Formal Explainability for Artificial Intelligence in Dynamic Environments

Jime Cuartas Granada

Supervisors:
Alexey Ignatiev
Peter J. Stuckey
Julian Gutierrez

A Progress Review report at
Monash University in 2026
School of Information Technology

Abstract

In dynamic environments, a goal of Artificial Intelligence (AI) is to build intelligent agents capable of addressing sequential decision-making settings. In this context, there are two important challenges for humans to understand decisions made by agents: (1) the sequential decisions are connected, (2) the environment can play a role in the outcome and (3) the agents may use opaque black-box models for each decision.

Despite the success of AI in sequential decision-making (e.g. Reinforcement Learning), the lack of transparency in understanding their decisions can make the agents hard to validate. To address the need for transparency, there are efforts to develop Explainable Artificial Intelligence (XAI). XAI is a set of methods designed to make AI models easier to comprehend. Despite the importance of Explainable Reinforcement Learning in developing trustworthy intelligent agents, there are gaps in current research to make sequential decision-making explainable.

This project proposes to explain sequential decision-making using formal reasoning. To achieve this goal, the proposal focuses on (1) Formal Explainability for Finite Automata, to address sequential actions in deterministic environments, (2) Formal Explainability for Pushdown Automata, to address sequential actions with memory, and (3) Formal Explainability for stochastic models, where outcomes are subject to environmental uncertainty.

Contents

Chapter 1

Introduction

The deployment of Artificial Intelligence (AI) algorithms has necessitated the need for eXplainability AI (XAI) methods in order to ensure transparency, trust, and accountability. While much of the field has focused on heuristic explanations for opaque models, there is an interest in formal approaches that provide rigorous guarantees about the explanations generated [1, 2].

A fundamental challenge in dynamic environments is explaining sequential decision-making. To address this, we model these processes using Automata, which provide a symbolic and tractable representation of sequential decision functions. This approach allows us to generate formal explanations, why a specific sequence of actions leads to a particular outcome. Automata are widely used in software verification [3], design of communication protocols [4],and sintax parsing in compiler [5]. When a computational model, such as a Finite Automaton (FA) or a Pushdown Automaton (PDA), accepts or rejects an input string, the reasoning behind that decision can be non-trivial. Understanding why a specific input was accepted or rejected is crucial for debugging, and refinement purposes.

This research project investigates the formalization of explanations for sequential decision-making. Having addressed an approach to deliver formal explanations for Finite Automata (FA) in the first stage of this research, and submitting it to a ICALP 2026. I now move to address explanations for Context-Free Languages (CFG) / Pushdown Automata (PDA) decisions.

Current literature focuses on the *performance* of parsing rather than the *interpretability* of the decision. Modern parsing algorithms, such as Tree-sitter [6] and ANTLR 4 [7], utilise sophisticated incremental parsing and adaptive LL(*) algorithms to ensure low-latency feedback for integrated development environments (IDEs). While these methods are efficient at error recovery, often scaling linearly with input size, they treat the decision process as the main objective.

1.1 Refined scope - problem statement

While standard XAI focuses on feature attribution in classifiers, the ”features” in formal languages are sequential and structural. Since the confirmation report, the research scope has been refined to address three primary gaps:

- **Research Problem 1 (Completed): Explaining Finite Automata.** How can we provide Formal Explanations for Finite Automata decisions?
  - To Define Formal explanations for understanding the acceptance/rejection of inputs in Finite Automata
  - To develop a method to compute Formal Explanations for Finite Automata
- **Research Problem 2: Explaining Pushdown Automata (PDA).** How can we provide Formal Explanations for Pushdown Automata decisions?
  - To Define Formal explanations for understanding the acceptance/rejection of inputs in Pushdown Automata.
  - To develop a method to compute Formal Explanations for Pushdown Automata.
  - To propose a method to identify the most significant or “most likely” explanations among explanations.
- **Research Problem 3: Explaining Decisions of Sthocastic models.**
  - To Define Formal explanations for understanding the probable outcomes in Stochastic Models.
  - To develop a method to compute Formal Explanations for Stochastic Models (e.g., Probabilistic Finite Automata or Markov Models).
  - To propose a method to identify the most significant or “most likely” explanations among probabilistic paths.

1.2 Contributions - achieved and projected

This research provides both theoretical and practical contributions:

Achieved Contributions:

- Development of a theoretical and practical approach to explain Finite Automata.
- A paper submitted to ICALP 2026 tittled “A Formal Framework for the Explanation of Finite Automata Decisions”

Projected Contributions:

- Explaining Pushdown Automata decisions: (In Progress) Extending the formal explanation framework to PDAs, which recognize context-free languages. This involves developing algorithms to identify the minimal contrastive explanations (CXPs) and Abductive Explanations (AXPs), and quantifying the contribution of specific tokens to the decision (acceptance or rejection).
- Explaining Decisions of Sthocastic models: Extending the formal explanation framework to Stochastic Models. The goal is to provide verifiable explanations able to identify the environmental factors or decision points that lead to a particular outcomes.

Chapter 2

Progress

2.1 Model Proposal: Context-Free Grammar (CFG) Explanations

The research has evolved from the study of Finite Automata (FA) to more expressive computational models. While FA provided a baseline for explaining sequential behaviors, many complex problems require the model to “remember” an arbitrary number of previous inputs in the sequence to determine the validity of subsequent inputs

Consider the abstract language $L = \{a^n b^n \mid n \geq 1\}$, which represents a sequence where every ‘a’ must be matched by a corresponding ‘b’.

- The Limitation: A standard Finite Automaton (FA) possesses no auxiliary memory. Therefore, it cannot count the number of a’s to ensure they match the number of b’s once n exceeds the number of states in the machine.
- The Explanation Failure: If a PDA were used to validate a sequence, it would process inputs. Upon encountering a mismatch (e.g., a^4b^2), it might reject the sequence, but it lacks the structural context to generate a contrastive explanation such as: ”The sequence is invalid because the third or fourth ‘a’ was not closed by a matching ‘b’.” Instead, it can only report a “transition failure” at the specific index, obscuring the root cause of the error.
- **From FA to PDA:** We propose the use of Pushdown Automata (PDAs) to model decision-making processes with memory.
  Unlike FA, the addition of a stack allows the description of more complex languages. Here the challenge is how to explain PDA decisions.

2.2 Motivation: The Gap between Detection and Explanation

One well established applications of Context-Free Grammars is in the design of programming languages and compilers. Compilers are highly efficient at detecting when a sequence of tokens fails to belong to a grammar. However, a fundamental question is: are they able of generating a useful *explanation* for why the failure occurred or how to fix it?

Consider the following C code, where a typo has introduced a double opening bracket {{ in the `for` loop:

```
1 int main(){
2     for(int i=0; i<10; i++){{   // <--- Error: Double bracket
3         printf("hello");
4     }
5 }
```

When compiled (e.g., using GCC or Clang), the parser consumes the input until it reaches <<EOF>>(end-of-file), finding only then that EOF was not expected, instead there is an incomplete structure, the token '}' is expected before <<EOF>>. The resulting error message is:

```
error: expected '}' at end of input
   5 | }
     |  ^
```

The Explanation: While the compiler's output is correct, the file ended while the stack still contained an open brace that was not closed yet. It is *misleading*.

- **Root Cause:** The compiler points to line 5 (the end of the file) as the location of the error. However, the root cause is located at line 2.
- **Lack of Contrastive Reasoning:** A human (or a formal explainer) would identify that the input is "almost correct". The explanation should not just report a missing symbol at the end, but rather propose a *minimal correction.*

In this case, the minimal correction is not to add a brace at the end, but to remove the redundant opening brace at the loop initialization:

```
1 int main(){
2     for(int i=0; i<10; i++){
3         printf("hello");
4     }
5 }
```

This discrepancy motivates the need for our proposed formal framework. We aim to move beyond isolated decisions to eplained decisions revealing these minimal set of edits (Contrastive Explanations).

2.3 Preliminaries

2.3.1 Pushdown Automata and Context-Free Grammars

To provide a foundation for the proposed explanation extraction methods, standard definitions and notations for Pushdown Automata and Context-Free Grammar are adopted [8, 9].

Definition 2.1 (Pushdown Automaton)**.** A Pushdown Automaton (PDA) extends the capabilities of a Finite Automaton by incorporating an infinite memory stack. A PDA is formally defined as a 7-tuple $\mathcal{A} = (Q, \Sigma, V, \delta, q^0, v^0, F)$, where:

- Q is a finite set of states.
- Σ is the input alphabet, a finite terminal alphabet.
- V is the stack alphabet, a finite nonterminal alphabet that can be pushed onto or popped from the stack.
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times V \to \mathcal{P}(Q \times V^*)$ [1] is the transition fuction. It dictates how the machine transitions between states and modifies the stack based on the current state, input symbol, and the top symbol of the stack.
- $q^0 \in Q$ is the initial state.
- $v^0 \in V$ is the initial pushdownstore symbol.
- $F \subseteq Q$ is the set of accepting states.

A configuration of $\mathcal{A}$ is a triple $c = (q, \gamma, x)$ in $Q \times V^* \times \Sigma^*$. And the automaton moves from c into configuration $c' = (q', \gamma', x')$, denoted as $c \vdash c'$ if:

- $\gamma = v\gamma_1 (v \in V)$ [2] , $x = ax'(a \in \Sigma)$, $\gamma' = m\gamma_1 (m \in V^*)$ and $(q', m) \in \delta(q, a, v)$, namely "a-move";
- or $\gamma = v\gamma_1 (v \in V)$, $x = x'$, $\gamma' = m\gamma_1 (m \in V^*)$ and $(q', m) \in \delta(q, \epsilon, v)$, namely "$\epsilon$-move".

A *word* is accepted by a pushdown automaton if, starting with an empty stack, there is a path through the automaton such that the automaton stops in an accepting state after the entire string has been read. The *language* recognized by a PDA $\mathcal{A}$ is the set of all accepted words, denoted as $L(\mathcal{A})$.

The following PDA recognizes the language of balanced parentheses (a subset of the Dyck language) [3] and is used throughout the document to illustrate the proposed ideas.

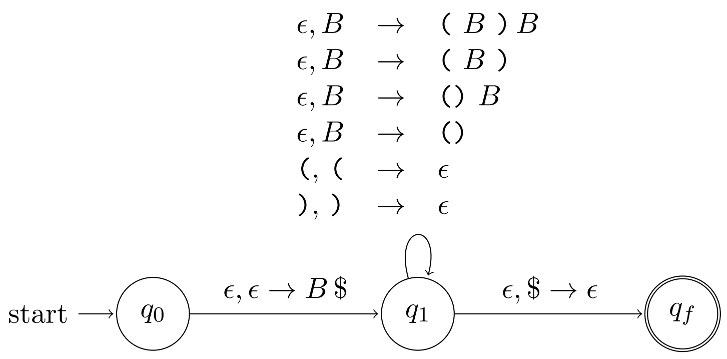
Example 2.1. *Let $\mathcal{A}$ be the PDA that accepts the language generated by G in Example 2.2. The PDA uses a stack to track the depth of nesting, pushing a symbol for every open parenthesis and popping for every closed one. B represents Balanced in G.*

- *States:* $Q = \{q_0, q_1, q_f\}$, $q^0 = q_0$, $F = \{q_f\}$
- *Input Alphabet:* $\Sigma = \{\, (\, ,\,)\, \}$
- *Stack Alphabet:* $V = \{\, (\, ,\,)\, , B, \$\}$, $v^0 = B$

[1]$\mathcal{P}$ denotes the power set (the set of all its subsets). V^* and Σ^* denote the Kleene closures of the stack and input alphabets, respectively, representing the sets of all finite strings formed by those alphabets.

[2]$\gamma_1 \in V^*$ represents the remaining symbols on the stack below the top symbol v. Thus, γ represents the full current stack, formed by the top symbol v (to be processed) and the rest of the stack γ_1.

[3]The Dyck language describes a set of strings with balanced and properly nested brackets (e.g., (), [], {}) [9]. The example focuses solely on non-empty sequences of balanced (and).



$\epsilon, B \to (\ B\ )\ B$
$\epsilon, B \to (\ B\ )$
$\epsilon, B \to ()\ B$
$\epsilon, B \to ()$
$(,\ (\to \epsilon$
$),\ ) \to \epsilon$
start
q_0
$\epsilon, \epsilon \to B\,\$$
q_1
$\epsilon, \$ \to \epsilon$
q_f

FIGURE 2.1: A Pushdown Automaton $\mathcal{A}$ accepting the language of balanced parentheses. The symbol \$ is used as the bottom-of-stack marker.

- *Transitions:*
  1. $\delta(q_0, \epsilon, \epsilon) = \{(q_1, B\,\$)\}$ *(Initialize stack with Start Symbol)*
  2. $\delta(q_1, \epsilon, B) = \{(q_1, (B)B), (q_1, (B)), (q_1, ()B), (q_1, ())\}$ *(Expand B)*
  3. $\delta(q_1, (, () = \{(q_1, \epsilon)\}$ *(Match input '(' with stack '(')*
  4. $\delta(q_1,),)) = \{(q_1, \epsilon)\}$ *(Match input ')' with stack ')')*
  5. $\delta(q_1, \epsilon, \$) = \{(q_f, \epsilon)\}$ *(Accept if bottom marker is reached)*

Figure 2.1 illustrates this PDA. The configuration history of $\mathcal{A}$ for the input ()() is:

$$\begin{aligned}
& (q_0, ()(), \epsilon) \\
\vdash\ & (q_1, ()(), B\$) && \textit{(Initialize)} \\
\vdash\ & (q_1, ()(), ()B\$) && \textit{(Expand } B \to ()B\textit{)} \\
\vdash\ & (q_1,)(),)B\$) && \textit{(Match '(')} \\
\vdash\ & (q_1, (), B\$) && \textit{(Match ')')} \\
\vdash\ & (q_1, (), ()\$) && \textit{(Expand } B \to ()\textit{)} \\
\vdash\ & (q_1,),)\$) && \textit{(Match '(')} \\
\vdash\ & (q_1, \epsilon, \$) && \textit{(Match ')')} \\
\vdash\ & (q_f, \epsilon, \epsilon) && \textit{(Accept)}
\end{aligned} \tag{2.1}$$

Definition 2.2 (Context-Free Grammar)**.** A Context-Free Grammar (CFG) is defined as a 4-tuple $G = (V, \Sigma, R, S)$, where:

- V (Variables/Non-terminals) is a finite set of variables (non-terminal symbols).
- Σ (Terminals) is a finite set of terminal symbols, disjoint from V.
- R is a finite set of production rules of the form $A \to \alpha$, where $A \in V$ describes a variable and $\alpha \in (V \cup \Sigma)^*$ is a string of variables and terminals.
- $S \in V$ is the start variable.

A fundamental equivalence between CFGs and PDAs: a language L is context-free iff there exists a PDA $\mathcal{A}$ such that $L(\mathcal{A}) = L$ [8, 9]. This equivalence allows us to use grammar-based parsing algorithms, such as CYK, to analyze the behavior of PDAs.

Example 2.2. *Let $G = (\{Balanced\}, \{\,(\,,\,)\,\}, R, Balanced)$ be the grammar defined by the following production rules R:*

$$\begin{aligned} Balanced &\to (\,Balanced\,)\,Balanced && \textit{(Rule 1)} \\ Balanced &\to (\,Balanced\,) && \textit{(Rule 2)} \\ Balanced &\to (\,)\,Balanced && \textit{(Rule 3)} \\ Balanced &\to (\,) && \textit{(Rule 4)} \end{aligned} \tag{2.2}$$

This grammar generates the language of properly nested parentheses. For instance, the string $()()$ can be derivated as follows.

$$\begin{aligned} Balanced &\Rightarrow (\ )\,\mathbf{Balanced} && \textit{(Rule 3: parentheses and Balanced)} \\ &\Rightarrow (\ )\ (\ ) && \textit{(Rule 4: Reduce `Balanced' to `()')} \end{aligned} \tag{2.3}$$

To analyze the behavior of the PDA using grammar-based approaches, we must first standardize the grammar structure (e.g. Chomsky Normal Form). This enables the use of efficient parsing algorithms like CYK (Cocke-Younger-Kasami) [10, 11].

2.3.2 Chomsky Normal Form

Parsing algorithms often require the grammar to be in a canonical form to ensure predictable execution complexity.

Definition 2.3 (Chomsky Normal Form)**.** A Context-Free Grammar $G = (V, \Sigma, R, S)$ is in *Chomsky Normal Form* (CNF) [8] if every production rule in R is of one of the following two forms:

- $A \to BC$ (where $A, B, C \in V$)
- $A \to a$ (where $a \in \Sigma$ and $a \neq \epsilon$)

For every CFG G whose languge contains at least one string other than ϵ, then there is a grammar G_1 in Chomsky Normal Form.

Example 2.3. *Consider the grammar G from Example 2.2. We transform G into an equivalent grammar $G' = (V', \Sigma, R', S)$ in CNF.*

Step 1: Terminals to Non-terminals. *To introduce variables L and R for terminals '(' and ')'.*

$$L \to (\quad \textit{and} \quad R \to)$$

Step 2: Binary decomposition. *To rewrite the original rules using new variables and break down productions into binary steps.*

The resulting production rules R' are:

$$\begin{array}{rcll}
\textit{Balanced} & \rightarrow & \textit{Nested Balanced} & \textit{(Represents `(Balanced) Balanced')} \\
 & | & \textit{Unclosed R} & \textit{(Represents `(Balanced)')} \\
 & | & \textit{Pair Balanced} & \textit{(Represents `( ) Balanced')} \\
 & | & \textit{L R} & \textit{(Represents `( )')} \\
\textit{Nested} & \rightarrow & \textit{Unclosed R} & \\
\textit{Unclosed} & \rightarrow & \textit{L Balanced} & \\
\textit{Pair} & \rightarrow & \textit{L R} & \\
L & \rightarrow & \textit{(} & \\
R & \rightarrow & \textit{)} &
\end{array}$$

2.3.3 The CYK Algorithm

The Cocke-Younger-Kasami (CYK) [10, 11] algorithm is a bottom-up parsing method that given a CFG G in CNF determines whether a string w belongs to a language $L(G)$. It operates via dynamic programming, constructing a triangular table where each cell $T[i,j]$ contains the set of non-terminals that can generate the substring of w starting at i and ending at j.

Definition 2.4 (CYK Table Construction)**.** For an input string $w = w_1 w_2 \dots w_n$:

1. **Base Case:** For each $i \in \{1, \dots, n\}$, $T[i,i]$ contains A if there is a rule $A \rightarrow w_i$.
2. **Recursive Step ($j > i$):** $T[i,j]$ contains A if there exists a rule $A \rightarrow BC$ and a split point k ($i \leq k < j$) such that $B \in T[i,k]$ and $C \in T[k+1,j]$.

The string is accepted if starting symbol $S \in T[1,n]$.

Example 2.4. *The Table 2.1 illustrates a CYK Table construction to verify the acceptance of the string w = ()() using the CNF grammar derived in Example 2.3. The top-right cell $T[1,4]$ represents the entire string It contains the Start symbol B (Balanced) because there is a rule 'Balanced $\rightarrow$ Pair Balanced' for a split point $k = 2$ where Pair $\in T[1,2]$ and Balanced $\in T[3,4]$.*

Similarly, the cell $T[1,2]$ contains P (Pair) because there is a rule Pair $\rightarrow$ L R for a split point $k = 1$ ($L \in T[1,1]$ and $R \in T[2,2]$).

$w_1 =$ '(' $\{L\}$	$\{B,P\}$	$\emptyset$	$\{\mathbf{B}\}$
	$w_2 =$ ')' {R}	$\emptyset$	$\emptyset$
		$w_3 =$ '(' {L}	{B,P}
			$w_4 =$ ')' {R}

Table 2.1: CYK Table for w = ()(). The cells $T[i,j]$ correspond to the substring starting at i and ending at j. The diagonal elements contain the input terminals and unary rules able to generate them. **Abbreviations:** B = Balanced, P = Pair

2.3.4 Classification Problems and Formal Explanations

Following XAI literature for classification problems [2, 12], we consider a classifier $\kappa : \mathbb{F} \rightarrow \mathcal{K}$ over a feature space $\mathbb{F} = \prod_{i=1}^{m} \mathbb{D}_i$ and a set of classes K.

For an instance $\mathbf{v} \in \mathbb{F}$ with prediction $\kappa(\mathbf{v}) = \ c \in \ \mathcal{K}$, an *abductive explanation* $\mathcal{X} \subseteq \mathcal{F}$ *sufficient* for the prediction. Formally, $\mathcal{X}$ is defined as:

$$\forall(\mathbf{x} \in \mathbb{F}).\ \left[\bigwedge\nolimits_{i\in\mathcal{X}} (x_i = v_i)\right] \rightarrow (\kappa(\mathbf{x}) = c) \tag{2.4}$$

Similarly, a *contrastive explanation* (CXp) is a minimal subset of features $\mathcal{Y} \subseteq \mathcal{F}$ that, if allowed to change, enables the prediction's alteration. Formally, a contrastive explanation $\mathcal{Y}$ is defined as follows:

$$\exists(\mathbf{x} \in \mathbb{F}).\ \left[\bigwedge\nolimits_{j\notin\mathcal{Y}} (x_j = v_j)\right] \wedge (\kappa(\mathbf{x}) \neq c) \tag{2.5}$$

Observe that abductive explanations are used to explain *why* a prediction is made by the classifier κ for a given instance, while contrastive explanations can be seen to answer *why not* another prediction is made by κ. Alternatively, CXps can be seen as answering *how* the predication can be changed.

Importantly, abductive and contrastive explanations are known to enjoy a minimal hitting set duality relationship [13]. Given $\kappa(\mathbf{v}) = \ c$, let $\mathbb{A}_\mathbf{v}$ be the complete set of AXps and $\mathbb{C}_\mathbf{v}$ be the complete set of CXps for this prediction. Then each AXp $\mathcal{X} \in \mathbb{A}_\mathbf{v}$ is a minimal hitting set of $\mathbb{C}_\mathbf{v}$ and, vice versa, each CXp $\mathcal{Y} \in \mathbb{C}_\mathbf{v}$ is a minimal hitting set of $\mathbb{A}_\mathbf{v}$.[4] This fact is the basis for the algorithms used for formal explanation *enumeration* [12, 14].

2.4 Explaining Pushdown Automata decisions

This report describes ongoing work to explain the behaviour of a Pushdown Automaton (PDA) $\mathcal{A}$ on an input $w \in \Sigma^*$. Similar to explaining Finite Automata decisions [15], this problem can be viewed as a classifier mapping input w to a class in $\mathcal{K} = \{\text{accept}, \text{reject}\}$. We propose two types of explanations: **Abductive Explanations** (AXps), which answer "Why does $\mathcal{A}$ accept/reject w?"; and **Contrastive Explanations** (CXps), which answer "How can w be modified to alter the response of $\mathcal{A}$?".

The class of explanations that are considered in this work are defined by regular expressions [5] formed by replacing some characters in w by Σ

Typically, accepted words in Context-Free Languages are easy to modify to flip the prediction to **rejected**. Conversely, rejected words are often robustly wrong, requiring multiple coordinated changes (or single specific one) to repair the word.

The following example illustrates this behaviour.

[4]Given a collection of sets $\mathbb{S}$, a *hitting set* of $\mathbb{S}$ is a set H such that for each $S \in \mathbb{S}$, $H \cap S \neq \emptyset$. A hitting set is *minimal* if no proper subset of it is a hitting set.

[5]Standard regular expression notation is used. $\emptyset$ denote the empty language, $c \in \Sigma$ denotes the language $\{c\}$, and Σ the language $\{\Sigma\}$, the set of all strings of length 1. In general, given a regular expression R, the language it defines is denoted by $L(R)$. The concatenation of two regular expressions R_1R_2 denotes the regular language $\{r_1r_2 \mid r_1 \in L(R_1), r_2 \in L(R_2)\}$.

Example 2.5 (Fragility of Acceptance)**.** *Consider the accepted word $w =$ `()()`. The validity relies on every token. Modifying any single index is sufficient to flip the prediction to **rejected**.*

Every single index constitutes a CXP:

- ***Index 1:*** *Changing '`(`' $\to$ '`)`' makes `))()` (Rejected: starts with closing).*
- ***Index 2:*** *Changing '`)`' $\to$ '`(`' makes `(((` (Rejected: unmatched open).*
- ***Index 3:*** *Changing '`(`' $\to$ '`)`' makes `()))` (Rejected: unmatched close).*
- ***Index 4:*** *Changing '`)`' $\to$ '`(`' makes `()((` (Rejected: unmatched open).*

Since every feature is critical, the AXP (the subset of features required to guarantee acceptance) is the entire word.

The problem becomes significantly more interesting for rejected words.

Example 2.6 (Robustness of Rejection)**.** *Consider the rejected word $w =$ `))))`. Changing a single index is insufficient to flip the prediction to **accepted**. There are only two Minimal CXPs for this word:*

- *$\mathcal{CXP}_1 = \{1, 2\}$: Generates $\underline{\texttt{((}}\texttt{))}$.*
- *$\mathcal{CXP}_2 = \{1, 3\}$: Generates $\underline{\texttt{(}}\texttt{)}\underline{\texttt{(}}\texttt{)}$.*

From this, we can extract meaningful AXPs that are sufficient to guarantee rejection:

- *$\mathcal{AXP}_1 = \{1\}$: The first symbol '`)`' guarantees rejection regardless of the remaining suffix.*
- *$\mathcal{AXP}_2 = \{2, 3\}$: The substring '`))`' at indices 2 and 3 guarantees rejection for any word of length 4.*

2.4.1 Extracting CXps

Definition 2.5 (CYK-based Verification of CXp)**.** For an input string $w = w_1 w_2 \dots w_n$ and a grammar in Chomsky Normal Form, the CYK algorithm is modified to verify if an index set $S \subseteq \{1, \dots, n\}$ is a CXp for a rejected word as follows:

1. **Base Case:** For each $i \in \{1, \dots, n\}$:
   - If $i \in S$: $T[i, i] = \{A \in V \mid \exists \alpha \in \Sigma, A \to \alpha\}$ (all non-terminal symbols with at least one unary rule).
   - If $i \notin S$ (Fixed): $T[i, i] = \{A \in V \mid A \to w_i\}$ (as Standard CYK).
2. **Recursive Step ($j > i$):** Standard CYK update

The set S is a valid CXp if the start symbol belongs to $T[1, n]$.

$w_1 = \Sigma$ {L,R}	{B,P}	{L,U}	**{B}**
	$w_2 = \Sigma$ {L,R}	{B,P}	$\emptyset$
		$w_3 =$) {R}	$\emptyset$
			$w_4 =$ ')' {R}

Table 2.2: Modified CYK Table verifying CXp($\{1, 2\}$, `))))`). Indices 1 and 2 are treated as wildcards (Σ). **Abbreviations:** B = Balanced, P = Pair, U = Unclosed

Algorithm 1 EXTRACTCXP – a Single CXp Extraction

Input: Context-Free Grammar G, Candidate set $\mathcal{Y}$ (initially $\{1.\ldots.|w|\}$), word w
Output: Minimal CXp $\mathcal{Y}$

```
1: if not IsCXp($G, \mathcal{Y}, w$) then
2:     return $\perp$
3: for all $i \in \mathcal{Y}$ do
4:     if IsCXp($G, \mathcal{Y} \setminus \{i\}, w$) then
5:         $\mathcal{Y} \leftarrow \mathcal{Y} \setminus \{i\}$
6: return $\mathcal{Y}$
```

Algorithm 1 extracts a minimal CXp using a greedy deletion strategy. Initially, $\mathcal{Y}$ includes all indices (effectively treating the whole word as wildcards). The algorithm iterates through the candidates, attempting to "recover" the original token w_i at each position. If the function IsCXp (Definition 2.5) confirms that the word can still be corrected *without* changing index i, then i is redundant and removed from the explanation, otherwise, i is part of the explanation. The result is a minimal CXp: removing any index from $\mathcal{Y}$ would find a pattern containing no valid strings in $L(G)$.

Abductive Explanations (AXps) are extracted similarly. Since AXPs represent sufficient conditions for rejection, a set S is a valid AXP if treating indices in S as fixed (and all others as wildcards) results in the start symbol not appearing in $T[1, n]$.

2.4.2 Prioritizing Explanations

While the extraction of Minimal CXps provides a set of valid corrections, it often produces multiple candidates. For the rejected word $w =$ `))))`, we identified two minimal sets: $\mathcal{CXP}_1 = \{1, 2\}$ (suggesting `(())`) and $\mathcal{CXP}_2 = \{1, 3\}$ (suggesting `()()`).

A ranking approach is *Feature Attribution*, which counts the frequency of an index across all minimal explanations. In our case:

- **Index 1:** Frequency 1.0.
- **Index 2:** Frequency 0.5.
- **Indix 3:** Frequency 0.5.

However, symbolic frequency ignores the how likely is resulting correction. In domains like programming, certain structures (e.g., deeply nested brackets vs. sequential pairs) have different probabilities. To refine our suggestions, we use **Probabilistic Context-Free Grammars (PCFGs)**[16].

Example 2.7 (Rule Counting). *Consider a training dataset $D = \{$`()()()`, `(())()`, `()(())`$\}$. Figure 2.3 shows the derivation trees, and Table 2.3 the rule counting to estimate the probability of each production rule (B denotes the non-terminal Balanced).*

In this probabilistic model, the correction `()()` suggested by CXP_2 would likely receive a higher likelihood score than the nested structure suggested by CXP_1, allowing the explainer to present the most helpful minimal correction to the user.

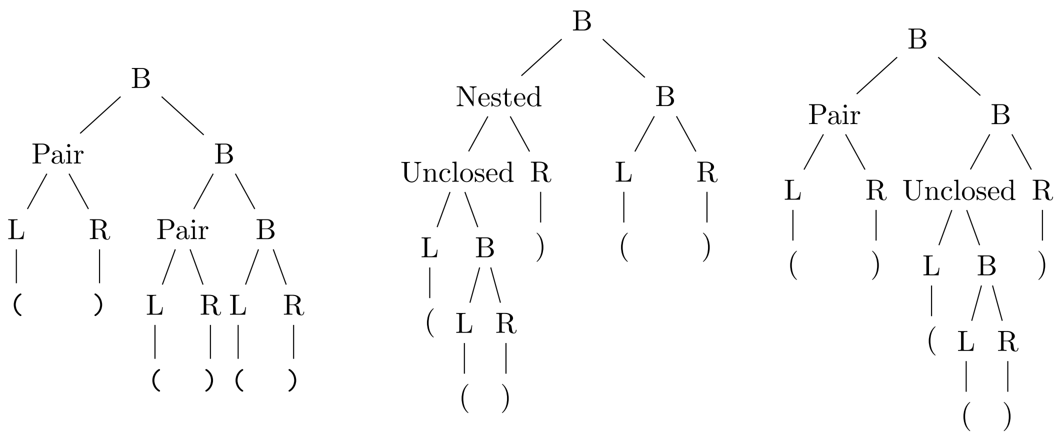


FIGURE 2.2: Parse trees for words `()()()`, `(())()`, `()(())`.

Rule	**Count**	**Probability (P)**
$B \to L\ R$	4	$4/9 = 44.\overline{4}$
$B \to$ Pair B	3	$3/9 = 33.\overline{3}$
$B \to$ Nested B	1	$1/9 = 11.\overline{1}$
$B \to$ Unclosed R	1	$1/9 = 11.\overline{1}$

TABLE 2.3: Probability distribution derived from D.

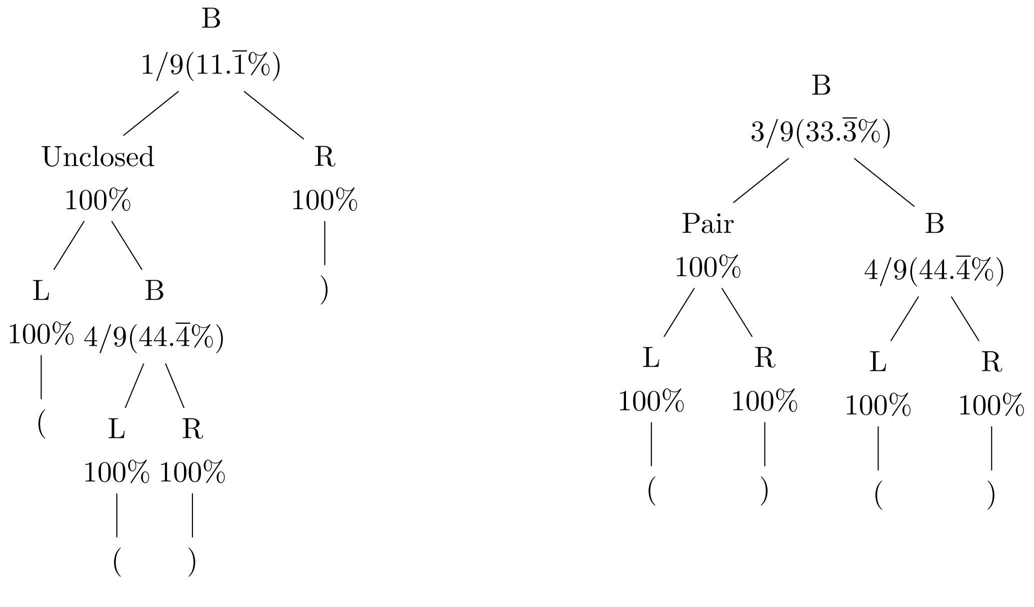


FIGURE 2.3: Parse trees for words `(())` and `()()`.

2.4.2.1 Ranking Explanations via PCFGs

To select the best correction, we calculate the probability of generating suggested correction given a fixed length constraint. For our running example to explain the rejected word $w =$ `))))` we consider the set of all valid words in the language of length 4 $L_4 = \{$`()()`, `(())`$\}$.

The total probability for valid words of length 4 is:

$$P(L_4) = P(\texttt{()()}) + P(\texttt{(())}) \tag{2.6}$$

Using the estimated rule probabilities, we compute the likelihood of the corrections suggested by our CXps (shown in Figure 2.3):

- $\mathcal{CXP}_1 \rightarrow$ `(())`: $P(\texttt{(())}) = \frac{1}{9} \times \frac{4}{9} \approx 0.049$
- $\mathcal{CXP}_2 \rightarrow$ `()()`: $P(\texttt{()()}) = \frac{3}{9} \times \frac{4}{9} \approx 0.148$

We define the *Relative Likelihood Score* for a CXp suggesting correction w' as:

$$Score(w') = P(w')/P(L_{|w|}) \tag{2.7}$$

$$Score(\mathcal{CXP}_1) = \frac{0.049}{0.148 + 0.049} \approx \mathbf{0.25} \quad \text{vs} \quad Score(\mathcal{CXP}_2) \approx \mathbf{0.75}$$

Under this PCFG, the correction `()()` is more likely than `(())`. This allows the explainer to prioritize the most statistically probable fix.

2.5 Conclusion

We present a polynomial-time algorithm for extracting Abductive and Contrastive Explanations relying on linear calls to a CYK based algorithm. Additionally, incorporating training data, this approach can prioritize the most relevant explanations. Future work will focus on benchmarks to evaluate the scalability

Chapter 3

Future Plan

In my third year, I plan to finalize the validation and theoretical details for Formal Explanations for Pushdown Automata. Then, I will address Research Problem 3, investigating formal explanations for models with sthocastic behaviour.

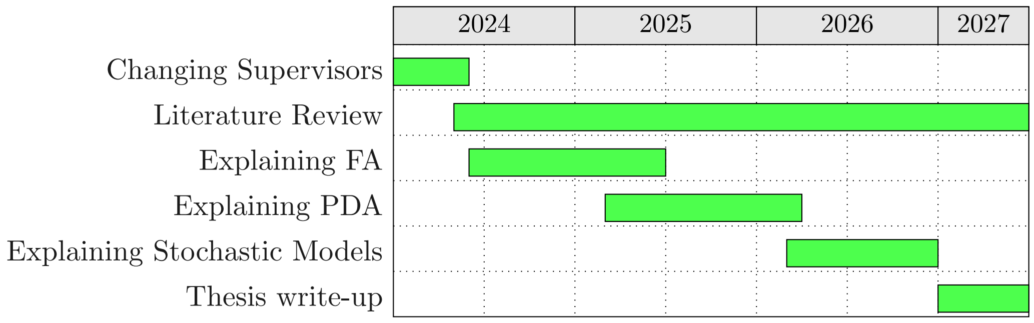


FIGURE 3.1: Timeline to completion of the PhD project

Bibliography

[1] João Marques-Silva. Logic-based explainability in machine learning. In Leopoldo E. Bertossi and Guohui Xiao, editors, *Reasoning Web. Causality, Explanations and Declarative Knowledge - 18th International Summer School 2022, Berlin, Germany, September 27-30, 2022, Tutorial Lectures*, volume 13759 of *Lecture Notes in Computer Science*, pages 24–104. Springer, 2022. doi: 10.1007/978-3-031-31414-8_2. URL https://doi.org/10.1007/978-3-031-31414-8_2.

[2] Adnan Darwiche. Logic for explainable AI. In *LICS*, pages 1–11. IEEE, 2023.

[3] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008. ISBN 978-0-262-02649-9.

[4] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23 (5):279–295, 1997. doi: 10.1109/32.588521. URL https://doi.org/10.1109/32.588521.

[5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986. ISBN 0-201-10088-6. URL https://www.worldcat.org/oclc/12285707.

[6] Max Brunsfeld. Tree-sitter: A new parsing system for programming tools. Presentation at Strange Loop Conference, 2018. Available at https://tree-sitter.github.io/.

[7] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive ll(*) parsing: the power of dynamic analysis. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 579–598. ACM, 2014. doi: 10.1145/2660193.2660202. URL https://doi.org/10.1145/2660193.2660202.

[8] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley, 2007. ISBN 978-0-321-47617-3.

[9] Jean Berstel and Luc Boasson. Context-free languages. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 59–102. Elsevier and MIT Press, 1990. doi: 10.1016/B978-0-444-88074-1.50007-X. URL https://doi.org/10.1016/b978-0-444-88074-1.50007-x.

[10] Daniel H. Younger. Recognition and parsing of context-free languages in time nˆ3. *Inf. Control.*, 10(2):189–208, 1967. doi: 10.1016/S0019-9958(67)80007-X. URL https://doi.org/10.1016/S0019-9958(67)80007-X.

[11] Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques - A Practical Guide*. Monographs in Computer Science. Springer, 2008. ISBN 978-0-387-20248-8. doi: 10.1007/978-0-387-68954-8. URL https://doi.org/10.1007/978-0-387-68954-8.

[12] João Marques-Silva and Alexey Ignatiev. Delivering trustworthy AI through formal XAI. In *AAAI*, pages 12342–12350. AAAI Press, 2022.

[13] Alexey Ignatiev, Nina Narodytska, Nicholas Asher, and João Marques-Silva. From contrastive to abductive explanations and back again. In *AI*IA*, volume 12414 of *Lecture Notes in Computer Science*, pages 335–355. Springer, 2020.

[14] Jinqiang Yu, Alexey Ignatiev, and Peter J. Stuckey. On formal feature attribution and its approximation. *CoRR*, abs/2307.03380, 2023.

[15] Jaime Cuartas Granada, Alexey Ignatiev, and Peter J. Stuckey. A formal framework for the explanation of finite automata decisions, 2026. URL https://arxiv.org/abs/2602.13351.

[16] Taylor L. Booth and Richard A. Thompson. Applying probability measures to abstract languages. *IEEE Trans. Computers*, 22(5):442–450, 1973. doi: 10.1109/T-C.1973.223746. URL https://doi.org/10.1109/T-C.1973.223746.