Bachelor's Thesis Computing Science

Radboud University Nijmegen

Control Improvisation for Deterministic Finite Tree Automata

Author:
Christy Bongers
s1061420

First supervisor/assessor:
dr. Jurriaan Rot

Second assessor:
dr. Sebastian Junges

July 1, 2025

Abstract

In this thesis, we extend the Control Improvisation (CI) framework to tree-structured data using tree automata. CI is a method for generating random outputs that are both valid according to strict rules and diverse within specified probabilistic limits. The primary research question asks how CI can be adapted to generate random tree structures that satisfy its strict validity requirements while remaining within its probabilistic diversity limits. Using deterministic finite tree automata, we develop algorithms for feasible control improvisation, including uniform generation methods for both top-down and bottom-up tree automata. The results demonstrate that CI can be applied to tree structures, enabling the generation of diverse and valid tree-structured data.

Contents

Chapter 1

Introduction

In modern computational applications, systems often need to generate outputs that are not only valid according to strict rules but also diverse or unpredictable. This balancing act is critical in domains like software testing, robotic control, and music generation [11, 2]. Control Improvisation (CI) addresses this need by generating sequences (or 'words') that conform to predefined constraints while maintaining a degree of randomness. These sequences, generated by a probabilistic algorithm, meet hard constraints to ensure validity and soft constraints to incorporate variation and unpredictability [6].

While CI has been applied in areas like software testing, music generation, and robotic surveillance, most applications focus on words [5, 7]. This limitation raises the question of whether the CI framework can be extended to more complex structures, such as trees, which are used in many computational fields, such as in areas like natural language processing and tree-structured data representation.
This thesis explores extending CI to tree structures, using tree automata as a formal framework to manage structural and improvisational constraints. Tree automata are a generalization of finite automata, capable of operating on tree-like data structures instead of simple words. By extending CI to tree automata, we can handle more sophisticated data representations [4].

Control Improvisation, formally introduced by Fremont et al., is a framework for generating random but constrained outputs. The CI problem is typically formalized over regular languages, where the improviser generates words that meet both hard constraints (e.g., syntactic correctness) and soft constraints (e.g., randomness, diversity, or resemblance to a template) [6]. A key contribution of this framework is the guarantee of feasibility: the improviser must be able to produce an output whenever one is possible.
Tree automata generalize the CI framework from words to tree-structured

data, enabling validation and generation of tree-structured outputs. Tree automata have been extensively studied and are useful in various contexts such as XML document validation, semantic parsing, and verification of recursive programs [4, 3]. Unlike classical automata that process strings, tree automata operate over terms or trees, allowing for a more expressive handling of tree-structured structures.

Combining CI with tree automata introduces theoretical challenges, especially in ensuring feasibility under tree-based constraints.

The primary research question guiding this work is: *Can the Control Improvisation framework be extended to tree structures using tree automata?* This research seeks to investigate how CI can be adapted for generating random tree structures that satisfy both hard and soft constraints. Specifically, the objective is to develop an improviser capable of efficiently producing diverse, structured outputs that adhere to predefined rules while introducing a controlled level of randomness.
To address this question, we establish the following objectives:

1. Analyze the principles of Control Improvisation as applied to linear sequences, as described in [6], and identifying key challenges in extending these principles to tree structures.

2. Investigate the properties and capabilities of tree automata, focusing on their applicability to tree-structured data and their potential for integrating with CI.

3. Design and implement an improvisation algorithm based on tree automata that can generate random tree structures while respecting both hard constraints (valid tree forms) and soft constraints (diversity).

The remainder of this thesis is structured as follows:

- **Chapter 2: Preliminaries** introduces the concepts needed for the rest of the thesis. It reviews automata theory, including both deterministic and non-deterministic finite automata, and provides a detailed introduction to Control Improvisation. The chapter concludes with an overview of tree automata, including both top-down and bottom-up variants.

- **Chapter 3: Research** presents the original contributions of the thesis. It begins with a discussion of the restrictions imposed by deterministic finite tree automata, then develops formal definitions for Control Improvisation in the tree setting. Two improvisation algorithms are proposed: one for top-down deterministic tree automata and one for bottom-up variants. Their feasibility and correctness are discussed.

- **Chapter 4: Related Work** surveys existing literature related to automata theory, probabilistic generation, and applications of Control Improvisation.

- **Chapter 5: Conclusions** summarizes the main contributions, reflects on the findings, and outlines directions for future work.

Chapter 2

Preliminaries

In this chapter, we introduce the core concepts of Control Improvisation (CI), along with the necessary basics in automata theory used to apply it. We start by introducing DFAs and NFAs, which are needed to understand CI. These automata are the building blocks for the models we use to understand and apply CI.

The automata theory follows standard conventions as presented in [9]. For Control Improvisation, we adopt the definitions and results from [6], which form the basis of the theoretical framework and problem formulation.

2.1 Automata Theory

2.1.1 Basic Definitions and Notation

This section introduces the automata theory needed to understand and apply Control Improvisation.
We begin with basic notations,

- Σ: The alphabet for a given language is a finite set of symbols,
- $|A|$: The size of set A, *i.e.*, , the number of elements in A.
- $\mathcal{P}(A)$: The power set of A, *i.e.*, the set of all subsets of A.

To work with a language, we must first understand what its elements are, namely words defined over the alphabet.

Definition 1 (Words)**.** *A **word** over an alphabet Σ is a finite sequence of symbols from Σ. The set of all such words, including the empty word, is denoted by Σ^*.*

Examples for a word, over the alphabet $\Sigma = \{0, 1\}$, are '0101', '1' , '0000', etc.
Using these notations, we now define the notion of a language.

Definition 2 (Language)**.** *A* ***language*** *L over an alphabet Σ is any subset of Σ^*, i.e., $L \subseteq \Sigma^*$.*

We often need to refer not only to the length of a word but also to how frequent certain symbols appear within. And here the length of a word w is the number of symbols in w.

Definition 3 (Length of a word and symbol count)**.** *The* ***length*** *of a word w, written as $|w|$, is the amount of symbols in its sequence. The* ***number of occurrences*** *of a symbol 'a' from the alphabet Σ in a word w is denoted by $|w|_a$.*

Here we show a couple of examples:

- If $w = aab$, then $|w| = 3$ and $|w|_a = 2$.
- If $w = aabbb$, then $|w| = 5$ and $|w|_b = 3$.

We now introduce regular languages, a fundamental class of formal languages with limited but useful expressive power. These languages are precisely characterized by two equivalent formalisms: they can be described by regular expressions (patterns using concatenation, union, and repetition) and recognized by finite-state machines called deterministic or nondeterministic finite automata (DFAs/NFAs), which we will define in the next section.

2.1.2 Deterministic Finite Automata (DFA)

A Deterministic Finite Automaton (DFA) is a mathematical model used to define regular languages in a structured, finite way. The way to define this is:

Definition 4 (DFA)**.** *A* ***Deterministic Finite Automaton (DFA)*** *is represented as a tuple $D = (Q, \Sigma, \delta, q_0, F)$, where:*

- *Q is a finite set of states,*
- *Σ is a finite alphabet,*
- *$\delta : Q \times \Sigma \to Q$ is the transition function, which maps each state and symbol to a new state,*
- *$q_0 \in Q$ is the initial/starting state,*
- *$F \subseteq Q$ is the set of accepting/final states.*

Definition 5 (Acceptance of a word)**.** *A word $w \in \Sigma^*$ is* ***accepted*** *if $\delta^*(q_0, w) \in F$, with q_0 the initial state*

When we are given a DFA $D = (Q, \Sigma, \delta, q_0, F)$, the **language recognized by** D is the set, $L(D)$, of all words that are accepted. This is formalized in the definition below, where we expand the notion of δ, with $\delta^* : Q \times \Sigma^* \to Q$. Here the function δ^* is the inductive extension of δ to words, and it returns the state that the automaton reaches after processing the word w.

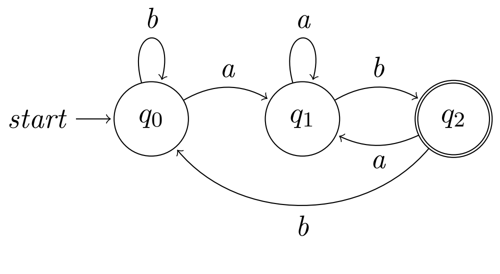
Definition 6 (Recognized Language of a DFA)**.**

$$L(D) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}, \textit{with } \delta^* : Q \times \Sigma^* \to Q$$

Example 1 (DFA Accepting Strings Ending in 'ab')**.** *Consider the DFA below, over the alphabet $\Sigma = \{a, b\}$ that accepts all words ending in 'ab'. The DFA tuple looks like this: $D = (Q, \Sigma, \delta, q_0, F)$, and is further defined as follows:*

- $Q = \{q_0, q_1, q_2\}$*, the set of states.*
- $\Sigma = \{a, b\}$*, the alphabet.*
- δ*, the transition function:*
  - $\delta(q_0, b) = q_0$
  - $\delta(q_0, a) = q_1$
  - $\delta(q_1, b) = q_2$
  - $\delta(q_1, a) = q_1$
  - $\delta(q_2, a) = q_1$
  - $\delta(q_2, b) = q_0$
- q_0*, the initial state.*
- $F = \{q_2\}$*, the set of accepting states.*

The DFA accepts strings like 'ab', 'aab', 'bbab', but does not accept 'ba' and 'bb'. The language recognized by this DFA is $L(D) = \{w \in \Sigma^* \mid w \text{ ends in 'ab'}\}$. The DFA D is visually represented as:



2.1.3 Operations on DFAs

We can perform operations on the languages recognized by DFAs, such as intersections, unions and making the complement, which correspond to constructing new automata from existing ones.

Definition 7 (Intersection of DFAs)**.** *The* ***intersection*** *of two DFAs D_1 and D_2, denoted as $D_1 \times D_2$, is a DFA that accepts a word if both D_1 and D_2 accept it.*
To define a DFA $D_1 \times D_2$, let $D_1 = (Q_1, \Sigma, \delta_1, q_{0_1}, F_1)$ and $D_2 = (Q_2, \Sigma, \delta_2, q_{0_2}, F_2)$ then $D_1 \times D_2 = (Q_{D_1 \times D_2}, \Sigma, \delta_{D_1 \times D_2}, q_{0_{D_1 \times D_2}}, F_{D_1 \times D_2})$, where we use pairs of states of the original automata and make transitions simultaneously.
We can define the components of $D_1 \times D_2$ as follows:

- $Q_{D_1 \times D_2} = \{(q_1, q_2) \mid q_1 \in Q_1 \text{ and } q_2 \in Q_2\}$
- $\delta_{D_1 \times D_2} : Q_{D_1 \times D_2} \times \Sigma \to Q_{D_1 \times D_2}$,
  $\delta_{D_1 \times D_2}((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$
- $q_{0_{D_1 \times D_2}} = (q_{0_1}, q_{0_2})$
- $F_{D_1 \times D_2} = \{(q_1, q_2) \mid q_1 \in F_1 \text{ and } q_2 \in F_2\}$

Lemma 1. *The language recognized by the automata denoted as $D_1 \times D_2$ is $L(D_1 \times D_2) = L(D_1) \cap L(D_2)$*

Definition 8 (Complement of a DFA)**.** *The* ***complement*** *of a DFA D, denoted $\overline{L(D)}$, is the DFA that accepts all words that D does not accept.*

The complement of a DFA is constructed by turning all accepting states into non-accepting states, and vice versa.

Definition 9 (Union of DFAs)**.** *The* ***union*** *of two DFAs D_1 and D_2, denoted as $D_1 \cup D_2$, is a DFA that accepts a word if D_1 or D_2 accepts it.*

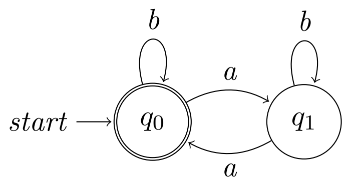
Lemma 2. *The language recognized by the DFA denoted as $D_1 \cup D_2$, is $L(D_1) \cup L(D_2)$.*

We can construct the union using De Morgan's Law, resulting in:

$$L(D_1) \cup L(D_2) = \overline{\overline{L(D_1)} \cap \overline{L(D_2)}}$$

Which means that we first take the complement of both languages, take the intersection of the resulting language, and then take the complement of the intersection to obtain the language for the union.

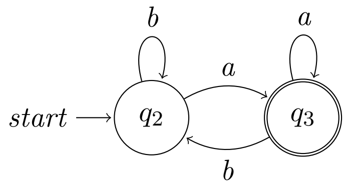
Example 2. *Let D_1 be the DFA that accepts the language $\{w \in \Sigma^* \mid |w|_a \text{ is even}\}$. The DFA can be visually represented as follows:*



Thus, $D_1 = (\{q_0, q_1\}, \{a, b\}, \delta_1, q_0, \{q_0\})$*, where the transition function* δ_1 *is defined as:*

$$\delta_1(q_0, a) = q_1, \quad \delta_1(q_0, b) = q_0, \quad \delta_1(q_1, a) = q_0, \quad \delta_1(q_1, b) = q_1.$$

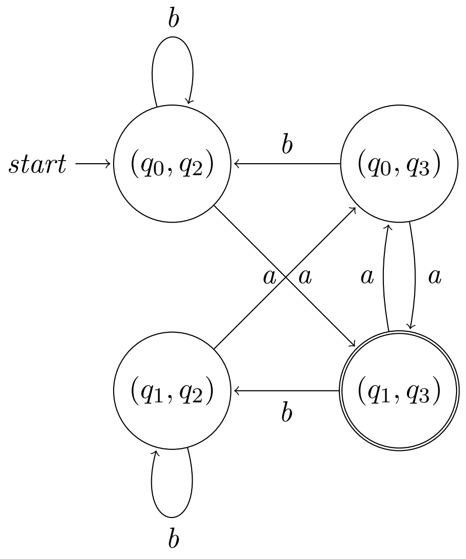
Now, let D_2 *be the DFA that accepts the language* $\{w \in \Sigma^* \mid w$ *ends in* 'a'$\}$*. The DFA can be visually represented as follows:*



Thus, $D_2 = (\{q_2, q_3\}, \{a, b\}, \delta_2, q_2, \{q_3\})$*, where the transition function* δ_2 *is defined as:*

$$\delta_2(q_2, a) = q_3, \quad \delta_2(q_2, b) = q_2, \quad \delta_2(q_3, a) = q_3, \quad \delta_2(q_3, b) = q_2.$$

Now, using the previously defined notions of the complement *and* intersection *of DFAs, the new DFA* $D_3 = \overline{D_1} \cap D_2$ *accepts the language* $\{w \in \Sigma^* \mid |w|_a$ *is odd, and* w *ends in* 'a'$\}$*. The DFA* D_3 *is visually represented as follows:*



Thus, $D_3 = (\{(q_0, q_2), (q_0, q_3), (q_1, q_2), (q_1, q_3)\}, \{a, b\}, \delta_3, (q_0, q_2), \{(q_1, q_3)\})$, *where the transition function* δ_3 *is defined as:*

$$\delta_3((q_0, q_2), a) = (q_1, q_3), \quad \delta_3((q_0, q_2), b) = (q_0, q_2),$$

$$\delta_3((q_0, q_3), a) = (q_1, q_3), \quad \delta_3((q_0, q_3), b) = (q_0, q_2),$$
$$\delta_3((q_1, q_2), a) = (q_0, q_3), \quad \delta_3((q_1, q_2), b) = (q_1, q_2),$$
$$\delta_3((q_1, q_3), a) = (q_0, q_3), \quad \delta_3((q_1, q_3), b) = (q_1, q_2).$$

2.1.4 Non-Deterministic Finite Automata (NFA)

A non-deterministic finite automaton (NFA) is defined as a tuple $N = (Q, \Sigma, q_0, \delta, F)$, where:

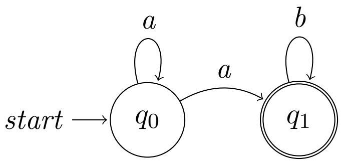
- Q is a finite set of states,
- Σ is a finite set of input symbols (alphabet),
- $q_0 \in Q$ is the initial state,
- $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ is the transition function, and
- $F \subseteq Q$ is the set of final/accepting states.

A word $w \in \Sigma^*$ is recognized by an NFA if there exists at least one path from the initial state to a final state. In contrast to deterministic finite automata (DFAs), an NFA can have multiple possible transitions for the same input symbol, or even no transitions at all.

Example 3. *Consider the NFA* $N = (\{q_0, q_1\}, \{a, b, c\}, \delta, q_0, \{q_1\})$, *where the transition function* δ *is defined as:*

$$\delta(q_0, a) = \{q_0, q_1\}, \quad \delta(q_1, b) = \{q_1\}.$$

This NFA accepts the language $\{a^x ab^y \mid x, y \in \mathbb{N}\}$. *We can visualize this NFA as follows:*



Although any NFA can be converted to an equivalent DFA by creating states representing all possible combinations of NFA states (yielding a DFA with up to $2^{|Q|}$ states, where $|Q|$ is the number of states in the original NFA), both models have identical expressive power. They recognize exactly the class of regular languages, which are also describable by regular expressions.

2.2 Control Improvisation

Control Improvisation is a framework for generating words that satisfy both hard constraints (ensuring all generated words are valid) and soft constraints (allowing some variability). The goal is to ensure randomness while adhering to the required specifications. In particular, the framework guarantees that the generated words are not only valid but also sufficiently diverse, by enforcing probabilistic bounds. This typically means that no single word is generated with a too high probability.

2.2.1 Specifications and Improvisations

In Control Improvisation, we distinguish between two types of specifications:

- **Hard specification ($\mathcal{H}$)**: A DFA as representation of a finite language that describes a strict set of rules that all generated words must satisfy.
- **Soft specification ($\mathcal{S}$)**: A DFA as representation of a regular language that describes a more relaxed set of rules, where at least $(1 - \epsilon) * 100\%$ percent of the generated words should adhere to these rules. Which we do for a chosen error probability $\epsilon \in [0, 1] \cap \mathbb{Q}$

To ensure finiteness, we restrict the hard specification $\mathcal{H}$ to a finite language. This means the automaton for $\mathcal{H}$ must not contain cycles that generate infinitely many words. Namely, any cycles must lead to a sink state from which no accepted words can be formed. As a result, the set of valid outputs is finite and can be fully explored. While [6] enforces finiteness through explicit length bounds on words, our method naturally prevents repetition of elements in accepted words, making it particularly suitable for applications where such patterns are undesirable.
When $\mathcal{H}$ is finite, all generated words come from this fixed set. The soft specification $\mathcal{S}$ then serves only as a probabilistic preference: it does not change which words are allowed, only how frequently they should appear. Therefore, $\mathcal{S}$ itself need not define a finite language.

Definition 10 (Improvisation)**.** *An* ***improvisation*** *is any word w accepted by the chosen hard specification $\mathcal{H}$. The set of all improvisations is defined as I:*

$$I = \{w \in L(\mathcal{H})\}.$$

While the set I is equivalent to the language $L(\mathcal{H})$ recognized by the hard specification, we adopt the I notation following the convention established in the Control Improvisation literature [6], where it serves to emphasize the distinction between the improvisation space and general language acceptance.

Definition 11 (Admissible Improvisation)**.** *An* ***admissible improvisation*** *is any improvisation that also is accepted by the soft specification $\mathcal{S}$. The set of all admissible improvisations is denoted A:*

$$A = \{w \in I \mid w \in L(\mathcal{S})\}.$$

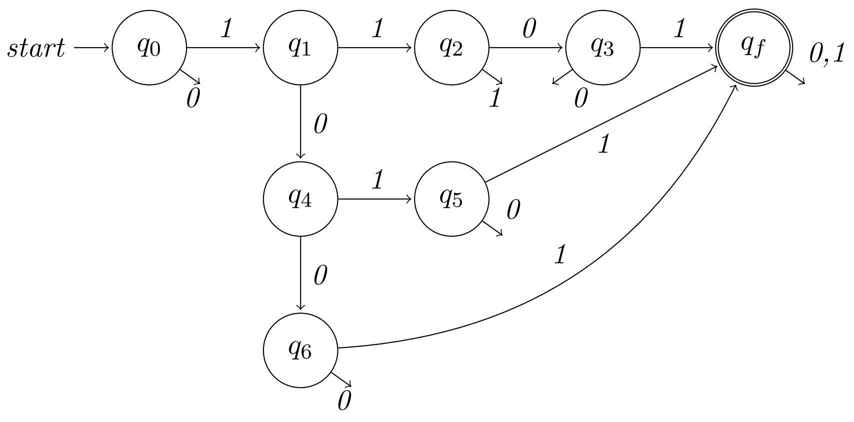
Since $L(\mathcal{H})$ is finite, the sets I and A are also finite, and therefore $|I|$ and $|A|$ are finite as well.

We introduce the concepts with an example.

Example 4 (Running example)**.** *Our goal is to generate variations of the word w = 1001. So, to generate words of length 4, that start with a 1 and end with a 1.*
For that task, we use the alphabet $\Sigma = \{0,1\}^*$ and the hard specification $\mathcal{H}$, which is defined as a finite DFA that accepts all words of length 4 that start with 1 and end with 1 and where the middle two have to be either 0 or 1. To ensure that the generated words remain close to the word w, we define the soft specification $\mathcal{S}$ as a DFA that accepts words that differ from w in at most one position. The set of improvisations I contains the words: 1001, 1011, 1101 and 1111. And the set of admissible improvisations A contains only the words: 1001, 1011, 1101. We visually represent the DFA for the hard specification $\mathcal{H}$ as below:

$$\begin{tikzcd}
\textit{start} \arrow[r] & q_0 \arrow[r, "1"] \arrow[rrrd, bend right, "0"] & q_1 \arrow[r, bend left, "0"] \arrow[r, bend right, "1"'] & q_2 \arrow[r, bend left, "0"] \arrow[r, bend right, "1"'] & q_3 \arrow[r, "1"'] \arrow[d, "0"] & q_f \arrow[ld, bend left, "0,1"] \\
 & & & & q_{sink} \arrow[loop below, "0,1"] &
\end{tikzcd}$$

As we can see, some transitions will send us to a state that will leave us stranded, they are going to the state q_{sink}. We can also remove the state all together and keep the arrows, it will represent the same, but it will be less visually crowded. Using this way of using the sink state, the visually representation of the DFA for the admissible words A looks like this:



This example shows how the constraints determine which words can be generated. But to really call it improvisation, we also want some randomness, so that the same word does not always come up, and every valid word has a fair probability of being chosen. This idea is captured by what is called an *improvising distribution.*

2.2.2 Improvising Distribution

Definition 12 (Control Improvisation problem Instance)**.** *A* ***Control Improvisation problem instance*** *is a tuple*

$$C = (\mathcal{H}, \mathcal{S}, \epsilon, \lambda, \rho)$$

where $\mathcal{H}$ denotes a hard specification, $\mathcal{S}$ denotes the a specification, $\epsilon \in [0,1] \cap \mathbb{Q}$ is the error probability, $\lambda \in [0,1] \cap \mathbb{Q}$ is the lower bound on probability and $\rho \in [0,1] \cap \mathbb{Q}$ is the upper bound on probability.

Definition 13 (Improvising Distribution)**.** *Given $C = (\mathcal{H}, \mathcal{S}, \epsilon, \lambda, \rho)$, with the variables as described above, an* ***improvising distribution*** *D, $D : \Sigma^* \to [0,1]$ is a probability distribution over I that satisfies:*

1. ***Hard constraint:*** *$Pr[w \in I \mid w \leftarrow D] = 1$*
2. ***Soft constraint:*** *$Pr[w \in A \mid w \leftarrow D] \geq 1 - \epsilon$*
3. ***Randomness:*** *$\lambda \leq D(w) \leq \rho$ for all $w \in I$*

To better understand the ideas presented here, we will go through each part:

- Hard constraint: $Pr[w \in I \mid w \leftarrow D] = 1$:
  $w \leftarrow D$ is a notation where w is generated by improviser D. The equitation can also be written as: $\sum_{w \in I} D(w) = 1$. This says that the probability (Pr) that W is generated from D is 100%. This means that the improviser must always ensure that it is generating words according to the set of all improvisations.

- Soft constraint: $Pr[w \in A \mid w \leftarrow D] \geq 1 - \epsilon$:
  The probability of the generated w's from D that are a part of A must lie between 1 and $1 - \epsilon$. Where the equation can be written again as a sum, namely $\sum_{w \in A} D(w) \geq 1 - \epsilon$. Which means that the probability of the words that are in I, but not in A are together at most ϵ. In other words, there is a probability of at most ϵ that a word is generated without satisfying the soft constraint. This requirement can be left out, because ϵ operates like a skewed coin flip controlling whether the soft constraint is applied.

- Randomness: $\lambda \leq D(w) \leq \rho$ for all $w \in I$:
  This represents the domain where the probability of each word lies. The smaller the domain, the more uniformly the words must be generated. The bigger the domain, the more random.

Example 5 (Running Example). *Using the same example as above, we consider the set of improvisations* $I = \{1001, 1011, 1101, 1111\}$ *and the set of admissible improvisations* $A = \{1001, 1011, 1101\}$. *We now introduce a probability distribution D over the set I, which assigns probabilities to each word in I while satisfying the conditions of the definition. And we are using* $\lambda = 0.1, \rho = 0,5$ *and* $\epsilon = 0,2$. *For instance, the distribution D could assign the following probabilities:* $D(1001) = 0.3$, $D(1011) = 0.3$, $D(1101) = 0.2$, $D(1111) = 0.2$ *This distribution satisfies all the conditions of definition 13:*

- *Requirement 1. The total probability of improvisations is* $0.3 + 0.3 + 0.2 + 0.2 = 1$, *which satisfies the requirement.*

- *Requirement 2. The total probability of the inadmissible improvisation* 1111 *is 0.2, which is within the allowed limit of* $\epsilon = 0.2$.

- *Requirement 3. Each improvisation has a probability of at least* 0.1. *And no improvisation has a probability exceeding* 0.5.

Now, consider distributing the probabilities uniformly over the set I, with a λ of 0.3. A uniform distribution would give each word in I a probability of $\frac{1}{|I|} = \frac{1}{4} = 0.25$. *However, this would violate the conditions of D:*
Requirement 2. The inadmissible improvisation 1111 *would have a probability of* 0.25, *resulting in exceeding the allowed limit of* $\epsilon = 0.2$.
Requirement 3. Each of the improvisations has a probability of 0.25, which is lower then $\lambda = 0.3$ *requires.*
Thus, a uniform distribution, while easiest to generate, is not allowed under the given constraints. To satisfy the conditions, the probabilities must be carefully assigned to ensure that the total probability of inadmissible improvisations does not exceed ϵ, while also respecting the lower and upper bounds λ and ρ for all improvisations.

2.2.3 Feasibility and Improvisation Algorithms

To make use of the control improvisation framework in practice, we first need to understand when a problem can actually be solved.

Definition 14 (Feasible)**.** *The tuple $C = (\mathcal{H}, \mathcal{S}, \epsilon, \lambda, \rho)$ is said to be* ***feasible*** *if there exists an improvising distribution D that meets the constraints described in definition 13.*

Once feasibility is established, the next step is to actually generate words according to such a distribution. This is done by a special kind of algorithm, called an *improviser.*

Definition 15 (Improviser Algorithm)**.** *An* ***improviser*** *is an algorithm that generates words according to an improvising distribution D for a feasible Control Improvisation problem C.*

The overall task of control improvisation is not just to check whether a solution is possible, but also to build such an algorithm if it is.

Definition 16 (Control Improvisation Problem)**.** *A* ***Control Improvisation problem*** *is the task of determining whether a given problem $C = (\mathcal{H}, \mathcal{S}, \epsilon, \lambda, \rho)$ is feasible, and, if feasible, generating an improviser.*

2.2.4 Feasibility Theorem

This section presents a theorem that characterizes when a Control Improvisation problem is feasible, by giving necessary and sufficient conditions for the existence of a valid improviser. The result aligns with [6] but uses our control improvisation instance C, rather than their instance with n and m as parameters, without affecting the theorem's validity.

Theorem 1 (Feasibility of Control Improvisation)**.** *For any Control Improvisation problem $C = (\mathcal{H}, \mathcal{S}, \epsilon, \lambda, \rho)$, the following conditions are equivalent:*

1. *C is feasible.*
2. *The following inequalities hold:*
   (a) $1/\rho \leq |I| \leq 1/\lambda$
   (b) $(1 - \epsilon)/\rho \leq |A|$
   (c) $|I| - |A| \leq \epsilon/\lambda$.
3. *There exists an improviser for C.*

To prove this theorem, we use the proof from [6], but written with more steps for clarity and understandability.

Proof of Theorem. **1⇒2**

a. We take D as an improvising distribution. Then the equality $\rho|I| = \sum_{w\in I}\rho$ holds, where $|I|$ is the number of words in I. And so $\sum_{w\in I}\rho \geq \sum_{w\in I} D(w)$ holds due to requirement 3 of definition 13. Where we can say

$$\sum_{w\in I} D(w) = Pr[w \in I \mid w \leftarrow D] = 1,$$

because of requirement 1 of definition 13. This leads to $|I| \geq \frac{1}{\rho}$. In the same way using requirements 1 and 3 of definition 13, we get

$$\lambda|I| = \sum_{w\in I}\lambda \leq \sum_{w\in I} D(w) = Pr[w \in I \mid w \leftarrow D] = 1,$$

which gives us $|I| \leq \frac{1}{\lambda}$. Using both results together we have $\frac{1}{\rho} \leq |I| \leq \frac{1}{\lambda}$.

b. Using the definition of A, we can say that $A \subseteq I$, because of that we also have $\rho|A| = \sum_{w\in A}\rho \geq \sum_{w\in A} D(w)$ using requirement 3 of definition 13. Using the definition of the probability, we can write it as: $\sum_{w\in A} D(w) = Pr[w \in A \mid w \leftarrow D]$, and using requirement 2 of definition 13 this holds: $Pr[w \in A \mid w \leftarrow D] \geq 1 - \epsilon$. Which leads to $\rho|A| \geq 1 - \epsilon$ *i.e.*, $|A| \geq \frac{1-\epsilon}{\rho}$.

c. We take $\lambda|I \setminus A| = \sum_{w\in I\setminus A}$, where we again take requirement 3 of definition 13, which makes $\sum_{w\in I\setminus A}\lambda \leq \sum_{w\in I\setminus A} D(w)$. Again using the definition of the probability we write it as

$$Pr[w \in I \setminus A \mid w \leftarrow D] = Pr[w \in I \mid w \leftarrow D] - Pr[w \in A \mid w \leftarrow D],$$

so we are looking at the probability that w satisfies the hard constraint minus the probability that w satisfies the soft constraint. Using both requirements 1 and 2 of definition 13 on those probabilities, we write

$$Pr[w \in I \mid w \leftarrow D] - Pr[w \in A \mid w \leftarrow D] \leq 1 - (1 - \epsilon) = \epsilon.$$

Combining this result with our starting value, we get $\lambda|I \setminus A| \leq \epsilon \Rightarrow |I| - |A| \leq \frac{\epsilon}{\lambda}$.

With this, we have showed that when C is feasible, the inequalities of (2) hold.

2⇒3 We define the improviser D to be the distribution on I which picks uniformly from A with probability $1 - \epsilon_{opt}$, and otherwise it picks uniformly from $I \setminus A$. Furthermore we define ϵ_{opt} as $max(1 - \rho|A|, \lambda|I \setminus A|)$, using $1 - \rho|A|$ from (2b) and $\lambda|I \setminus A|$ from (2c). Where the distribution D is well-defined, since if $A = \emptyset$ then $\epsilon_{opt} = 1$, and if $I \setminus A = \emptyset$, then we can say that $\rho|A| = \rho|I|$ and with (2a) $\rho|I| \geq 1$, which shows $\epsilon_{opt} = 0$.

1. D satisfies the hard constraint, because from the definition it follows that it picks from I, so $Pr[w \in I \mid w \leftarrow D] = 1$.

2. If we have $\epsilon_{opt} = 1 - \rho|A|$,it follows that $\epsilon_{opt} \leq 1 - \rho\frac{1-\epsilon}{\rho} = \epsilon$ due to (2b). And if $\epsilon_{opt} = \lambda|I \setminus A|$, it follows that $\epsilon_{opt} \leq \lambda\frac{\epsilon}{\lambda} = \epsilon$, due to (2c). Where in both cases we have

$$Pr[w \in I \mid w \leftarrow D] = 1 - \epsilon_{opt} \geq 1 - \epsilon,$$

which satisfies the soft constraint.

3. We can say, due to the uniformness and the fact that the probability that $w \in A$ is $1 - \epsilon_{opt}$, that $\forall w \in A : D(w) = \frac{1-\epsilon_{opt}}{|A|}$. If $\epsilon_{opt} = 1 - \rho|A|$, then $D(w) = \frac{1-(1-\rho|A|)}{|A|} = \rho$, using (2a) gives $D(w) = \rho \geq \lambda$. And otherwise, if $\epsilon_{opt} = \lambda|I \setminus A|$, then $D(w) = \frac{1-\lambda|I\setminus A|}{|A|}$. We can write this differently by splitting the $|I \setminus A|$ and using (2a):

$$D(w) = \frac{1 - \lambda|I| + \lambda|A|}{|A|} \geq \frac{1 - 1 + \lambda|A|}{|A|} = \lambda,$$

thus $D(w) \geq \lambda$ in both of the cases. Likewise, when $\forall w \in I \setminus A$ we have $D(w) = \frac{\epsilon_{opt}}{|I\setminus A|} \geq \frac{\lambda|I\setminus A|}{|I\setminus A|} = \lambda$ due to (2c) if $\epsilon_{opt} = \lambda|I \setminus A|$. And if $\epsilon_{opt} = 1 - \rho|A|$, we can write $D(w) = \frac{1-\rho|A|}{|I\setminus A|} = \frac{1-\rho|A|}{|I|-|A|}$, and due to (2a) $\leq \frac{1-\rho|A|}{\frac{1}{\rho}-|A|} = \rho$. So for any $w \in I$ we have that D satisfies $\lambda \leq D(w) \leq \rho$, which satisfies the randomness constraint.

Concluding, this shows that D is an improvising distribution. Because the distribution has finite support and uses finite, rational probabilities, there is a probabilistic algorithm that samples from it in expected finite time. This makes it a valid improviser for C.

3⇒1 This is immediate, because when there exist an algorithm, we can get the distribution from it. □

2.2.5 Algorithm for Feasible Control Improvisation

To construct an improviser for a feasible Control Improvisation problem, we use the following algorithm, which is based on the proof of Theorem 1:

Algorithm 1 Improviser Algorithm

```
1: **Input:** $C = (\mathcal{H}, \mathcal{S}, \epsilon, \lambda, \rho)$
2: **Output:** Word $w \in I$
3: $A \leftarrow$ admissible improvisations
4: $I \leftarrow$ all improvisations
5: $p \leftarrow 1 - \epsilon$
6: **if** Flip coin with probability $p$ **then**
7:     Pick uniformly $w \in A$
8: **else**
9:     Pick uniformly $w \in I \setminus A$
10: **end if**
11: **Return** $w$
```

Theorem 2. *Algorithm 1 constructs an improviser for a feasible Control Improvisation problem using a DFA to represent the hard specification $\mathcal{H}$.*

To prove this theorem, we provide an outline of the reasoning behind the algorithm's correctness.

Proof. This algorithm constructs an improviser for a feasible Control Improvisation problem by flipping a biased coin to decide whether to select a word from the set of admissible improvisations A or from the set of non-admissible improvisations $I \setminus A$. First, the sets A and I are initialized, where A consists of admissible improvisations and I consists of all improvisations. A biased coin is then flipped with probability $p = 1 - \epsilon$, where ϵ is the allowed error probability. If the coin lands on heads, a word is selected uniformly from A. Otherwise, if the coin lands on tails, a word is selected uniformly from $I \setminus A$. The selected word w is then returned.
The algorithm guarantees that a word is always picked uniformly from either A or $I \setminus A$, which is always possible because if λ and ρ would not be in the range to pick uniformly from it, it would not satisfy all the constraints, and thus C would not be feasible.
This algorithm satisfies the following conditions:

- Hard Constraint: The word is always chosen from I, ensuring it is a valid improvisation.
- Soft Constraint: The probability of selecting from A is $1 - \epsilon$, ensuring that the majority of generated words meet the soft specification.
- Randomness: Words are selected uniformly from either A or $I \setminus A$, ensuring that the probability of selection is always between λ and ρ, thus satisfying the randomness constraint.

By ensuring that all constraints are respected, this algorithm works as an improviser in a feasible Control Improvisation problem. □

2.2.6 Visual Improviser Algorithm

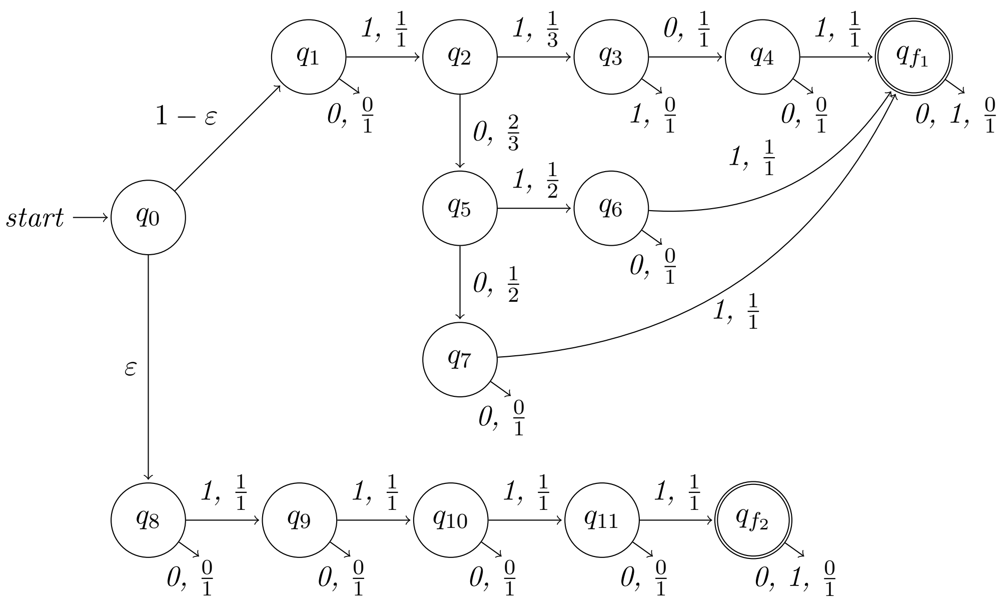
To not only construct an improviser but also visualize its behavior, we can assign probabilities to each transition in the DFA. These probabilities reflect the likelihood of each transition being taken when sampling from an improviser.
This is achieved by calculating the number of successful paths to final states through each transition and distributing probabilities accordingly.

Algorithm 2 Visual Improviser Algorithm

```
1: Input: DFAs $M_1$ representing $A$ and $M_2$ representing $I \setminus A$, error probability $\epsilon$
2: Output: Combined DFA with transition probabilities
3: Add a new initial state
4: Add transition from new initial state to $M_1$'s initial state with probability $1 - \epsilon$
5: Add transition from new initial state to $M_2$'s initial state with probability $\epsilon$
6: for each state $q$ in the combined DFA do
7:     $\texttt{paths}(q) \leftarrow$ #accepting paths from $q$
8: end for
9: for each transition $(q, q')$ do
10:     $p(q \to q') \leftarrow \frac{\texttt{paths}(q')}{\sum \texttt{paths}(q_i)}$, sum over successors $q_i$ of $q$
11: end for
12: for each transition leading to a sink state do
13:     Assign probability 0
14: end for
15: Return Combined DFA with transition probabilities
```

The algorithm first makes a combined automaton by joining two DFAs: one for admissible improvisations (A) and one for inadmissible ones ($I \setminus A$). Starting from the new initial state, we trace how many accepting paths come from each state. Then, for each transition, we compute the probability of choosing it by comparing how many accepting paths continue through that transition to the total number of accepting paths from that state. This models the behavior of a uniform improviser, by distributing probability uniformly to accepted outcomes.
The algorithm assigns higher probabilities to transitions that lead more words to a accepting state. In this way, the automaton ends up choosing between valid improvisations uniformly. Transitions that lead to a sink states, always get probability 0, so they are never used. As a result, this shows how a uniform improviser would look used on the DFA.

Example 6 (Running Example)**.** *Using the algorithm to visualize the improviser on the DFA, we get the following visualization:*



2.3 Tree Automata

We now introduce tree automata, a generalization of classical automata for processing tree-structured data instead of strings, following the formal framework of [4] In contrast to DFAs, which process linear sequences of symbols, tree automata must handle nodes with multiple children simultaneously. This structural difference makes it impossible to simulate tree behavior using standard DFAs, as they lack a mechanism to track multiple subtrees concurrently.
To reason formally about trees and tree automata, we need new definitions that capture the hierarchical and recursive nature of trees. In the following, we lay the foundation for working with tree automata by defining the key components, starting with ranked alphabets and the arity of function symbols.

Definition 17 (Arity and Ranked Symbols)**.** *A* ***ranked alphabet*** $\mathcal{F}$ *is a finite set of function symbols where each symbol* $f \in \mathcal{F}$ *is assigned an* ***arity*** *arity*$(f) \in \mathbb{N}$*, representing the number of children the symbol requires.*

The set of symbols with arity $x$is denoted by $\mathcal{F}_x$. The arity of a function symbol corresponds to the number of its arguments, which equals one plus the number of commas in its notation. For example:

- $f(,,)$ has arity 3 (two commas)
- $g(,)$ has arity 2 (one comma)

- A constant symbol a has arity 0 (no commas)

Definition 18 (Set of variables)**.** *χ is a set of constants called **variables**. Where the sets χ and $\mathcal{F}_0$ are disjoint.*

Definition 19 (Set of terms)**.** *The **set of terms** $T(\mathcal{F}, \chi)$ over the ranked alphabet $\mathcal{F}$ and set of variables χ is the smallest set defined by:*

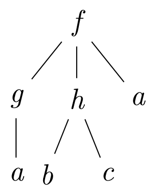
- $\mathcal{F}_0 \subseteq T(\mathcal{F}, \chi)$
- $\chi \subseteq T(\mathcal{F}, \chi)$
- *if $p \geq 1$, $f \in \mathcal{F}_p$ and $t_1, \ldots, t_p \in T(\mathcal{F}, \chi)$, then $f(t_1, \ldots, t_p) \in (\mathcal{F}, \chi)$*

When there are terms in $T(\mathcal{F})$, that is, where $\chi = \emptyset$, then they are called ground terms.
We use the name tree for a term that is represented graphically.

Example 7 (Arity)**.** *Let $\mathcal{F} = \{a, b, c, g(), h(,), f(,,)\}$ with:*

$$arity(a) = arity(b) = arity(c) = 0, \quad arity(g()) = 1, \quad arity(h(,)) = 2, \quad arity(f(,,)) = 3.$$

Using the functions from $\mathcal{F}$ together, we can make the ground term $f(g(a), h(b, c), a)$, which we can visualize in a graphical way as a tree:



Remark that a tree cannot be finite if there are no function symbols with arity 0 *i.e.*, constants. This is because it cannot stop growing downward, since every node has a child.

Definition 20. *A linear term $C \in T(\mathcal{F}, \chi_n)$ is called a **context**, where χ_n is a sequence of $x_1, \ldots, x_n$.*

When we use the expression $C[t_1, \ldots, t_n]$ for $t_1, \ldots, t_n \in T(\mathcal{F})$, it means that for $1 \leq i \leq n$ every variable x_i in C is replaced by t_i.

2.3.1 Bottom-Up Tree Automata

Now that we have covered the basics, we can define tree automata.

Definition 21 (Deterministic Finite Tree Automaton)**.** *A **Deterministic Finite Tree Automaton** is a tuple $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$, where:*

- *Q is a finite set of states,*

- *$\mathcal{F}$ is a ranked alphabet,*
- *$Q_f \subseteq Q$ are the accepting states,*
- *Δ is a set of rewrite rules of the following type: $f(q_1(x_1), \ldots, q_n(x_n)) \to q(f(x_1, \ldots, x_n))$.*
  Where $n \geq 0$, $f \in \mathcal{F}_n$, $q, \ldots, q_n \in Q$, $x_1, \ldots, x_n \in \chi$
  No two distinct rules in Δ have identical left-hand sides for the same function symbol and states.

Here we specifically say Deterministic and Finite, because it will narrow down a lot of different possibilities. We say finite because it operates on a finite set of states and a finite set of rules. Determinism of the automaton in this scenario means that there are no two rules with the same left-hand side.

Definition 22 (Transitions $\to_{\mathcal{A}}$)**.** *Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ be a Tree Automaton over $\mathcal{F}$. The relation $t \to_{\mathcal{A}} t'$ is defined by: let $t, t' \in T(\mathcal{F} \cup Q)$,*

$$t \to_{\mathcal{A}} t' := \begin{cases} \exists C \in K(\mathcal{F} \cup Q),\ \exists u_1, \ldots, u_n \in T(\mathcal{F}), \\ \exists f(q_1(x_1), \ldots, q_n(x_n)) \to q(f(x_1, \ldots, x_n)) \in \Delta, \\ t = C[f(q_1(u_1)), \ldots, q_n(u_n))], \\ t' = C[q(f(u_1, \ldots, u_n))]. \end{cases}$$

This means that if a node f has subtrees rooted in states $q_1, \ldots, q_n$, and there is a rule specifying how these combine into a parent state q, then the automaton can rewrite the subtree rooted at f by replacing the children's states with a single state q at the parent. The transition applies locally to this node, while the surrounding tree remains unchanged.

Definition 23 (Reflexive-Transitive Closure of Transitions)**.** *Let $\mathcal{A}$ be a Tree Automaton. We write $t \xrightarrow[\mathcal{A}]{*} t'$ to denote that t' can be obtained from t through zero or more applications of the transition relation $\to_{\mathcal{A}}$. This is the reflexive-transitive closure of $\to_{\mathcal{A}}$.*

To demonstrate how transitions apply to a tree, consider the following example.

Example 8 (Basic transitions)**.** *Let $\mathcal{F} = \{a, f()\}$, with arity$(a) = 0$, arity$(f) = 1$. Let $Q = \{q_a, q_f\}$ and $Q_f = \{q_f\}$. Define the transitions:*

$$\Delta = \{a \to q_a(a), \quad f(q_a(x)) \to q_f(f(x))\}$$

We look at the transitions of the ground term $f(a)$.

We can apply $a \to_{\mathcal{A}} q_a(a)$ just like this:

$$\begin{tikzcd}
f \arrow[d, no head] \\
q_a \arrow[d, no head] \\
a
\end{tikzcd}$$

After that, we apply $f(q_a(x)) \rightarrow_{\mathcal{A}} q_f(f(x))$ *like so:*

$$\begin{tikzcd}
q_f \arrow[d, no head] \\
f \arrow[d, no head] \\
a
\end{tikzcd}$$

Now we have seen how the transitions are used on initial rules, which are transition rules for constant symbols, and transitions where the arity is only 1. The next example demonstrates how transitions can handle function symbols with higher arity.

Example 9 (Transitions with higher arity)**.** *Given* $f(q_1(x), q_2(y)) \rightarrow q_3(f(x, y))$*, we can use this transition on the tree below, substituting the variables* x *and* y *as* a *and* b *respectively.*

$$f(q_1(a), q_2(b)) \rightarrow_{\mathcal{A}} q_3(f(a, b))$$

$$\begin{tikzcd}
 & f \arrow[dl, no head] \arrow[dr, no head] & & & & q_3 \arrow[d, no head] & \\
q_1 \arrow[d, no head] & & q_2 \arrow[d, no head] & \rightarrow_{\mathcal{A}} & & f \arrow[dl, no head] \arrow[dr, no head] & \\
a & & b & & a & & b
\end{tikzcd}$$

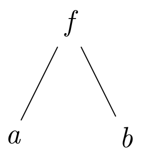
Definition 24 (Acceptance)**.** *A term* $t \in T(\mathcal{F})$ *is* ***accepted*** *by a tree automaton* $\mathcal{A}$ *if* $t \xrightarrow[\mathcal{A}]{*} q(t)$ *and* $q \in Q_f$*.*

Example 10. *Let the ranked alphabet* $\mathcal{F} = \{a, b, f(,)\}$ *and state set* $Q = \{q_a, q_b, q_f\}$*, with* $Q_f = \{q_f\}$*. Define the transition rules:*

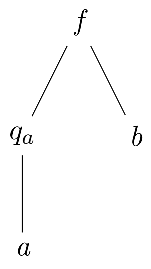
$$\Delta = \{a \rightarrow q_a(a), \quad b \rightarrow q_b(b), \quad f(q_a(x), q_b(y)) \rightarrow q_f(f(x, y))\}$$

We want to show how the ground term $f(a,b)$ is accepted. And that it follows out of $f(a,b) \xrightarrow[\mathcal{A}]{*} f(a,b)$

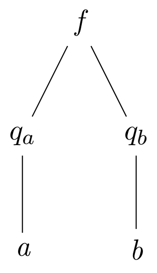
Initial tree:



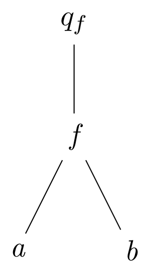
Step 1: *Apply $a \to q_a(a)$*



Step 2: *Apply $b \to q_b(b)$*



Step 3: *Apply $f(q_a(x), q_b(y)) \to q_f(f(x,y))$*



The final term $q_f(f(a,b))$ is accepted because $q_f \in Q_f$.

Definition 25 (Recognized Language)**.** *The **language** recognized by a tree automaton $\mathcal{A}$ is:*

$$L(\mathcal{A}) = \{t \in T(\mathcal{F}) \mid t \xrightarrow[\mathcal{A}]{*} q(t),\ q \in Q_f\}$$

The above language is the set consisting of all ground terms that are accepted by $\mathcal{A}$.

2.3.2 Top-Down Tree Automata

We distinguish between top-down and bottom-up tree automata. In the previous section, we discussed bottom-up automata, which start at the leaves of a tree and move upward toward the root, following the typical tree diagram with the root at the top.
In this section, we introduce top-down automata, which begin at the root in an initial state and proceed downward along the tree, level by level.

Definition 26 (Deterministic Finite Top-Down Tree Automaton)**.** *A* ***Deterministic Finite Top-Down Tree Automaton*** *is defined as* $\mathcal{A} = (Q, \mathcal{F}, q_0, \Delta)$*, where:*

- Q *is a finite set of states,*
- $\mathcal{F}$ *is a ranked alphabet,*
- $q_0 \in Q$ *is the initial state,*
- Δ *consists of rules* $q(f(x_1, \dots, x_n)) \to f(q_1(x_1), \dots, q_n(x_n))$*.*
  Where $n \geq 0$*,* $f \in \mathcal{F}_n$*,* $q, \dots, q_n \in Q$*,* $x_1, \dots, x_n \in \chi$
  No two distinct rules in Δ *have identical left-hand sides for the same function symbol and states.*

Here we say it is Finite because it operates on a finite set of states and finite set of rules. Determinism is ensured when there is exactly one initial state and no two rules share the same left-hand side.
As we can see in the definition of the top-down automata, the transition rules are different from the bottom-up automata, to accompany the change in direction that the proceedings will use.

Definition 27 (Transitions (Top-Down))**.** *Let* $\mathcal{A} = (Q, \mathcal{F}, q_0, \Delta)$ *be a top down Tree Automaton over* $\mathcal{F}$*. The relation* $t \to_{\mathcal{A}} t'$ *is defined by: let* $t, t' \in T(\mathcal{F} \cup Q)$*,*

$$t \to_{\mathcal{A}} t' := \begin{cases} \exists C \in K(\mathcal{F} \cup Q), \exists u_1, \dots, u_n \in T(\mathcal{F}), \\ \exists q(f(x_1), \dots, x_n) \to f(q_1(x_1), \dots, q_n(x_n)) \in \Delta, \\ t = C[q(f(u_1, \dots, u_n))], \\ t' = C[f(q_1(u_1), \dots, q_n(u_n))]. \end{cases}$$

This means that if a transition rule matches a node labeled f with state q, then the automaton rewrites that node by assigning new states $q_1, \dots, q_n$ to its children, according to the rule. The tree is updated locally at that position, while the rest of the tree remains unchanged.

Definition 28 (Acceptance (Top-Down)). *A term $t \in T(\mathcal{F})$ is* ***accepted*** *by a top-down tree automaton $\mathcal{A}$ if $q_0(t) \xrightarrow[\mathcal{A}]{*} t,\ q_0$ as initial state.*

Thus, a term t is accepted by a top-down tree automaton if, starting from the root in q_0, none or multiple transitions from Δ lead to leaf nodes, existing of constants.
When we collect all the ground terms, it is the language recognized by the automaton.

Definition 29 (Language Recognition (Top-Down)). *The* ***variables*** *recognized by a tree automaton $\mathcal{A}$ is:*

$$L(\mathcal{A}) = \{t \in T(\mathcal{F}) \mid q_0(t) \xrightarrow[\mathcal{A}]{*} t,\ q_0 \textit{ as initial state}\}$$

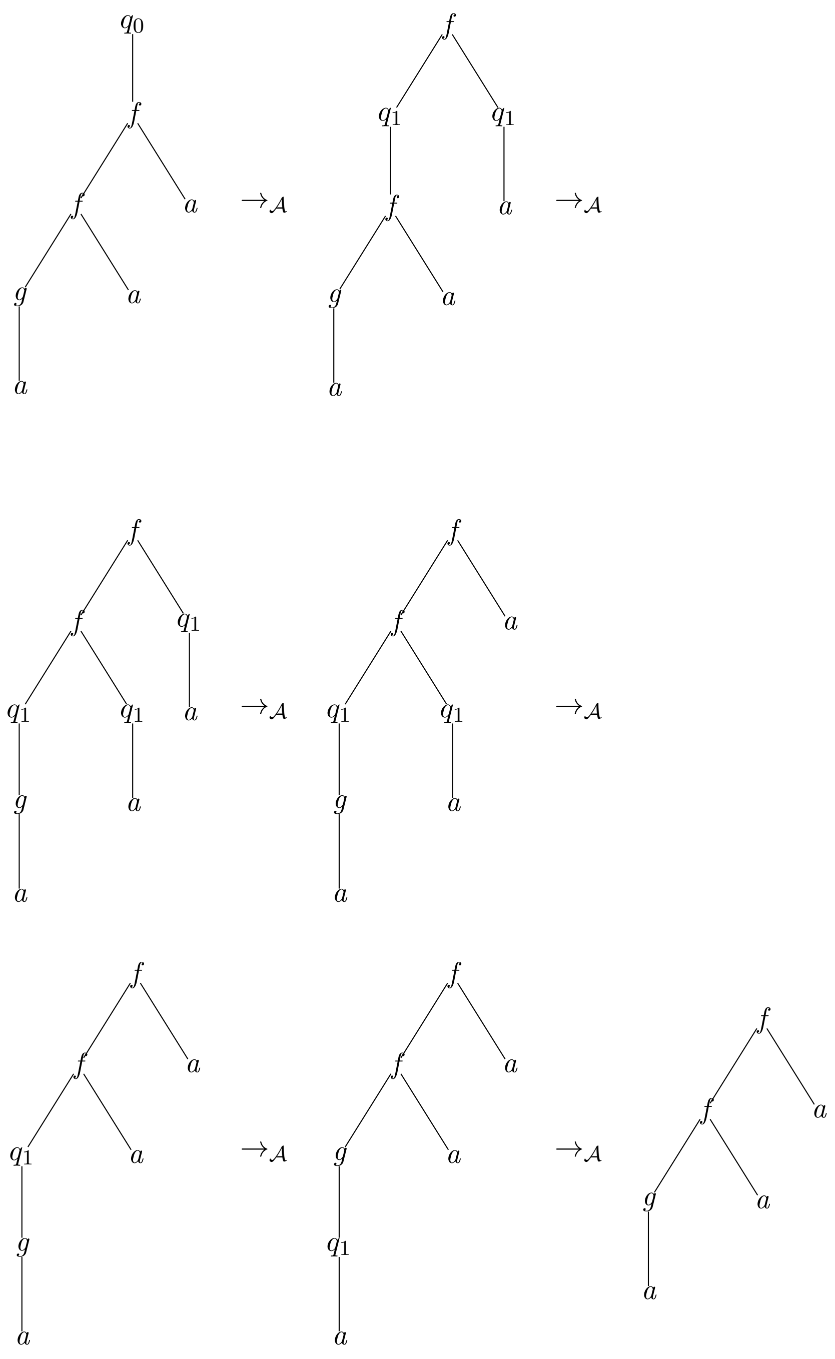
In the following, we will show how the transitions work and when they are accepted, using visuals to make it clearer.

Example 11 (Top-Down Acceptance). *Let $\mathcal{F} = \{a, f(,), g()\}$, the state set be $Q = \{q_0, q_1\}$, with initial state q_0, and define the top-down tree automaton $\mathcal{A} = (Q, \mathcal{F}, q_0, \Delta)$, where:*

$$\Delta = \begin{Bmatrix} q_0(f(x,y)) \rightarrow f(q_1(x), q_1(y)), \\ q_1(f(x,y)) \rightarrow f(q_1(x), q_1(y)), \\ q_1(g(x)) \rightarrow g(q_1(x)), \\ q_1(a) \rightarrow a \end{Bmatrix}$$

With $\chi = \{x, y\}$. We show that the term $q_0(f(f(g(a), a), a))$ is accepted:

$$\begin{aligned} q_0(f(f(g(a),a),a)) &\rightarrow_{\mathcal{A}} f(q_1(f(g(a),a)), q_1(a)) \\ &\rightarrow_{\mathcal{A}} f(f(q_1(g(a)), q_1(a)), q_1(a)) \\ &\rightarrow_{\mathcal{A}} f(f(q_1(g(a)), q_1(a)), a) \\ &\rightarrow_{\mathcal{A}} f(f(q_1(g(a)), a), a) \\ &\rightarrow_{\mathcal{A}} f(f(g(q_1(a)), a), a) \\ &\rightarrow_{\mathcal{A}} f(f(g(a), a), a) \end{aligned}$$



As we can see, the tree is accepted by the automaton, because $q_0(f(f(g(a),a),a)) \xrightarrow[\mathcal{A}]{*}$ $f(f(g(a),a),a)$ *and* q_0 *is an initial state.*

2.3.3 Deterministic vs Non-deterministic FTA

When researching tree automata, determinism plays a big role in understanding its expressive power. The relation between deterministic and non-deterministic models vary depending on whether the automaton operates in a top-down or bottom-up way.
For bottom-up finite tree automata, determinism does not restrict the class of recognized languages. Every tree language that is recognizable by a non-deterministic bottom-up tree automaton can also be recognized by a deterministic bottom-up tree automaton. Also it appears that non-deterministic top-down tree automata and bottom-up tree automata have the same expressive power. These languages are known as the regular tree languages. This equivalence is analogous to the result in DFAs and is confirmed in Theorem 1.6.1 of [4].

The situation is different for top-down tree automata. A top-down tree automaton is said to be deterministic if it has a single initial state and no two transitions with the same left-hand side. Unlike the bottom-up case, deterministic top-down tree automata are strictly less powerful than their non-deterministic counterparts. This is formally established in Proposition 1.6.2 of [4].
The class of tree languages that can be recognized by deterministic top-down tree automata is known to be path-closed. Intuitively, these are languages that are closed under the removal of paths.

Definition 30 (Path Language of a Tree)**.** *Let t be a ground term over a ranked alphabet $\mathcal{F}$. The* ***path language*** *$\pi(t)$ of the groundterm t is defined inductively as follows:*

$$\pi(t) = \begin{cases} t & \textit{if } t \in \mathcal{F}_0 \textit{ (a constant)} \\ \bigcup_{i=1}^{n} \{f_i w \mid w \in \pi(t_i)\} & \textit{if } t = f(t_1, \ldots, t_n) \end{cases}$$

where f_i denotes the function symbol f annotated with the child position i, and w is a path from the subterm t_i.

The path language $\pi(t)$ of a term captures all root-to-leaf paths, where each path is a sequence of symbols that record the structure of the term. If the term is just a constant, its only path is itself. If the term is a function symbol applied to subterms, each path from a subterm is prefixed with the corresponding f_i to indicate which branch it came from.

Definition 31 (Path Language and Path Closure of a Tree Language)**.** *Let $L \subseteq T(\mathcal{F})$ be a tree language. The* ***path language*** *of L is defined as:*

$$\pi(L) = \bigcup_{t \in L} \pi(t).$$

The ***path closure*** *of L is defined as:*

$$\text{pathclosure}(L) = \{t \in T(\mathcal{F}) \mid \pi(t) \subseteq \pi(L)\}.$$

The path language $\pi(L)$ collects all paths from all trees in L. The path closure of L, written pathclosure(L), is the set of all terms whose paths are entirely contained in $\pi(L)$. Which means it includes every term that doesn't introduce any new path beyond those already in L.

Definition 32 (Path-Closed Tree Language)**.** *A tree language $L \subseteq T(\mathcal{F})$ is said to be* ***path-closed*** *if:*

$$\text{pathclosure}(L) = L.$$

A path-closed language is one that contains all terms that are consistent with its set of paths. This means that if a term t is in the language, then any term t' that results from pruning subterms (removing branches and thus reducing paths) must also be in the language.
Due to their path-closed nature, deterministic top-down tree automata cannot recognize all regular tree languages, nor even all finite tree languages. For example, consider the finite language

$$L = \{g(f(a), b),\ g(a, f(b))\}.$$

If pruning either tree to $g(a, b)$ (where all paths of $g(a, b)$ exist in L), the resulting term is not in L. Thus, L is not path-closed and cannot be recognized by a deterministic top-down automaton.

Chapter 3

Research

This chapter develops the theoretical and algorithmic foundations for extending the Control Improvisation framework to trees. We begin by analyzing the structural restrictions necessary to ensure finiteness in the setting of deterministic finite tree automata. With these constraints in place, we adapt the general CI framework which we previously applied to word-based inputs, to work with tree-structured data.
The chapter culminates with algorithms that implement uniform sampling for feasible instances, both in the top-down DFTA (TDDFTA) and bottom-up DFTA (BUDFTA) settings.

3.1 Restrictions on Deterministic Finite Tree Automata

In this chapter, there is one important restriction we impose, and it is a bound on the height of trees. Without any restrictions, trees could be made with a infinite hight, resulting in an infinite language. For such a language, an improviser becomes impractical: either we must pick a finite subset (which can then be described by its own tree automaton), or we keep the infinite language, in which case the probabilities assigned to individual trees become so small that they are essentially meaningless.
In the case of our defined Deterministic Finite Tree Automata (DFTA), we observe that infinite trees can only arise if the automaton has cycles. More specifically, a cycle occurs when the same state appears both on the left-hand side and on the right-hand side of a rule. For example, consider these rules from example 11:

$$q_1(a) \to a \quad \text{and} \quad q_1(g(x)) \to g(q_1(x))$$

These rules allow the generation of terms like $g(g(\dots g(a)\dots))$, with arbitrary depth. To avoid such cases, we restrict our attention to DFTA that are acyclic—that is, they contain no rules where a state recursively calls itself.

As discussed in the Preliminaries, such automata recognize finite languages. This ensures that all generated trees are of bounded height, keeping the language finite and the improvisation problem tractable.

3.2 Control Improvisation for DFTA

To apply Control Improvisation to DFTA, we adapt the framework used for DFA to trees by replacing words with trees and classical automata with tree automata. The language of a tree automaton then consists of all accepted trees, as defined earlier.
In Control Improvisation for trees, we again distinguish between two types of specifications:

- **Hard specification ($\mathcal{H}$)**: A BUDFTA or TDDFTA as a representation of a finite tree language (or a path-closed tree language in the top-down case), describing a strict set of rules that all generated trees must satisfy.
- **Soft specification ($\mathcal{S}$)**: A BUDFTA or TDDFTA as a representation of a regular tree language that describes a more relaxed set of rules, where at least $(1 - \epsilon) \times 100\%$ of the generated trees should satisfy it, with $\epsilon \in [0, 1] \cap \mathbb{Q}$ the chosen error probability.

As with DFAs, it suffices to restrict the hard specification $\mathcal{H}$ to ensure that the set of generated trees is finite.

Definition 33 (Improvisation)**.** *An* ***improvisation*** *is any tree t that is accepted by the hard specification $\mathcal{H}$. The set of all such trees is called I:*

$$I = \{t \in L(\mathcal{H})\}.$$

As introduced in the Preliminaries, we denote the set of improvisations by I, following the standard Control Improvisation notation. We continue this usage here to avoid confusion with other language symbols, even though $I = L(\mathcal{H})$.

Definition 34 (Admissible Improvisation)**.** *An* ***admissible improvisation*** *is any tree $t \in I$ that is also accepted by the soft specification $\mathcal{S}$. The set of all admissible improvisations is denoted by A:*

$$A = \{t \in I \mid t \in L(\mathcal{S})\}.$$

To show how this is used with the DFTA, we will give an example.

Example 12 (Running example DFTA)**.** *In this example, we construct trees that satisfy the following set of rules:*

- *The function symbol f, of arity 2, must appear at the root of each tree.*
- *Each child of f can be either a constant or a function g of arity 1.*
- *The function g can have only constants as a child.*
- *We only use constants a and b.*

These rules are captured by the hard specification $\mathcal{H}$. We model $\mathcal{H}$ using both a BUDFTA and a TDDFTA, each accepting the same tree language.

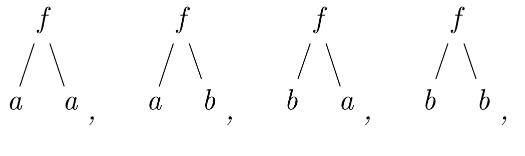
***TDDFTA**: Let $\mathcal{F}_1 = \{a, b, g(), f(,)\}$, the state set be $Q_1 = \{q_0, q_1, q_2\}$, with initial state q_0 and define the TDDFTA $\mathcal{A}_1 = (Q_1, \mathcal{F}_1, \{q_0\}, \Delta_1)$, with $\chi_1 = \{x, y\}$, where:*

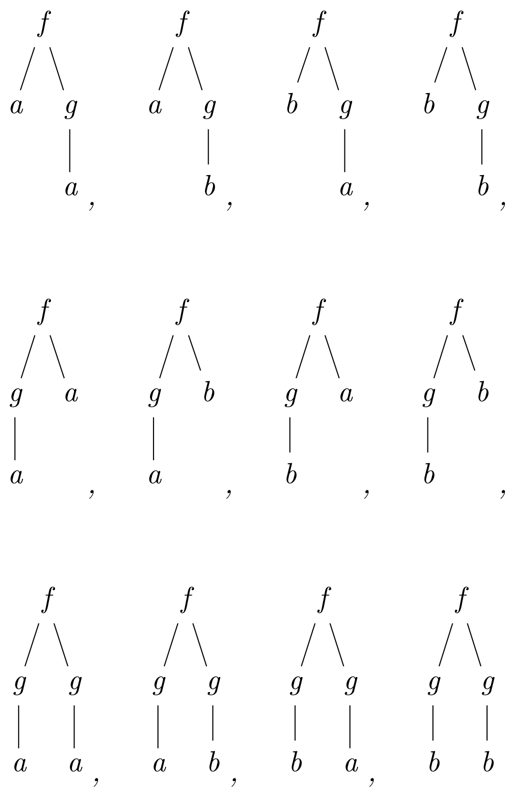
$$\Delta_1 = \left\{ \begin{array}{l} q_0(f(x,y)) \to f(q_1(x), q_1(y)), \\ q_1(g(x)) \to g(q_2(y)), \\ q_1(a) \to g(q_1(x)), \\ q_1(a) \to a, \\ q_2(a) \to a, \\ q_1(b) \to b, \\ q_2(b) \to b \end{array} \right\}$$

***BUDFTA**: Let $\mathcal{F}_2 = \{a, b, g(), f(,)\}$, the state set be $Q_2 = \{q_0, q_1, q_f\}$, with $Q_{2_f} = \{q_f\}$ and define the BUDFTA $\mathcal{A}_2 = (Q_2, \mathcal{F}_2, \{q_f\}, \Delta_2)$, with $\chi_2 = \{x, y\}$, where:*

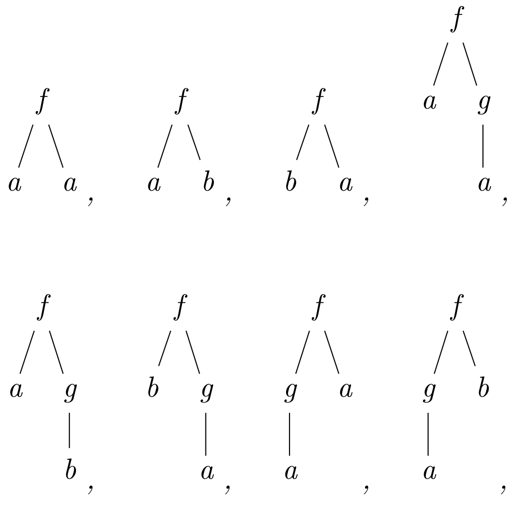
$$\Delta_2 = \left\{ \begin{array}{l} a \to q_0(a), \\ b \to q_0(b), \\ g(q_0(x)) \to q_1(g(x)), \\ f(q_0(x), q_0(y)) \to q_f(f(x,y)), \\ f(q_0(x), q_1(y)) \to q_f(f(x,y)), \\ f(q_1(x), q_0(y)) \to q_f(f(x,y)), \\ f(q_1(x), q_1(y)) \to q_f(f(x,y)) \end{array} \right\}$$

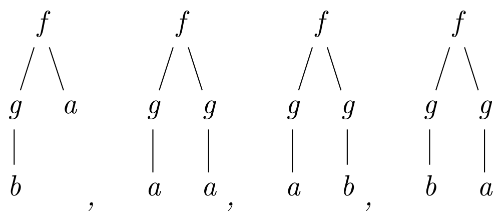
These DFTA generate the set of improvisations I, which consists of the following trees:





To refine this set, we define a soft specification $\mathcal{S}$ requiring that each tree must contain at least one occurrence of the constant a. This excludes all trees in which both constants are b. Now we have the set of admissible improvisations A, which consists of the following trees:





To formally define a Control Improvisation problem instance for trees, we adopt the same tuple used in our DFA implementation. Each component retains its original role, but now interpreted over tree languages. Without modifications needed since finiteness is already ensured by the structure of $\mathcal{H}$.

Definition 35 (Control Improvisation problem instance)**.** *We define a **Control Improvisation problem instance** as the tuple*

$$C = (\mathcal{H}, \mathcal{S}, \epsilon, \lambda, \rho)$$

where $\mathcal{H}$ denotes a hard specification, $\mathcal{S}$ denotes a soft specification, $\epsilon \in [0,1] \cap \mathbb{Q}$ is the error probability, $\lambda \in [0,1] \cap \mathbb{Q}$ is the lower bound on probability and $\rho \in [0,1] \cap \mathbb{Q}$ is the upper bound on probability.

Definition 36 (Improvising Distributionfor DFTA)**.** *Given a Control Improvisation problem $C = (\mathcal{H}, \mathcal{S}, \epsilon, \lambda, \rho)$, where all variables are as described above, an **improvising distribution** D is a probability distribution over the set I of improvisations (i.e., accepted trees) such that:*

1. ***Hard constraint:*** $\Pr[t \in I \mid t \leftarrow D] = 1$
2. ***Soft constraint:*** $\Pr[t \in A \mid t \leftarrow D] \geq 1 - \epsilon$
3. ***Randomness:*** $\lambda \leq D(t) \leq \rho$ *for all* $t \in I$

There is no need to modify the constraints for the tree based scenario, since they depend solely on inclusion in sets and probability bounds, not on the underlying structure of the elements in I or A. Thus, the same constraints apply regardless of whether we generate words or trees.

Example 13 (Running example DFTA)**.** *We continue with the sets and transition rules defined in the previous example. In this example, we define a probability distribution D over the set I. We use the previously defined specifications $\mathcal{H}$ and $\mathcal{S}$, with parameters $\epsilon = 0.3$, $\lambda = 0.01$, and $\rho = 0.1$.*
Suppose we want the probability of trees that include the function symbol g to be at least twice as high as those that do not. Additionally, to show that we do not require a uniform distribution, we add to the hard specification the requirement that the total probability of trees containing two instances of g must be exactly 0.06.

Looking at the trees in the set I, we construct the distribution D as follows: The first 4 trees (those without g) each get a probability of 0.03*. The next 8 trees (those with one instance of g) each get a probability of* 0.08*. The final 4 trees (those with two instances of g) each get a probability of* 0.06*.*
This distribution satisfies all required conditions:

- *Requirement 1: The total probability of improvisations is* $(0.03 \times 4 + 0.06 \times 4 + 0.08 \times 8 = 1$*, which satisfies the requirement.*
- *Requirement 2: The total probability of inadmissible improvisations is* $0.03 + 0.08 + 0.08 + 0.06 = 0.25$*, which is within the allowed limit of* $\epsilon = 0.03$*.*
- *Requirement 3: Each improvisation has a probability of at least* 0.01 *and at most* 0.1*.*

Definition 37 (Feasible)**.** *The tuple $C = (\mathcal{H}, \mathcal{S}, \epsilon, \lambda, \rho)$ is **feasible** if there exists an improvising distribution D that satisfies the constraints above.*

Since the definition of D is structurally identical, the notion of feasibility remains unchanged.

Definition 38 (Control Improvisation Problem for DFTA)**.** *A **Control Improvisation problem** is the task of determining whether a given tuple $C = (\mathcal{H}, \mathcal{S}, \epsilon, \lambda, \rho)$ is feasible, and if so, constructing an improviser.*

Once feasibility is established, the next step is to generate trees according to such a distribution.

Definition 39 (Improviser Algorithm for DFTA)**.** *An **improviser** is an algorithm that generates trees according to the improvising distribution D for a feasible Control Improvisation problem C.*

Theorem 3 (Feasibility of Control Improvisation for DFTA)**.** *For any Control Improvisation problem $C = (\mathcal{H}, \mathcal{S}, \epsilon, \lambda, \rho)$, the following conditions are equivalent:*

1. *C is feasible.*
2. *The following inequalities hold:*
   - *(a)* $1/\rho \leq |I| \leq 1/\lambda$
   - *(b)* $(1 - \epsilon)/\rho \leq |A|$
   - *(c)* $|I| - |A| \leq \epsilon/\lambda$
3. *There exists an improviser for C.*

Since the construction of the sets I and A, as well as the process of generating trees using DFTA, does not interfere with any assumptions in the original proof of Theorem 1, the proof for feasibility carries over directly from the string based framework.

3.3 Algorithms for Feasible Control Improvisation for DFTA

Having established that feasible control improvisation can be achieved through uniform generation while satisfying both hard and soft constraints, we now turn to concrete algorithmic realizations of this idea for tree structured inputs. Specifically, we present two approaches for constructing uniform improvisers: one for TDDFTA and another for BUDFTA. Both algorithms generate trees that satisfy the hard and soft constraints of the control improvisation problem, while ensuring uniform distribution within the admissible and inadmissible subsets. These methods a solutions for feasible control improvisation over regular tree languages.

3.3.1 Uniform Improviser for TDDFTA

We describe a method for uniformly sampling trees from a TDDFTA, structured to enforce the required probabilistic constraints through recursive rule weighting and selection.
We make an algorithm that operates in two main phases: a preprocessing phase for counting and a generation phase for sampling. In the first phase, it computes for each state in the given TDDFTA the total number of trees obtainable from that state. This is done recursively using depth-first exploration. For every rule of the form $q(f(x_1, \dots, x_n) \to f(q_1(x_1), \dots, q_n(x_n))$, the number of trees it can generate is calculated as the product of the number of trees derivable from each of the substates q_1 through q_n. Summing over all such rules for a state q gives the total count of derivable trees from that state. These counts are memorized to avoid redundant computation.
We write this computation in a formal way using the recursive shape to define inductively the way we define the base step for Count and the inductive step for Count respectively as:

Definition 40 (Count for TDDFTA)**.** *The number of trees that derive a state q, denoted* Count(q)*, is defined inductively as:*

$$Count(q) = |\{q(u) \to u \in \Delta \mid u \in \mathcal{F}_0\}| + \sum_{\substack{q(f(x_1,\dots,x_n)) \to f(q_1(x_1),\dots,q_n(x_n)) \in \Delta \\ n \geq 1}} \prod_{i=1}^{n} \mathrm{Count}(q_i)$$

And with this we can make the following statement:

Lemma 3. *Count(q) for TDDFTA equals the cardinality of the language recognized from state q. Thus:*

$$Count(q) = |[\![q]\!]|$$

Algorithm 3 Preprocessing Tree Counts in a TDDFTA

```
1: Input: TDDFTA $M = (Q, \Sigma, \Delta, q_0)$
2: Output: Fill array count$[q]$ represented number of trees derivable from
   each state $q$
3: function CountTrees($q$)
4:     if count$[q]$ already computed then
5:         return count$[q]$
6:     end if
7:     total $\leftarrow 0$
8:     for all rules $q \to f(q_1, \ldots, q_n) \in \Delta$ do
9:         product $\leftarrow 1$
10:        for $i = 1$ to $n$ do
11:            product $\leftarrow$ product $\times$ CountTrees($q_i$)
12:        end for
13:        total $\leftarrow$ total $+$ product
14:    end for
15:    count$[q] \leftarrow$ total
16:    return total
17: end function
18: CountTrees($q_0$)
19: Return count
```

Algorithm 4 Uniform Tree Sampling from TDDFTA

```
1: Input: TDDFTA $M = (Q, \Sigma, \Delta, q_0)$, precomputed count$[q]$
2: Output: Term $t \in L(M)$
3: function Sample($q$)
4:     Let rules $\leftarrow \{q \to f(q_1, \ldots, q_n) \in \Delta\}$
5:     For each rule $r \in$ rules, compute:

           $$\texttt{weight}[r] \leftarrow \prod_{i=1}^{n} \texttt{count}[q_i]$$

6:     Choose rule $r$ with probability proportional to weight$[r]$
7:     Let $r = q \to f(q_1, \ldots, q_n)$
8:     for $i = 1$ to $n$ do
9:         $t_i \leftarrow$ Sample($q_i$)
10:    end for
11:    return term $f(t_1, \ldots, t_n)$
12: end function
13: Return Sample($q_0$)
```

Given the inductive definition of Count, which specifies the number of trees derivable from each state, and lemma 3 that establishes $Count(q) = |[\![q]\!]|$, we use these values to guide the generation phase of the algorithm. Starting from the initial state, the algorithm selects a rewrite rule with probability proportional to the number of trees it can generate. Computed as the product of the counts of its subtrees, according to the definition of Count. This selection process goes recursively through the tree structure. Since each derivable tree contributes equally to the total count and rules are selected proportionally, the algorithm samples uniformly from $[\![q]\!]$, and in particular from $[\![q_0]\!]$, the language recognized by the automaton.

To implement the soft constraint, we treat the admissible set A and the inadmissible set $I \setminus A$ as being represented by two separate TDDFTA, M_1 and M_2, respectively. The overall generation process performs a biased coin flip at the start: a tree is sampled uniformly from M_1 with probability $1 - \epsilon$ or a tree is sampled uniformly from M_2 with probability ϵ. This corresponds to assigning a multiplicative weight of $1 - \epsilon$ to each tree in A, and a weight of ϵ to each tree in $I \setminus A$.

Since tree sampling within each automaton is uniform (due to the recursive rule weighting from the Count definition), and the selection between M_1 and M_2 is independent and probabilistically fixed, the resulting process respects the soft constraint: every tree in A is generated with equal probability, scaled by $1 - \epsilon$. And every tree in $I \setminus A$ is likewise generated uniformly, scaled by ϵ. This guarantees that the output distribution satisfies both the randomness and constraint requirements of the Control Improvisation framework.

Theorem 4. *Assume that the admissible set $I \setminus A$ can be represented by a TDDFTA. Then, the construction described above, including the handling of the soft constraint, together with Algorithms 3 and 4, yields an improviser for feasible Control Improvisation problems using TDDFTAs.*

Proof. By construction of the algorithm, we can conclude that the constraints for an improviser are satisfied:

- **Hard Constraint:** Trees are always sampled from the set $I = A \cup (I \setminus A)$, as established in the proof of Theorem 1 and applied again in Theorem 3. In the context of the control improvisation problem, the sets I and $I \setminus A$ can be constructed using automata. Specifically, $I = L(\mathcal{H})$, where $\mathcal{H}$ is a TDDFTA, and $I \setminus A$ can be constructed as described by the theorem.
- **Soft Constraint:** The trees from A are generated with probability of at least $1 - \epsilon$.
- **Randomness:** Each tree in A and $I \setminus A$ is generated with an equal probability within its set. This is due to the inductive use of uniform

sampling, ensuring uniformness in every step, by Lemma 3 and the definition of Count for TDDFTA. This guarantees that the probability of generation is always between λ and ρ.

Thus, the algorithm is a valid improviser for a feasible Control Improvisation problem over TDDFTA. $\square$

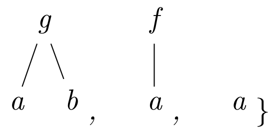
To illustrate an instance that satisfies the conditions of the theorem, we present the following example.

Example 14. *We illustrate our approach with a simple example. Suppose the hard specification $\mathcal{H}$ is given by the following TDDFTA.*
Let $\mathcal{A} = \big(Q = \{q_0, q_1, q_2\}, \mathcal{F} = \{f(), g(,), a, b\}, q_0, \Delta\big)$, where the transitions are:

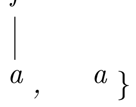
$$\Delta = \begin{Bmatrix} q_0(g(x_1, x_2)) \to g(q_1(x_1), q_2(x_2)), \\ q_0(f(x)) \to f(q_1(x)), \\ q_0(a) \to a, \\ q_1(a) \to a, \\ q_2(b) \to b. \end{Bmatrix}$$

The soft specification $\mathcal{S}$ states that no trees should contain functions symbols of arity greater than or equal to 2. We use an error probability parameter ϵ to allow a relaxation of this rule.

From $\mathcal{H}$, the set of improvisations is:
$I = \{$



And the admissible set under the soft constraint is:
$A = \{$



We construct two TDDFTA representing A and $I \setminus A$ respectively:
We give the automaton for the admissible set A:
$\mathcal{A}_1 = (Q_1 = \{r_0, r_1\},\ \mathcal{F}_1 = \{f(), a\},\ r_0,\ \Delta_1)$, *with transitions:*

$$\Delta_1 = \begin{Bmatrix} r_0(f(x)) \to f(r_1(x)), \\ r_1(a) \to a, \\ r_0(a) \to a. \end{Bmatrix}$$

We give he automaton for the inadmissible set $I \setminus A$:
$\mathcal{A}_2 = (Q_2 = \{p_0, p_1, p_2\},\ \mathcal{F}_2 = \{g(), a, b\},\ p_0$ *initial,* $\Delta_2)$, *with transitions:*

$$\Delta_2 = \begin{Bmatrix} p_0(g(x_1, x_2)) \to g(p_1(x_1), p_2(x_2)), \\ p_1(a) \to a, \\ p_2(b) \to b. \end{Bmatrix}$$

Using Algorithm 3 to compute the counts of trees derivable from each state, we find:
$Count(r_0) = 1 + Count(r_1),\ Count(r_1) = 1,\ Thus\ Count(r_0) = 1 + 1 = 2,$
$Count(p_0) = Count(p_1) \times Count(p_2),\ Count(p_1) = 1,\ Count(p_2) = 1,\ Thus\ Count(p_0) = 1 \times 1 = 1$

Using algorithm 4 we know that the trees from $\mathcal{A}_1$ have a probability of being chosen of 0.5 and the trees from $\mathcal{A}_2$ have a probability of 1. And applying the errorbound from the soft specification: Trees from $\mathcal{A}_1$ are chosen with probability $1-\epsilon$*, and trees from $\mathcal{A}_2$ are chosen with probability ϵ.*
Within each automaton, trees are sampled uniformly. Thus, each tree in A has probability $\frac{1-\epsilon}{2}$,and the unique tree in $I \setminus A$ has probability ϵ.
Overall, the probability distribution over all improvisations sums to 1:

$$2 \times \frac{1-\epsilon}{2} + 1 \times \epsilon = (1-\epsilon) + \epsilon = 1.$$

This construction ensures uniform sampling within each set, with the soft constraint encoded as a biased coin flip between the two automata, controlled by ϵ.

Note that the running example introduced earlier does not satisfy the requirements of the theorem above, as A cannot be expressed using a TDDFTA. We elaborate on this limitation in the extended discussion of the running example below.

Example 15 (Running Example: TDDFTA Limitation)**.** *Using the previous example, we cannot split the TDDFTA representation of the hard specification $\mathcal{H}$ into a TDDFTA representation of A and a representation of $I \setminus A$. This is due to the limited expressiveness of TDDFTA: they can only represent path-closed tree languages and therefore cannot enforce global structural constraints such as 'at most one occurrence of b' in a tree.*
Our language A includes terms such as $f(g(a), b)$, which induce paths like $f_1 g_1 a$, $f_2 b$, and so on. Therefore, the path language $\pi(L)$ includes strings like $f_1 g_1 b$, $f_2 a$, and $f_1 a$. Now consider the term $t = f(g(b), g(b))$. This term has the path set $\pi(t) = \{f_1 g_1 b, f_2 g_1 b\}$, and each of these paths appears in $\pi(L)$, so $\pi(t) \subseteq \pi(L)$, implying $t \in \mathrm{pathclosure}(L)$.

However, $t \notin L$ because it violates the 'at most one b' constraint. Therefore, $L \neq \text{pathclosure}(L)$, meaning L is not path-closed and thus cannot be represented by a TDDFTA.

In summary, we developed a method for uniformly sampling from tree languages recognized by TDDFTAs, and extended it to handle the constraints of CI. Our approach combines a recursive counting procedure with probabilistic rule selection to ensure uniform generation of terms. We showed how to incorporate a soft constraint by constructing separate automata for admissible and inadmissible trees and sampling between them using a weighted strategy. This construction guarantees that the resulting distribution is uniform within each subset and satisfies both the randomness and constraint conditions of the improvisation framework.

3.3.2 Improviser for BUDFTA

We now present a Bottom-Up counterpart to the previous section's TDDFTA based algorithm. This version, for BUDFTA, likewise ensures that all generated trees satisfy the hard constraint, respect the soft constraint with probability $1-\epsilon$, and are sampled uniformly within admissible and inadmissible groups.
We make an algorithm that also proceeds in two stages: preprocessing and sampling. In the preprocessing phase we start with the rules with constants and then go upward through the automaton. For each rule of the form $f(q_1(x_1), \ldots, q_n(x_n)) \to q(f(x_1, \ldots, x_n))$, the total number of trees it adds to q is the product of the counts from each of the child states q_1 through q_n. We sum these outcomes iteratively until an accepting state is reached.
We write this computation in a formal way using the recursive shape to define inductively the way we define the base step for Count and the inductive step for Count respectively as:

Definition 41 (Count for BUDFTA)**.** *The number of trees that derive a state q, denoted $Count(q)$, is defined inductively as:*

$$Count(q) = |\{f \to q(f) \in \Delta \mid f \in \mathcal{F}_0\}| + \sum_{\substack{f(q_1(x_1),\ldots,q_n(x_n)) \to q(f(x_1,\ldots x_n)) \in \Delta \\ n \geq 1}} \prod_{i=1}^{n} \text{Count}(q_i(x_i))$$

Lemma 4. *$Count(q)$ for BUDFTA equals the cardinality of the language recognized from state q. Thus:*

$$Count(q) = |[\![q]\!]|$$

Algorithm 5 Preprocessing Tree Counts in a BUDFTA

```
1: Input: BUDFTA $M = (Q, \Sigma, \Delta, F)$
2: Output: Map $\texttt{count}[s]$ of number of trees generating state $s$
3: for all function symbols $f \in \Sigma$ of arity 0 do
4:     for all rules $f \to q \in \Delta$ do
5:         $\texttt{count}[q] \leftarrow \texttt{count}[q] + 1$
6:     end for
7: end for
8: repeat
9:     for all rules $f(q_1, \dots, q_n) \to q \in \Delta$ do
10:        $\texttt{product} \leftarrow \prod_{i=1}^{n} \texttt{count}[q_i]$
11:        $\texttt{count}[q] \leftarrow \texttt{count}[q] + \texttt{product}$
12:    end for
13: until no updates to any $\texttt{count}[q]$
14: Return $\texttt{count}$
```

Algorithm 6 Uniform Tree Sampling from BUDFTA

```
1: Input: BUDFTA $M = (Q, \Sigma, \Delta, F)$, precomputed $\texttt{count}[q]$
2: Output: Term $t \in L(M)$
3: function SAMPLE($q$)
4:     Let $\texttt{rules} \leftarrow \{f(q_1, \dots, q_n) \to q \in \Delta\}$
5:     For each rule $r \in \texttt{rules}$, compute:
           $$\texttt{weight}[r] \leftarrow \prod_{i=1}^{n} \texttt{count}[q_i]$$
6:     Choose rule $r$ with probability proportional to $\texttt{weight}[r]$
7:     Let $r = f(q_1, \dots, q_n) \to q$
8:     for $i = 1$ to $n$ do
9:         $t_i \leftarrow$ SAMPLE($q_i$)
10:    end for
11:    return term $f(t_1, \dots, t_n)$
12: end function
13: Choose a final state $q_f \in F$ with probability proportional to $\texttt{count}[q_f]$
14: Return SAMPLE($q_f$)
```

Using the inductive definition of $Count(q)$, the algorithm begins by selecting a final state $q_f \in F$ with probability proportional to $Count(q_f)$, *i.e.*, the number of trees that derive q_f. It then recursively constructs a tree by selecting a rule of the form $f(q_1, \dots, q_n) \to q_f$, with probability proportional to the product of $Count(q_i(x_i))$, as dictated by the recursive structure. By Lemma 4, $Count(q) = |[\![q]\!]|$, so the weights used during rule selection reflect the exact number of trees each rule configuration contributes. This guarantees that each tree in $\llbracket q_f \rrbracket$ is sampled with equal probability, ensuring uniform sampling across the language recognized by the automaton.

To implement the soft constraint, the algorithm uses two separate BUDFTA: M_1, representing the admissible set A, and M_2, representing the inadmissible set $I \setminus A$. A biased coin flip is simulated at the root: with probability $1-\epsilon$, a tree is sampled uniformly from M_1, and with probability ϵ, from M_2. This results in a uniform sampler over A scaled by a factor of $1-\epsilon$, and over $I \setminus A$ scaled by ϵ.

Theorem 5. *Thee construction described above, including the handling of the soft constraint, together with Algorithms 5 and 6, yields an improviser for feasible Control Improvisation problems using BUDFTA.*

Proof. Since sampling within each automaton is uniform, and the probabilistic selection between M_1 and M_2 satisfies the soft constraint, the overall tree generation meets all conditions:

- **Hard Constraint:** Trees are always sampled from the set $I = A \cup (I \setminus A)$, as established in the proof of Theorem 1 and applied again in Theorem 3. In the context of the control improvisation problem, the sets I and A can be constructed using BUDFTAs, since they recognize regular tree languages.
- **Soft Constraint:** The trees from A are selected with at least a total probability $1-\epsilon$.
- **Randomness:** Within A and $I \setminus A$, all trees are selected uniformly. This is due to the inductive use of uniform sampling, ensuring uniformness in every step, by Lemma 4 and the definition of Count for BUDFTA. So the probability of generation lies between λ and ρ.

Therefore, this algorithm defines a valid improviser for feasible Control Improvisation problems over BUDFTA. □

Example 16 (Running example BUDFTA)**.** *We now illustrate Algorithm 5 and 6 with an example. Suppose we have a BUDFTA for the language $\mathcal{H}$ from the previous example, and we want to split it into two parts: one representing the admissible set A, and the other representing the inadmissible set $I \setminus A$.*
To ensure that no tree in A contains the constant b more than once, we must encode this constraint explicitly. This requires explicitly defining all admissible trees to ensure the constraint is respected.
Let $\mathcal{F}_A = \{a, b, g(), f(,)\}$, the state set be $Q_A = \{q_a, q_b, q_1, q_2, q_f\}$, with final states $Q_{A_f} = \{q_f\}$. Define the BUDFTA $\mathcal{A}_A = (Q_A, \mathcal{F}_A, Q_{A_f}, \Delta_A)$, with variables $\chi_A = \{x, y\}$, and transition rules:

$$\Delta_A = \begin{Bmatrix} a \to q_a(a), \\ b \to q_b(b), \\ g(q_a(x)) \to q_1(g(x)), \\ g(q_b(x)) \to q_2(g(x)), \\ f(q_a(x), q_a(y)) \to q_f(f(x,y)), \\ f(q_a(x), q_b(y)) \to q_f(f(x,y)), \\ f(q_b(x), q_a(y)) \to q_f(f(x,y)), \\ f(q_1(x), q_a(y)) \to q_f(f(x,y)), \\ f(q_a(x), q_1(y)) \to q_f(f(x,y)), \\ f(q_1(x), q_b(y)) \to q_f(f(x,y)), \\ f(q_b(x), q_1(y)) \to q_f(f(x,y)), \\ f(q_2(x), q_a(y)) \to q_f(f(x,y)), \\ f(q_a(x), q_2(y)) \to q_f(f(x,y)), \\ f(q_1(x), q_1(y)) \to q_f(f(x,y)), \\ f(q_1(x), q_2(y)) \to q_f(f(x,y)), \\ f(q_2(x), q_1(y)) \to q_f(f(x,y)) \end{Bmatrix}$$

Applying Algorithm 5 to compute the count for each state:
$Count(q_f) = 0 + Count(q_a(x)) \times Count(q_a(y)) + Count(q_a(x)) \times Count(q_b(y)) + Count(q_b(x)) \times Count(q_a(y)) + Count(q_1(x)) \times Count(q_a(y)) + Count(q_a(x)) \times Count(q_1(y)) + Count(q_1(x)) \times Count(q_b(y)) + Count(q_2(x)) \times Count(q_a(y)) + Count(q_a(x)) \times Count(q_2(y)) + Count(q_b(x)) \times Count(q_1(y)) + Count(q_1(x)) \times Count(q_1(y)) + Count(q_1(x)) \times Count(q_2(y)) + Count(q_2(x)) \times Count(q_1(y))$,
$Count(q_1) = 0 + Count(q_a)$,
$Count(q_2) = 0 + Count(q_b)$,
$Count(q_a) = 1$,
$Count(q_b) = 1$,
Thus $Count(q_f) = 0 + 12 \times (1 \times 1) = 12$
The inadmissible set $I \setminus A$ *consists of the remaining trees that do contain two occurrences of* b*. Since only four such trees exist and their forms do not overlap, we list all rules directly.*
Let $\mathcal{F}_{I\setminus A} = \{b, g(), f(,)\}$*, the state set be* $Q_{I\setminus A} = \{q_0, q_1, q_f\}$*, with final state* $Q_{I\setminus A_f} = \{q_f\}$*. Define the BUDFTA* $\mathcal{A}_{I\setminus A} = (Q_{I\setminus A}, \mathcal{F}_{I\setminus A}, Q_{I\setminus A_f}, \Delta_{I\setminus A})$*, with* $\chi_{I\setminus A} = \{x, y\}$*, and transition rules:*

$$\Delta_{I\setminus A} = \left\{ \begin{array}{l} b \to q_0(b), \\ g(q_0(x)) \to q_1(g(x)), \\ f(q_0(x), q_0(y)) \to q_f(f(x,y)), \\ f(q_0(x), q_1(y)) \to q_f(f(x,y)), \\ f(q_1(x), q_0(y)) \to q_f(f(x,y)), \\ f(q_1(x), q_1(y)) \to q_f(f(x,y)) \end{array} \right\}$$

Applying Algorithm 5 again:
$Count(q_f) = 0 + Count(q_0(x)) \times Count(q_0(y)) + Count(q_0(x)) \times Count(q_1(y)) + Count(q_1(x)) \times Count(q_0(y)) + Count(q_1(x)) \times Count(q_1(y))$,
$Count(q_1) = Count(q_0)$,
$Count(q_0) = 1$,
Thus $Count(q_f) = 0 + 4 \times (1 \times 1) = 4$
When generating a tree, the algorithm samples from $\mathcal{A}_A$ *with probability* $1-\epsilon$*, and from* $\mathcal{A}_{I\setminus A}$ *with probability* ϵ*. Since the sampling within each automaton is uniform, the final probabilities are:*

- *For each tree in* $A : \frac{1-\epsilon}{12}$
- *For each tree in* $I \setminus A : \frac{\epsilon}{4}$

The total probability sums to 1, by $12 \cdot \frac{1-\epsilon}{12} + 4 \cdot \frac{\epsilon}{4} = 1 - \epsilon + \epsilon = 1$.
This construction ensures that each tree in A *is equally likely, with total probability* $1-\epsilon$*. Each tree in* $I \setminus A$ *is equally likely, with total probability* ϵ*. The overall sampling remains uniform within each partition and respects the soft constraint.*

To conclude, we developed a sampling algorithm for BUDFTA that satisfies the requirements of the improviser of definition 36. Our method ensures that all generated trees meet the hard constraint, and that the soft constraint is respected with high probability. By computing the number of trees for each state and using these counts during sampling, we guarantee that each tree is selected uniformly. We also showed how to handle the soft constraint by switching between two automata: one for admissible trees and one for inadmissible trees. This approach extends the improvisation framework to bottom-up tree automata, while keeping the guarantees of uniformity and feasibility.

Chapter 4

Related Work

Control Improvisation was first formalized by Fremont et al. [6], who introduced efficient algorithms for generating strings for finite problems, while ensuring that the output distribution approximately satisfies user-specified probabilistic bounds (e.g., uniformity or diversity), within a tolerable error margin. Earlier work by Valle et al. [11] applied similar principles to music generation, using automata to constrain improvisation.

Subsequent developments expanded CI to reactive settings, where the improviser must interact with an environment [5], and to cost- and diversity-aware scenarios [7]. However, these frameworks are all restricted to flat string representations, with no extension to hierarchical structures such as trees.

Tree automata are widely used in verification and XML document validation. For example, Cohen et al. [3] show how deterministic tree automata can be efficiently evaluated over probabilistic XML data. While their focus is on query evaluation, it highlights the potential of automata-based methods for reasoning under uncertainty in structured domains.

Random generation techniques for trees do exist. Chen et al. [2] review fuzzing methods that generate structured inputs, including those based on grammars, but these typically rely on heuristics and offer no guarantees of uniformity or constraint satisfaction. Rodionov and Choo [10] present efficient algorithms for generating random trees under structural constraints, such as bounded node degrees, though their work focuses on network modeling rather than formal language generation.

Other work focuses on the generation of trees with formal guarantees. Hanneforth et al. [8] study the random generation of nondeterministic finite-state tree automata, though their goal is to generate automata themselves, rather than tree instances. Bacher et al. [1] present efficient uniform samplers for binary and unary-binary trees using holonomic equations, offering practical tools for uniform generation but without support for constraint-guided randomness or general regular tree languages.

Our approach fills a gap by extending randomized synthesis techniques to tree-structured data. Specifically, we show that uniformly sampling trees accepted by a regular tree automaton is feasible in randomized polynomial time using dynamic programming. This forms the basis for applying CI in tree domains, where the challenge lies in ensuring both structural validity and controlled randomness.

While there has been work on the random generation of trees and on tree automata, we are not aware of any existing research that directly combines the control improvisation framework with regular tree automata for structured data generation. This thesis aims to explore that intersection by extending CI to regular tree languages, enabling structured, constraint-guided generation of tree-shaped data under soft randomness bounds. This represents a potential new application of both CI and tree automata theory.

Chapter 5

Conclusions

This thesis extends the Control Improvisation framework to tree structures using deterministic finite tree automata. The research demonstrates that the principles of balancing hard and soft constraints with probabilistic guarantees can be adapted to tree-structured data. Two key algorithms were developed: a uniform improviser for TDDFTAs and another for BUDFTAs, both ensuring feasibility and correctness. The findings highlight the capabilities of tree automata within random generation, while also emphasizing the limitations of specific models. Since TDDFTAs are restricted to path-closed languages, the top-down improvisation algorithm does not apply to all finite tree languages. In contrast, the bottom-up approach supports any regular finite tree language, making it the more generally applicable method.
Future work could explore relaxing the determinism requirement to allow non-deterministic tree automata, potentially increasing the expressiveness of the system. In addition, extending the framework to support infinite tree languages or unbounded height could make CI applicable to a broader class of problems, although this may require new strategies for maintaining uniformity and feasibility guarantees. Another promising direction is the development of a visualizer for tree automata, driven by the improviser to produce probabilistic visual outputs—similar to the illustrative example introduced in the preliminaries. Such a tool could aid in understanding the structure and variability of generated trees, and serve as an educational or debugging aid for users working with tree-based models.

Bibliography

[1] Axel Bacher, Olivier Bodini, and Alice Jacquot. Efficient random sampling of binary and unary-binary trees via holonomic equations. *CoRR*, abs/1401.1140, 2014.

[2] Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. A systematic review of fuzzing techniques. *Computers Security*, 75:118–137, 2018.

[3] Sara Cohen, Benny Kimelfeld, and Yehoshua Sagiv. Running tree automata on probabilistic xml. In *Proceedings of the Twenty-Eighth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '09, page 227–236, New York, NY, USA, 2009. Association for Computing Machinery.

[4] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications*. 2008.

[5] Daniel Fremont and Sanjit A. Seshia. Reactive control improvisation. In *30th International Conference on Computer Aided Verification (CAV)*, 2018.

[6] Daniel J. Fremont, Alexandre Donzé, and Sanjit A. Seshia. Control improvisation. *CoRR*, abs/1704.06319, 2017.

[7] Andreas Gittis, Eric Vin, and Daniel J. Fremont. Randomized synthesis for diversity and cost constraints with control improvisation, 2022.

[8] Thomas Hanneforth, Andreas Maletti, and Daniel Quernheim. Random generation of nondeterministic finite-state tree automata. *Electronic Proceedings in Theoretical Computer Science*, 134:11–16, November 2013.

[9] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 2nd edition*, volume 32. ACM Press, New York, NY, USA, 2001.

[10] Alexey S. Rodionov and Hyunseung Choo. On generating random network structures: Trees. In Peter M. A. Sloot, David Abramson, Alexander V. Bogdanov, Yuriy E. Gorbachev, Jack J. Dongarra, and Albert Y. Zomaya, editors, *Computational Science — ICCS 2003*, pages 879–887, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[11] Rafael Valle, Alexandre Donze, Ilge Akkaya, Sophie Libkind, Sanjit Seshia, and David Wessel. Machine improvisation with formal specifications. *ICMCSMC14*, 2014.