabilistic Symbolic Model Checking with PRISM: A Hybrid Approach”. In: *International Journal on Software Tools for Technology Transfer (STTT)* 6.2 (2004), pp. 128–142.

[75] D. Parker. “Implementation of Symbolic Model Checking for Probabilistic Systems”. University of Birmingham, 2002.

[76] Pavol Cerny, Thomas A Henzinger, and Arjun Radhakrishna. “Quantitative abstraction refinement”. In: *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2013, pp. 115–128.

[77] Christian Hensel et al. “The probabilistic model checker Storm”. In: *International Journal on Software Tools for Technology Transfer* (2021), pp. 1–22.

[78] Sebastian Junges et al. “The complexity of reachability in parametric Markov decision processes”. In: *Journal of Computer and System Sciences* 119 (2021), pp. 183–210.

[79] Sean R Eddy. “What is a hidden Markov model?” In: *Nature biotechnology* 22.10 (2004), pp. 1315–1316.

[80] Sean R Eddy. “Hidden markov models”. In: *Current opinion in structural biology* 6.3 (1996), pp. 361–365.

[81] Karl Johan Åström. “Optimal control of Markov processes with incomplete state information I”. In: *Journal of mathematical analysis and applications* 10 (1965), pp. 174–205.

[82] Gregory F Lawler and Vlada Limic. *Random walk: a modern introduction*. Vol. 123. Cambridge University Press, 2010.

[83] Edward A Codling, Michael J Plank, and Simon Benhamou. “Random walk models in biology”. In: *Journal of the Royal society interface* 5.25 (2008), pp. 813–834.

[84] Charles Grellois, Ugo Dal Lago, and Naoki Kobayashi. “On the Termination Problem for Probabilistic Higher-Order Recursive Programs”. In: *Logical Methods in Computer Science* 16 (2020).

[85] Naoki Kobayashi, Ugo Dal Lago, and Charles Grellois. “On the termination problem for probabilistic higher-order recursive programs”. In: *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE. 2019, pp. 1–14.

[86] Kousha Etessami and Mihalis Yannakakis. “Recursive Markov chains, stochastic grammars, and monotone systems of nonlinear equations”. In: *Annual Symposium on Theoretical Aspects of Computer Science*. Springer. 2005, pp. 340–352.

[87] Javier Esparza, Antonin Kucera, and Richard Mayr. “Model checking probabilistic pushdown automata”. In: *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004.* IEEE. 2004, pp. 12–21.

[88] C.-H. L. Ong. “On Model-Checking Trees Generated by Higher-Order Recursion Schemes”. In: *Proceedings of LICS*. Computer Society Press, 2006, pp. 81–90.

[89] N. Kobayashi and C.-H. L. Ong. “Complexity of Model Checking Recursion Schemes for Fragments of the Modal Mu-Calculus”. In: *Logical Methods in Computer Science* 7.4 (2011).

[90] Luke Ong. “Higher-order model checking: An overview”. In: *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science*. IEEE. 2015, pp. 1–15.

[91] Naoki Kobayashi. “Higher-order model checking: From theory to practice”. In: *2011 IEEE 26th Annual Symposium on Logic in Computer Science*. IEEE. 2011, pp. 219–224.

[92] Alfred V Aho. “Indexed grammars—an extension of context-free grammars”. In: *Journal of the ACM (JACM)* 15.4 (1968), pp. 647–671.

[93] Naoki Kobayashi. “Types and higher-order recursion schemes for verification of higher-order programs”. In: *Proceedings of POPL*. 2009, pp. 416–428.

[94] Christopher H. Broadbent and Naoki Kobayashi. "Saturation-Based Model Checking of Higher-Order Recursion Schemes". In: *Computer Science Logic 2013 (CSL 2013), CSL 2013, September 2-5, 2013, Torino, Italy*. Ed. by Simona Ronchi Della Rocca. Vol. 23. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013, pp. 129–148. URL: https://doi.org/10.4230/LIPIcs.CSL.2013.129.

[95] Steven J Ramsay, Robin P Neatherway, and C-H Luke Ong. "A type-directed abstraction refinement approach to higher-order model checking". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2014, pp. 61–72.

[96] Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. "Predicate abstraction and CEGAR for higher-order model checking". In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 2011, pp. 222–233.

[97] Hugo Paquet and Sam Staton. "LazyPPL: laziness and types in non-parametric probabilistic programs". In: *Advances in Programming Languages and Neurosymbolic Systems Workshop*. 2021.

[98] Noah D Goodman and Andreas Stuhlmüller. *The design and implementation of probabilistic programming languages*. 2014.

[99] Tobias Denkinger. "An automata characterisation for multiple context-free languages". In: *International Conference on Developments in Language Theory*. Springer. 2016, pp. 138–150.

[100] Hiroyuki Seki et al. "On multiple context-free grammars". In: *Theoretical Computer Science* 88.2 (1991), pp. 191–229. URL: https://www.sciencedirect.com/science/article/pii/030439759190374B.

[101] Alexander Clark. "An introduction to multiple context free grammars for linguists". en. In: (), p. 25.

[102] Pierre Clairambault and Andrzej Murawski. "On the expressivity of linear recursion schemes". In: (2019).

[103] J.-Y. Girard. "Linear Logic". In: *Theoretical Computer Science* 50 (1987), pp. 1–102.

[104] Steven Abney, David McAllester, and Fernando Pereira. "Relating probabilistic grammars and automata". In: *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics*. 1999, pp. 542–549.

[105] Tomáš Brázdil, Stefan Kiefer, and Antonín Kučera. "Efficient analysis of probabilistic programs with an unbounded counter". In: *International Conference on Computer Aided Verification*. Springer. 2011, pp. 208–224.

[106] Kousha Etessami and Mihalis Yannakakis. "Algorithmic verification of recursive probabilistic state machines". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2005, pp. 253–270.

[107] Rajeev Alur et al. "Analysis of recursive state machines". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 27.4 (2005), pp. 786–818.

[108] Michael Benedikt, Patrice Godefroid, and Thomas Reps. "Model checking of unrestricted hierarchical state machines". In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2001, pp. 652–666.

[109] Kousha Etessami and Mihalis Yannakakis. "Recursive Markov chains, stochastic grammars, and monotone systems of nonlinear equations". In: *Journal of the ACM (JACM)* 56.1 (2009), pp. 1–66.

[110] Tomás Brázdil et al. "Analyzing probabilistic pushdown automata". In: *Formal Methods Syst. Des.* 43.2 (2013), pp. 124–163.

[111] Kousha Etessami and Mihalis Yannakakis. "Model checking of recursive probabilistic systems". In: *ACM Transactions on Computational Logic (TOCL)* 13.2 (2012), pp. 1–40.

[112] Dejan Jovanović and Leonardo De Moura. "Solving non-linear arithmetic". In: *International Joint Conference on Automated Reasoning*. Springer. 2012, pp. 339–354.

[113] Javier Esparza, Antonín Kucera, and Richard Mayr. "Quantitative analysis of probabilistic pushdown automata: Expectations and variances". In: *20th Annual IEEE Symposium on Logic in Computer Science (LICS'05)*. IEEE. 2005, pp. 117–126.

[114] Tomáš Brázdil et al. "Runtime analysis of probabilistic programs with unbounded recursion". In: *Journal of Computer and System Sciences* 81.1 (2015), pp. 288–310.

[115] Mihalis Yannakakis and Kousha Etessami. "Checking LTL properties of recursive Markov chains". In: *Second International Conference on the Quantitative Evaluation of Systems (QEST'05)*. IEEE. 2005, pp. 155–164.

[116] Javier Esparza, Jan Křetínskỳ, and Salomon Sickert. "One theorem to rule them all: A unified translation of LTL into ω-automata". In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. 2018, pp. 384–393.

[117] Tomáš Brázdil, Antonín Kučera, and Oldřich Stražovskỳ. "On the decidability of temporal properties of probabilistic pushdown automata". In: *Annual Symposium on Theoretical Aspects of Computer Science*. Springer. 2005, pp. 145–157.

[118] Tomáš Brázdil. "Verification of probabilistic recursive sequential programs". PhD thesis. Masarykova univerzita, Fakulta informatiky, 2007.

[119] Tomáš Brázdil, Václav Brožek, and Vojtěch Forejt. "Branching-Time Model-Checking of Probabilistic Pushdown Automata". In: *Electronic Notes in Theoretical Computer Science* 239 (2009). Joint Proceedings of the 8th, 9th, and 10th International Workshops on Verification of Infinite-State Systems (INFINITY 2006, 2007, 2008), pp. 73–83. URL: `https://www.sciencedirect.com/science/article/pii/S1571066109001534`.

[120] Tomáš Brázdil et al. "Branching-time model-checking of probabilistic pushdown automata". In: *Journal of Computer and System Sciences* 80.1 (2014), pp. 139–156.

[121] Berndt Farwer. "ω-automata". In: *Automata logics, and infinite games*. Springer, 2002, pp. 3–21.

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

[123] J Richard Büchi. “On a decision method in restricted second order arithmetic”. In: *The collected works of J. Richard Büchi*. Springer, 1990, pp. 425–435.

[124] David E Muller. “Infinite sequences and finite machines”. In: *Proceedings of the Fourth Annual Symposium on Switching Circuit Theory and Logical Design (swct 1963)*. IEEE Computer Society. 1963, pp. 3–16.

[125] Tomáš Babiak et al. “Effective translation of LTL to deterministic Rabin automata: Beyond the (F, G)-fragment”. In: *Automated Technology for Verification and Analysis*. Springer, 2013, pp. 24–39.

[126] František Blahoudek, Mojmír Křetínskỳ, and Jan Strejček. “Comparison of LTL to deterministic Rabin automata translators”. In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer. 2013, pp. 164–172.

[127] Kousha Etessami, Dominik Wojtczak, and Mihalis Yannakakis. “Quasi-birth–death processes, tree-like QBDs, probabilistic 1-counter automata, and pushdown systems”. In: *Performance Evaluation* 67.9 (2010), pp. 837–857.

[128] Ji Zhang and Edward J Coyle. “Transient analysis of quasi-birth-death processes”. In: *Stochastic Models* 5.3 (1989), pp. 459–496.

[129] Rafael Pérez-Ocón and Delia Montoro-Cazorla. “A multiple system governed by a quasi-birth-and-death process”. In: *Reliability Engineering & System Safety* 84.2 (2004), pp. 187–196.

[130] Raymond W Yeung and Bhaskar Sengupta. “Matrix product-form solutions for Markov chains with a tree structure”. In: *Advances in Applied Probability* 26.4 (1994), pp. 965–987.

[131] Tetsuya Takine, Bhaskar Sengupta, and Raymond W Yeung. “A generalization of the matrix M/G/1 paradigm for Markov chains with a tree structure”. In: *Stochastic Models* 11.3 (1995), pp. 411–421.

[132] B Van Houdt and C Blondia. “Tree Structured QBD Markov Chains and Tree-Like QBD Processes”. In: (2003).

[133] Jos CM Baeten and Willem Paul Weijland. *Process algebra*. Cambridge university press, 1991.

[134] Suzana Andova. “Process algebra with probabilistic choice”. In: *International AMAST Workshop on Aspects of Real-Time Systems and Concurrent and Distributed Software*. Springer. 1999, pp. 111–129.

[135] Suzana Andova. “Probabilistic process algebra”. In: (2002).

[136] Kousha Etessami and Mihalis Yannakakis. “Recursive Markov decision processes and recursive stochastic games”. In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2005, pp. 891–903.

[137] Kousha Etessami, Dominik Wojtczak, and Mihalis Yannakakis. "Recursive stochastic games with positive rewards". In: *Automata, Languages and Programming: 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I 35*. Springer. 2008, pp. 711–723.

[138] Stefan Kiefer, Michael Luttenberger, and Javier Esparza. "On the convergence of Newton's method for monotone systems of polynomial equations". In: *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*. 2007, pp. 217–226.

[139] Tobias Winkler and Joost-Pieter Katoen. "Certificates for Probabilistic Pushdown Automata via Optimistic Value Iteration". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2023, pp. 391–409.

[140] Dominik Wojtczak and Kousha Etessami. "Premo: an analyzer for probabilistic recursive models". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2007, pp. 66–71.

[141] C. H.L. Ong. "On model-checking trees generated by higher-order recursion schemes". In: *Proceedings - Symposium on Logic in Computer Science*. 2006, pp. 81–90.

[142] P. Clairambault and A. S. Murawski. "On the expressivity of linear recursion schemes". In: *Proceedings of 44th International Symposium on Mathematical Foundations of Computer Science (MFCS)*. Vol. 138. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019, 50:1–50:14.

[143] Luke Ong. "Higher-order model checking: An overview". In: *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science*. IEEE. 2015, pp. 1–15.

[144] IUriĭ V. Matiiasevich et al. *Hilbert's Tenth Problem*. Google-Books-ID: 42h5VRwG5rcC. MIT Press, 1993. 296 pp.

[145] Noam Chomsky. *Syntactic structures*. 1956.

[146] Noam Chomsky. "ASPECTS OF THE THEORY OF SYNTAX". In: (1965).

[147] Noam Chomsky. "On certain formal properties of grammars". In: *Information and control* 2.2 (1959), pp. 137–167.

[148] Geoffrey K Pullum and Gerald Gazdar. "Natural languages and context-free languages". In: *Linguistics and Philosophy* 4.4 (1982), pp. 471–504.

[149] Aravind K Joshi, K Vijay Shanker, and David Weir. "The convergence of mildly context-sensitive grammar formalisms". In: *Technical Reports (CIS)* (1990), p. 539.

[150] P Stanley Peters and Robert W Ritchie. "A note on the universal base hypothesis". In: *Journal of linguistics* 5.1 (1969), pp. 150–152.

[151] David Jeremy Weir. "Characterizing mildly context-sensitive grammar formalisms". PhD thesis. University of Pennsylvania, 1988.

[152] Aravind Krishna Joshi. "Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions?" In: (1985).

[153] *Mildly context-sensitive grammar formalism*. In: *Wikipedia*. Page Version ID: 1029931978. June 22, 2021. URL: https://en.wikipedia.org/w/index.php?title=Mildly_context-sensitive_grammar_formalism&oldid=1029931978 (visited on 04/12/2022).

[154] Makoto Kanazawa. "Lecture 4: Tree-Adjoining Grammars and Related Formalisms". In: 2016. URL: https://makotokanazawa.ws.hosei.ac.jp/FormalGrammar/lecture4.pdf.

[155] Carl Jesse Pollard. "Generalized phrase structure grammars, head grammars, and natural language". In: *Ph. D. disertation, Stanford University* (1984).

[156] Eric Reuland. "An introduction to minimalist grammar". In: (). URL: https://www.academia.edu/2749802/An_introduction_to_minimalist_grammar (visited on 03/24/2022).

[157] Gerald Gazdar. "Applicability of indexed grammars to natural languages". In: *Natural language parsing and linguistic theories*. Springer, 1988, pp. 69–94.

[158] Makoto Kanazawa. "Lecture 5: Mildly Context-Sensitive Languages". In: 2016. URL: https://makotokanazawa.ws.hosei.ac.jp/FormalGrammar/lecture5.pdf.

[159] Meng-Che Ho. "The word problem of Zn is a multiple context-free language". In: *Groups Complexity Cryptology* 10.1 (2018), pp. 9–15.

[160] Sylvain Salvati. "MIX is a 2-MCFL and the word problem in $$\backslash$mathbb ${$Z$}$^ 2$ is solved by a third-order collapsible pushdown automaton". In: *Journal of Computer and System Sciences* 81.7 (2015), pp. 1252–1277.

[161] Mark-Jan Nederhof. "A short proof that $ O_2 $ is an MCFL". In: *arXiv preprint arXiv:1603.03610* (2016).

[162] Mark-Jan Nederhof. *Chapter 28: Free word order and MCFLs*. 2017. URL: https://mjn.host.cs.st-andrews.ac.uk/publications/2017a.pdf (visited on 04/05/2022).

[163] Sylvain Salvati. *Multiple Context-free Grammars - Course 5: MIX is a 2-MCFL*. en.

[164] *syntax - What is word order used for in "free word order" languages?* Linguistics Stack Exchange. URL: https://linguistics.stackexchange.com/questions/1476/what-is-word-order-used-for-in-free-word-order-languages (visited on 04/06/2022).

[165] *Word order*. In: *Wikipedia*. Page Version ID: 1080605114. Apr. 2, 2022. URL: https://en.wikipedia.org/w/index.php?title=Word_order&oldid=1080605114 (visited on 04/06/2022).

[166] *Latin word order*. In: *Wikipedia*. Page Version ID: 1066861811. Jan. 20, 2022. URL: https://en.wikipedia.org/w/index.php?title=Latin_word_order&oldid=1066861811 (visited on 04/06/2022).

[167] Kilian Gebhardt, Frédéric Meunier, and Sylvain Salvati. "O_n is an n-MCFL". In: *arXiv preprint arXiv:2012.12100* (2020).

[168] Michel Latteux. “Cônes rationnels commutatifs”. In: *Journal of Computer and System Sciences* 18.3 (1979), pp. 307–333.

[169] Makoto Kanazawa and Sylvain Salvati. “The copying power of well-nested multiple context-free grammars”. In: *International Conference on Language and Automata Theory and Applications*. Springer, 2010, pp. 344–355.

[170] Makoto Kanazawa. “Second-Order Abstract Categorial Grammars as Hyperedge Replacement Grammars”. en. In: *Journal of Logic, Language and Information* 19.2 (Apr. 2010), pp. 137–161. URL: `http://link.springer.com/10.1007/s10849-009-9109-6` (visited on 03/22/2022).

[171] Giorgio Satta and Marco Kuhlmann. “Efficient parsing for head-split dependency trees”. In: *Transactions of the Association for Computational Linguistics* 1 (2013). Publisher: MIT Press, pp. 267–278.

[172] Kelly Roach. “Formal properties of head grammars”. In: *Mathematics of Language* (1987), pp. 293–348.

[173] Mark Steedman. *The syntactic process*. MIT press, 2001.

[174] Frank Drewes, Hans-Joerg Kreowski, and Annegret Habel. “Hyperedge replacement graph grammars”. In: Feb. 1997, pp. 95–162.

[175] David Weir. “Linear context-free rewriting systems and deterministic tree-walking transducers”. In: *30th Annual Meeting of the Association for Computational Linguistics*. 1992, pp. 136–143.

[176] Sylvain Salvati. “Encoding Second-Order String ACGs with Deterministic Tree Walking Transducers”. en. In: (), p. 86.

[177] Sylvain Salvati. “Encoding second order string ACG with deterministic tree walking transducers”. In: *Proceedings of FG* (2006), pp. 143–156.

[178] Niko Paltzer. “Tree Transducers”. In: ().

[179] Owen Rambow and Giorgio Satta. “Independent parallelism in finite copying parallel rewriting systems”. In: *Theoretical Computer Science* 223.1-2 (1999). Publisher: Elsevier, pp. 87–120.

[180] Tadao Kasami. “Generalized context-free grammars, multiple context-free grammars and head grammars”. In: *Preprint of WG on Natural Language of IPSJ* (1987).

[181] J.-Y. Girard, A. Scedrov, and P. J. Scott. “Bounded linear logic”. In: *Theoretical Computer Science* 97 (1992), pp. 1–66.

[182] Steve Klabnik and Carol Nichols. *The Rust programming language*. No Starch Press, 2023.

[183] Jean-Philippe Bernardy et al. “Linear Haskell: practical linearity in a higher-order polymorphic language”. In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), pp. 1–29.

[184] Edwin Brady. “Idris 2: Quantitative type theory in practice”. In: *arXiv preprint arXiv:2104.00480* (2021).

[185] P. Clairambault, C. Grellois, and A. S. Murawski. "Linearity in higher-order recursion schemes". In: *Proceedings of the ACM on Programming Languages* 2.POPL (2018), 39:1–39:29.

[186] Achim Klenke. *Probability theory: a comprehensive course*. Springer Science & Business Media, 2013.

[187] Guanyan Li, Andrzej Murawski, and Luke Ong. "Probabilistic Verification Beyond Context-Freeness". In: *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science*. 2022, pp. 1–13.

[188] Shmuel Safra and Moshe Y Vardi. "On ω-automata and temporal logic". In: *Proceedings of the twenty-first annual ACM symposium on Theory of computing*. 1989, pp. 127–137.

[189] Sven Schewe and Thomas Varghese. "Determinising parity automata". In: *International Symposium on Mathematical Foundations of Computer Science*. Springer. 2014, pp. 486–498.

[190] Hubert Comon et al. *Tree automata techniques and applications*. 2008.

[191] Niki Vazou. *Liquid Haskell: Haskell as a theorem prover*. University of California, San Diego, 2016.

[192] G. Barthe et al. "Higher-Order Approximate Relational Refinement Types for Mechanism Design and Differential Privacy". In: *Proceedings of POPL'15*. 2015, pp. 55–68.

[193] Dan Romik. "Stirling's approximation for n!: The ultimate short proof?" In: *The American Mathematical Monthly* 107.6 (2000), pp. 556–557.

[194] John Canny. "Some algebraic and geometric computations in PSPACE". In: *Proceedings of the twentieth annual ACM symposium on Theory of computing*. 1988, pp. 460–467.

[195] Henry Gordon Rice. "Classes of recursively enumerable sets and their decision problems". In: *Transactions of the American Mathematical society* 74.2 (1953), pp. 358–366.

[196] Naoki Kobayashi, Ugo Dal Lago, and Charles Grellois. "On the Termination Problem for Probabilistic Higher-Order Recursive Programs". In: *Log. Methods Comput. Sci.* 16.4 (2020). URL: `https://lmcs.episciences.org/6817`.

[197] Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. "On the hardness of analyzing probabilistic programs". In: *Acta Informatica* 56.3 (2019), pp. 255–285.

[198] Steven Abney, David McAllester, and Fernando Pereira. "Relating probabilistic grammars and automata". In: *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics*. 1999, pp. 542–549.

[199] Kousha Etessami and Mihalis Yannakakis. "Recursive Markov chains, stochastic grammars, and monotone systems of nonlinear equations". In: *J. ACM* 56.1 (2009), 1:1–1:66. URL: `https://doi.org/10.1145/1462153.1462154`.

[200] Aleksandar Chakarov and Sriram Sankaranarayanan. "Probabilistic Program Analysis with Martingales". In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings.* Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 511–526. URL: https://doi.org/10.1007/978-3-642-39799-8%5C_34.

[201] Sheshansh Agrawal, Krishnendu Chatterjee, and Petr Novotný. "Lexicographic ranking supermartingales: an efficient approach to termination of probabilistic programs". In: *Proc. ACM Program. Lang.* 2.POPL (2018), 34:1–34:32. URL: https://doi.org/10.1145/3158122.

[202] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. "Bounded expectations: resource analysis for probabilistic programs". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018.* Ed. by Jeffrey S. Foster and Dan Grossman. ACM, 2018, pp. 496–512. URL: https://doi.org/10.1145/3192366.3192394.

[203] Fabian Meyer, Marcel Hark, and Jürgen Giesl. "Inferring Expected Runtimes of Probabilistic Integer Programs Using Expected Sizes". In: *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part I.* Ed. by Jan Friso Groote and Kim Guldstrand Larsen. Vol. 12651. Lecture Notes in Computer Science. Springer, 2021, pp. 250–269. URL: https://doi.org/10.1007/978-3-030-72016-2%5C_14.

[204] Martin Avanzini, Georg Moser, and Michael Schaper. "A modular cost analysis for probabilistic programs". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 172:1–172:30. URL: https://doi.org/10.1145/3428240.

[205] Marcel Moosbrugger et al. "The Probabilistic Termination Tool Amber". In: *International Symposium on Formal Methods*. Springer. 2021, pp. 667–675.

[206] Isaiah Berlin. *The hedgehog and the fox: An essay on Tolstoy's view of history.* Princeton University Press, 2013.

Appendices

List of Figures

List of Tables

List of Names