Bachelor's Thesis Computing Science

Active Learning of Incomplete Mealy Machines

Wouter Joosen
s1122537

June 25, 2026

First supervisor/assessor:
Prof. F.W. Vaandrager

Second assessor:
Dr J.C. Rot

Radboud University

Abstract

Active automata learning is a technique used to infer state models of black-box systems through queries and counterexamples. However, most active learning algorithms, including $L^{\#}$, assume a complete teacher, making them inapplicable to systems where behaviour is incomplete or partially hidden. While $L^{\#}_{\square}$, an extension of $L^{\#}$, addresses this for DFAs by using an SMT solver to find minimal separating automata, extending it to Mealy machines remains an open challenge. This thesis addresses this limitation by adapting the $L^{\#}_{\square}$ algorithm to support the active learning of incomplete Mealy machines. As a baseline, empirical evaluation on complete teachers shows that the algorithm produces correct and minimal hypotheses with predictable overhead over standard $L^{\#}$. More importantly, on incomplete teachers the algorithm reliably learns minimal machines, though efficiency depends heavily on the structure and complexity of the target machine.

Contents

Chapter 1

Introduction

The field of active automata learning was revolutionised in 1987, when Dana Angluin published her paper on using queries and counterexamples to learn regular languages [1]. Her paper first introduced the concept of the *learning game* used to learn an unknown target language L by asking questions to a teacher, an oracle with complete knowledge of the target language. This is called the *Minimally Adequate Teacher (MAT)* framework. In this framework, the learner can pose two types of queries: *membership queries* and *equivalence queries*. With a membership query, the learner asks the teacher if a specific word w is part of the target language L. With an equivalence query, the learner asks the teacher if the language recognised by the hypothesis DFA is equal to L. If the teacher responds with *"No"*, then it returns a counterexample. Otherwise, both languages are equivalent and the learning game is complete. Alongside the learning game, Angluin's paper proposed L^*: the first algorithm capable of playing this learning game.

In the decades after Angluin's paper, extensive research focused on improving the L^* algorithm. Many of these works aimed to optimise the algorithm, for instance [19, 12, 8, 13]. Over time, researchers also adapted the core ideas of L^* to learn other types of systems. For instance, variations were proposed to learn non-deterministic automata [3], weighted automata [2, 21], and Mealy machines [17]. A different approach to active automata learning was introduced with the $L^\#$ framework [20]. While L^*-based algorithms rely on proving equivalence of observations in an observation table or discrimination tree, $L^\#$ uses an *observation tree* and is based on *apartness*, which it uses to determine when two nodes in the tree represent distinct states in the target automaton.

However, both L^* variants and $L^\#$ typically assume a complete teacher, meaning it accurately answers any query without gaps in its knowledge. The concept of active learning from 'incomplete' or 'inexperienced' teachers, those that may return "don't know" ($\square$) responses, has been explored in the context of DFA learning by Leucker & Neider [11] and Moeller et al. [14]. This approach is essential in scenarios such as separating DFAs, where the teacher returns $\square$ for words falling outside disjoint languages L^+ and L^-, or when using 3DFAs (as introduced by Chen et al. [5]), where the acceptance function is partial. In these DFA settings, 3DFAs can be referred to as 'incomplete DFAs' due to their partial nature, which then establishes a direct correspondence to 'incomplete teachers'. More recent research introduced $L^\#_\square$, an $L^\#$ based algorithm for finding separating automata for DFAs using a Satisfiability Modulo Theories (SMT) solver. While this algorithm successfully handles incomplete behaviour in DFAs, many systems produce outputs other than binary acceptance, which are more naturally represented as Mealy machines, where

an incomplete teacher would then return $\square$ for an unspecified output. Extending this algorithm to Mealy machines remains an open challenge highlighted in the original $L^{\#}_{\square}$ paper [10]. This thesis aims to address this limitation by adapting the $L^{\#}_{\square}$ algorithm to support the learning of incomplete Mealy machines, where some outputs may be undefined ($\square$). The approach primarily relies on modifying the observation tree to store output sequences and reformulating the underlying SMT constraints to handle unknown output symbols.

This research is driven by the following three key questions:

RQ1. How can the $L^{\#}_{\square}$ algorithm, originally implemented for DFAs, be adapted to learn incomplete Mealy machines?

RQ2. How does the performance of $L^{\#}_{\square}$ compare to standard $L^{\#}$ when learning complete Mealy machines?

RQ3. How does $L^{\#}_{\square}$ perform and behave when learning incomplete Mealy machines?

The remainder of this thesis is structured as follows: Chapter 2 introduces the preliminary definitions and notation necessary for this work. Chapter 3 provides the details of the $L^{\#}_{\square}$ Mealy algorithm together with an illustrative example. Chapter 4 outlines the algorithm implementation and evaluates the performance of $L^{\#}_{\square}$ against standard $L^{\#}$ on complete teachers. Chapter 5 extends this evaluation to incomplete teachers, using case studies and random benchmarks to analyse $L^{\#}_{\square}$'s behaviour. Finally, Chapter 6 concludes the thesis and discusses potential limitations, as well as directions for future research.

Chapter 2

Preliminaries

This section introduces all definitions and notations necessary for understanding this thesis. The notation and definitions in this section follow the approach of [10].

We write $f : X \rightharpoonup Y$ to denote that f is a partial function from X to Y and we write $f(x) \downarrow$ if f is defined on x, that is, $\exists y \in Y : f(x) = y$. Conversely, we write $f(x) \uparrow$ if f is undefined for x. Function $f : X \rightharpoonup Y$ is a total function if $f(x) \downarrow$ for all $x \in X$. We then write $f : X \to Y$.

For this thesis, we fix a finite input alphabet Σ and a finite output alphabet Γ. We use standard notations for sequences. If X is a set, then X^* denotes the set of finite *sequences* (or *words*) over X. We write ϵ to denote the empty sequence, x to denote the sequence consisting of a single element $x \in X$, and σp, or $\sigma \cdot p$, to denote the concatenation of sequences $\sigma, p \in X^*$.

Below, we introduce the definition of a partial Mealy machine.

Definition 2.1 (Partial Mealy Machine). A partial Mealy machine is a tuple $\mathcal{M} = (Q, q^0, \delta, \lambda)$, where

- Q is a finite set of *states*,
- $q^0 \in Q$ is the *initial state*,
- $\delta : Q \times \Sigma \rightharpoonup Q$ is the *transition function*, and
- $\lambda : Q \times \Sigma \rightharpoonup \Gamma$ is the *output function*.

We distinguish three important cases:

- A **partial Mealy machine** has both the transition function and the output function as partial (Definition 2.1).
- A **transition complete Mealy machine** is a partial Mealy machine where the transition function is total, but the output function may be partial. This is the natural analogue of 3DFAs in the DFA setting [5], where the acceptance function is partial.
- A **complete Mealy machine** is a partial Mealy machine where both the transition function and the output function are total.

We require that if the output is defined for a state q and input a, then the transition for state q and input a must also be defined, i.e., $\lambda(q, a) \downarrow \Rightarrow \delta(q, a) \downarrow$.

We denote the absence of a concrete output for a transition from state q with input a as $\lambda(q,a) = \square$ (or $\lambda(q,a)\uparrow$), representing an *unspecified* or *unknown* output.

We extend the output function λ to sequences. We define $\lambda^* : Q \times \Sigma^* \to \Gamma \cup \{\square, \epsilon\}$ as follows. For $a \in \Sigma$ and $w \in \Sigma^+$:

- $\lambda^*(q,\epsilon) = \epsilon$
- $\lambda^*(q,a) = \lambda(q,a)$
- $\lambda^*(q,aw) = \lambda^*(\delta(q,a),w)$

Let $u, v \in \Gamma \cup \{\square, \epsilon\}$ be output symbols. We say that v is *consistent* with u, if either $u = \square$, $v = \square$, or $u = v$.

We write $\delta(q,a) = r$ if input a on state q leads to state r, and $\lambda(q,a) = b$ if input a on state q leads to output b. We write $q \xrightarrow{a} q'$ if $\delta(q,a) = q'$, and $q \xrightarrow{a/b} q'$ if $\delta(q,a) = q' \wedge \ \lambda(q,a) = b$. Additionally, we write $q \overset{w}{\Longrightarrow} q'$ if there is an input sequence w of arbitrary length that labels a path from state q to state q'. We use subscript $\mathcal{M}$ to disambiguate to which Mealy machine we refer, e.g. $Q_\mathcal{M}$ and $\delta_\mathcal{M}$. We call states p and q *conflicting* if $\lambda(p,a)\downarrow \wedge\, \lambda(q,a)\downarrow \wedge\, \lambda(p,a) \neq \lambda(q,a)$ for some input a.

We consider maps between Mealy machines that preserve initial states, the transition function, and the output function.

Definition 2.2 (Morphism). For Mealy machines $\mathcal{M}$ and $\mathcal{N}$, a morphism from $\mathcal{M}$ to $\mathcal{N}$ is a function $f : Q_\mathcal{M} \to Q_\mathcal{N}$ such that, for all $q, q' \in Q_\mathcal{M}$, and $a \in \Sigma$,

1. $f(q^0_\mathcal{M}) = q^0_\mathcal{N}$,
2. $\lambda_\mathcal{M}(q,a)\downarrow \implies \lambda_\mathcal{M}(q,a) = \lambda_\mathcal{N}(f(q),a)$, and
3. $q \xrightarrow{a}_\mathcal{M} q' \implies f(q) \xrightarrow{a}_\mathcal{N} f(q')$.

We write $\mathcal{M} \sqsubseteq_f \mathcal{N}$ if f is a morphism from $\mathcal{M}$ to $\mathcal{N}$, and $\mathcal{M} \sqsubseteq \mathcal{N}$ if there exists a morphism from $\mathcal{M}$ to $\mathcal{N}$.

We use an *observation tree* to represent a set of known transitions and their corresponding outputs.

Definition 2.3 (Observation Tree). A partial Mealy machine $\mathcal{T}$ is an *observation tree* iff for each $q \in Q_\mathcal{T}$ there is a unique sequence of inputs $w \in \Sigma^*$ with $q^0_\mathcal{T} \overset{w}{\Longrightarrow} q$. We write `access(q)` to denote this unique sequence. We say that $\mathcal{T}$ is an *observation tree* for a Mealy machine $\mathcal{M}$ if $\mathcal{T}$ is an observation tree and there exists a morphism $f : \mathcal{T} \to \mathcal{M}$.

Example 2.4. Figure 2.1 (bottom) shows a complete Mealy machine $\mathcal{M}$ with 4 states over the alphabet $\Sigma = \{a, b\}$. The machine outputs `1` if the input sequence processed so far contains both an even number of a's and an even number of b's, and outputs `0` otherwise. Figure 2.1 (top) shows an observation tree $\mathcal{T}$ for $\mathcal{M}$. In the observation tree $\mathcal{T}$, transitions for which the output is unspecified are denoted by the symbol $\square$, representing a "don't know" value. Morphism f is indicated via colouring of the states.

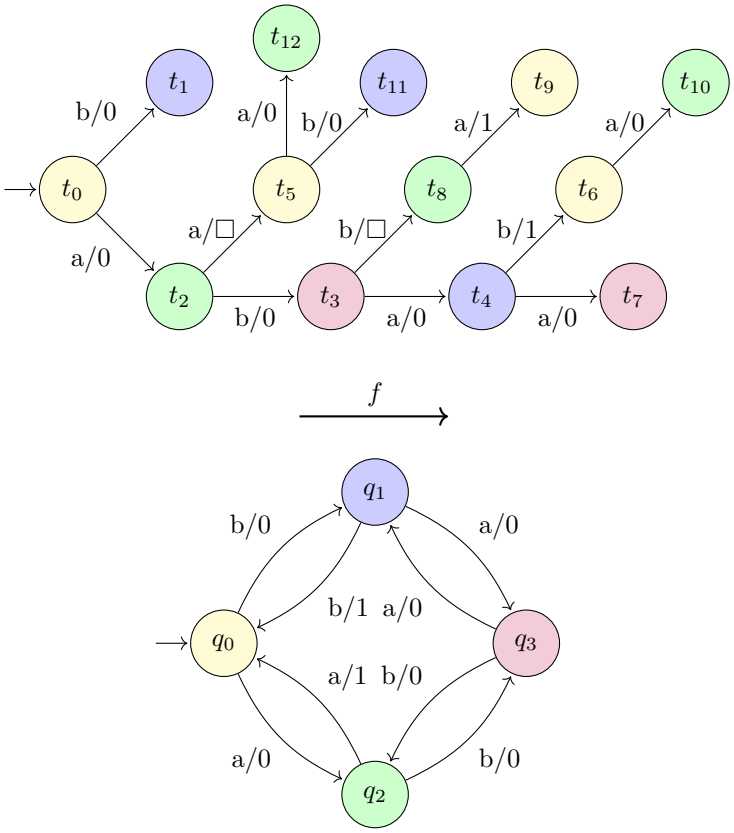


Figure 2.1: An observation tree $\mathcal{T}$ (top) for a Mealy machine $\mathcal{M}$ (bottom).

Two states of a Mealy machine are *compatible* if there exists a morphism that maps both of them to the same state.

Definition 2.5 (Compatibility). Let $\mathcal{M}$ be a Mealy machine $\mathcal{M}$. We call states p and q of $\mathcal{M}$ *compatible* if $\mathcal{M} \sqsubseteq_f \mathcal{N}$, for some Mealy machine $\mathcal{N}$ and morphism f with $f(p) = f(q)$. If p and q are not compatible then we say they are *incompatible*.

Apartness is a constructive form of incompatibility where we have a *witness* that proves the incompatibility.

Definition 2.6 (Apartness). For a Mealy machine $\mathcal{M}$, we say that states $p, q \in Q_{\mathcal{M}}$ are apart (written $p \# q$) if there is some sequence $u \in \Sigma^*$, an input $a \in \Sigma$, and $p', q' \in Q_{\mathcal{M}}$ such that:

1. $p \stackrel{u}{\Rightarrow} p'$, $q \stackrel{u}{\Rightarrow} q'$, and
2. $\lambda(p', a)\downarrow \wedge \lambda(q', a)\downarrow \wedge \lambda(p', a) \neq \lambda(q', a)$ (i.e., p' and q' are conflicting on input a).

We call $w = ua$ a *witness* for $p \# q$ and write $w \vdash p \# q$.

Back propagation allows us to identify two states that have a transition into two apart states as apart themselves.

Lemma 2.7 (Back Propagation). Let $\mathcal{M}$ be a Mealy machine with states p, p', q, and q'. Let $a \in \Sigma$. Then, $p \xrightarrow{a} p'$ and $q \xrightarrow{a} q'$ and $w \vdash p' \# q'$ implies $aw \vdash p \# q$.

Example 2.8. For the observation tree of Figure 2.1, we may derive a number of apartness pairs and corresponding witnesses:

$$a \vdash t_5 \# t_8 \qquad a \vdash t_8 \# t_6 \qquad b \vdash t_5 \# t_4 \qquad b \vdash t_2 \# t_4$$

Thus, by back propagation we have:

$$ba \vdash t_3 \# t_4$$

Two states are only considered apart if they produce distinct, concrete outputs for the same input sequence. Consequently, if state p yields the unspecified output $\square$ for an input a while state q yields a concrete output 0 for the same input a, they are not yet known to be apart. In Figure 2.1, this is why t_2 and t_3 cannot be declared apart despite their differing symbols.

Definition 2.9 (Consistent Bisimulation). Let $\mathcal{M} = (Q_\mathcal{M}, q^0_\mathcal{M}, \delta_\mathcal{M}, \lambda_\mathcal{M})$ be a Mealy machine and $\mathcal{N} = (Q_\mathcal{N}, q^0_\mathcal{N}, \delta_\mathcal{N}, \lambda_\mathcal{N})$ be another Mealy machine sharing a common input alphabet Σ and output alphabet Γ. A binary relation $R \subseteq Q_\mathcal{M} \times Q_\mathcal{N}$ is a bisimulation between $\mathcal{M}$ and $\mathcal{N}$ if, for all pairs of states $(p, q) \in R$ and for every input symbol $a \in \Sigma$, the following two conditions are satisfied:

1. $\lambda_\mathcal{M}(p, a) \downarrow \wedge\ \lambda_\mathcal{N}(q, a) \downarrow \implies \lambda_\mathcal{M}(p, a) = \lambda_\mathcal{N}(q, a)$
2. $(\delta_\mathcal{M}(p, a), \delta_\mathcal{N}(q, a)) \in R$

Two states $p \in \mathcal{M}$ and $q \in \mathcal{N}$ are consistently bisimilar, denoted by $p \sim q$, if there exists a bisimulation relation R such that $(p, q) \in R$.

Two Mealy machines $\mathcal{M}$ and $\mathcal{N}$ are consistently bisimilar, denoted by $\mathcal{M} \sim \mathcal{N}$, if and only if their initial states are consistently bisimilar: $(q^0_\mathcal{M}, q^0_\mathcal{N}) \in R$

To algorithmically verify whether two Mealy machines are consistently bisimilar according to Definition 2.9, we use a Breadth-First-Search (BFS) over all pairs of states from both machines. Algorithm 1 outlines this strategy. This check is used to establish equivalence, while at the same time constructing a counterexample to return when an output inconsistency is found.

Algorithm 1 Consistent Bisimulation Check

Require: Mealy Machines $\mathcal{M}$, $\mathcal{N}$ over input alphabet Σ

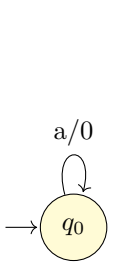
```
queue ← queue containing pair (q^0_M, q^0_N)
witness(q^0_M, q^0_N) ← ϵ
while queue is not empty do
    (p, q) ← queue.DEQUEUE()
    for a ∈ Σ do
        if λ_M(p, a)↓ ∧ λ_N(q, a)↓ ∧ λ_M(p, a) ≠ λ_N(q, a) then
            return witness(p, q) · a                          ▷ Counterexample found
        end if
        (p′, q′) ← (δ_M(p, a), δ_N(q, a))
        if (p′, q′) not yet visited then
            witness(p′, q′) ← witness(p, q) · a      ▷ Track path for counterexample generation
            queue.ENQUEUE((p′, q′))
        end if
    end for
end while
return "Yes"                                          ▷ M ∼ N: no inconsistency found
```

Complexity Analysis. In the worst-case scenario, the algorithm explores all reachable state pairs between the two machines. Let $|Q_{\mathcal{M}}|$ and $|Q_{\mathcal{N}}|$ denote the number of states in $\mathcal{M}$ and $\mathcal{N}$, respectively. Then, the maximum number of states visited is $|Q_{\mathcal{M}}| \times |Q_{\mathcal{N}}|$. For each unique pair in the queue, the algorithm iterates over all possible input symbols $a \in \Sigma$. Therefore, the total worst-case time complexity is $\mathcal{O}(|Q_{\mathcal{M}}| \times |Q_{\mathcal{N}}| \times |\Sigma|)$.

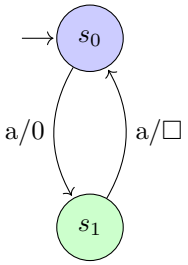
Example 2.10. We illustrate the execution of Algorithm 1 using the two Mealy machines depicted in Figure 2.2, which presents a hypothesis machine $\mathcal{H}$ and a target machine $\mathcal{M}$ over the alphabet $\Sigma = \{a\}$. We can verify whether the two machines are consistently bisimilar by tracing how the Breadth-First Search (BFS) constructs the relation R:

The search starts at the initial state pair, adding (q_0, s_0) to R. Evaluating the input symbol a results in $\lambda(q_0, a) = 0$ and $\lambda(s_0, a) = 0$. Since these outputs match, no inconsistency is detected. The transition functions lead to the next state pair since $\delta(q_0, a) = q_0$ and $\delta(s_0, a) = s_1$, so the pair (q_0, s_1) is added to R as an unexplored state pair.

Next, the pair (q_0, s_1) is evaluated for the input a. Here, $\lambda(q_0, a) = 0$ while $\lambda(s_1, a) = \square$. Because the unknown symbol $\square$ is consistent with any output symbol, this does not result in an inconsistency. The corresponding transitions lead to $\delta(q_0, a) = q_0$ and $\delta(s_1, a) = s_0$. The resulting state pair (q_0, s_0) has already been explored. Since all reachable state pairs have been checked without encountering an inconsistency, the construction of the relation $R = \{(q_0, s_0), (q_0, s_1)\}$ is complete. Thus, we can conclude that the hypothesis $\mathcal{H}$ is consistently bisimilar to the target machine $\mathcal{M}$ ($\mathcal{H} \sim \mathcal{M}$).



(a) Hypothesis $\mathcal{H}$.



(b) Target Mealy machine $\mathcal{M}$.

Figure 2.2: Example comparing a complete 1-state hypothesis with a transition-complete 2-state target machine.

Definition 2.11 (Equivalence). Let $\mathcal{M}$ and $\mathcal{N}$ be Mealy machines. Then, $\mathcal{M}$ is equivalent to $\mathcal{N}$, denoted by $\mathcal{M} \approx \mathcal{N}$, if and only if for all $\sigma \in \Sigma^*$, $\lambda^*_{\mathcal{M}}(q^0_{\mathcal{M}}, \sigma)$ is consistent with $\lambda^*_{\mathcal{N}}(q^0_{\mathcal{N}}, \sigma)$.

Lemma 2.12 (Characterisation of Equivalence). Let $\mathcal{M}$ and $\mathcal{N}$ be two Mealy machines sharing a common input alphabet Σ and output alphabet Γ. Then, $\mathcal{M}$ and $\mathcal{N}$ are consistently bisimilar if and only if they are equivalent:

$$\mathcal{M} \sim \mathcal{N} \iff \mathcal{M} \approx \mathcal{N}$$

This characterisation is essential for our bisimulation-based equivalence check (see Algorithm 1), as it allows us to verify equivalence by checking for a consistent bisimulation. (*The full proof is provided in Appendix A.*)

Remark 2.13 (Computational Complexity of Minimisation). Lemma 2.12 establishes how to characterise the equivalence between partial and complete Mealy machines. From an algorithmic standpoint this raises an important question: given a partial Mealy machine $\mathcal{M}$ and a bound k, how difficult is it to find an equivalent complete Mealy machine with fewer than k states?

While complete Mealy machines can be minimised quickly in polynomial time [6], adding incompleteness makes the problem much harder. In fact, the problem of minimising incompletely-specified finite-state machines has been proven to be NP-complete [18]. This hardness provides a strong motivation for using an SMT solver in our approach.

Chapter 3

The Algorithm

Let us assume that there exists a transition complete Mealy machine $\mathcal{M}$ and an incomplete teacher that represents machine $\mathcal{M}$. Assume that the $L^{\#}_{\square}$ algorithm is the learner. The goal of the $L^{\#}_{\square}$ algorithm is to learn $\mathcal{M}$ by posing two different types of queries to the teacher: an output query, and an equivalence query.

In an **output query**, the learner asks the teacher: *What is the last output symbol produced for this input sequence?* Unlike the standard $L^{\#}$ setting, in $L^{\#}_{\square}$ we consider the teacher to be incomplete, meaning that for some inputs it outputs $\square$ ("don't know"). Consequently, for an output query on input sequence $w \in \Sigma^*$, the teacher responds with $\lambda^*(q^0, w)$ (the last symbol of the output sequence corresponding to input sequence w), where q^0 is the initial state.

After posing multiple of these queries, the learner builds a complete hypothesis Mealy machine $\mathcal{H}$ using the observed information. Eventually, it poses an equivalence query.

In an **equivalence query**, the learner asks the teacher: *Is the hypothesis Mealy machine $\mathcal{H}$ equivalent to your target machine $\mathcal{M}$?* ($\mathcal{H} \approx \mathcal{M}$). By our Characterisation of Equivalence (Lemma 2.12), we know that checking this equivalence reduces to checking if the two machines are consistently bisimilar ($\mathcal{H} \sim \mathcal{M}$). To answer this equivalence query, the teacher attempts to verify consistent bisimilarity by performing a Breadth-First Search (BFS) to incrementally construct a valid bisimulation relation $R \subseteq Q_{\mathcal{H}} \times Q_{\mathcal{M}}$. The teacher can respond with one of two answers: if the BFS successfully explores the states without finding a consistency violation, the machines are equivalent, and it simply responds with *"Yes"*. Otherwise, the BFS will encounter a state pair where the bisimulation conditions fail. While our BFS implementation naturally returns the shortest path to this inconsistency as a minimal counterexample, in general the teacher may return *any* valid counterexample (not necessarily a minimal one). $L^{\#}_{\square}$ will still successfully process it and guarantee termination.

3.1 Observation Tree

In order to keep track of all query results, $L^{\#}_{\square}$ makes use of an observation tree $\mathcal{T}$. Every run of $L^{\#}_{\square}$ starts with an observation tree initialised with the initial state q^0. In order to efficiently construct a hypothesis, the algorithm partitions the nodes of the observation tree into three sets: the basis, the frontier, and the candidate set.

Definition 3.1 (Basis). The basis is a set of states $S \subseteq Q_{\mathcal{T}}$ such that S is a collection of states of observation tree $\mathcal{T}$ that are pairwise apart. Initially, $S = \{q^0\}$.

Definition 3.2 (Frontier). The frontier is a set of states F such that F consists of all non-basis states $q \in Q_{\mathcal{T}} \setminus S$ such that q is an immediate successor of some $r \in S$.

Definition 3.3 (Candidate Set). For each non-basis state $q \in Q_{\mathcal{T}} \setminus S$, we maintain its candidate set, that is, the set of basis states from which it is not apart:

$$C(q) = \{p \in S \mid \neg(q \# p)\}$$

Moreover, we keep track of an integer n, which defines the minimal number of states that the next hypothesis must have. Initially, n is set to 1 and $|S| \leq n$ must always hold.

3.2 Algorithm Rules

$L^{\#}_{\square}$ makes use of the following four rules that it applies non-deterministically, until it finds an equivalent complete Mealy machine:

Promotion Rule: Whenever we find a state q that is apart from all basis states, i.e. has an empty candidate set, we may add q to the basis S. Then, we set $n = \max(n, |S|)$.

Extension Rule: For any state $q \in S$, with access sequence $w = \texttt{access(q)}$ and a word $v \in \Sigma^*$ with length at most $n - |S| + 1$, we may pose an output query wv, if we have not done so before.

Identification Rule: For any state q with access sequence $w = \texttt{access(q)}$ that is a successor of a basis state (= frontier state), states $p, r \in C(q)$ and witness v such that $v \vdash p \# r$, we may pose an output query wv, if we have not done so before. Then, if the output for v differs from the output produced by basis state p, then v serves as a witness for $q \# p$. Conversely, if the output for v differs from the output produced by basis state r, then v serves as a witness for $q \# r$.

Validity Rule: We may ask the SMT solver if there exists a complete Mealy machine $\mathcal{H}$ with at most n states, with a morphism $f : \mathcal{T} \to \mathcal{H}$. If the SMT solver responds with "No", then we increment n. Otherwise, the SMT solver provides a Mealy machine $\mathcal{H}$. We use $\mathcal{H}$ as our hypothesis and pose an equivalence query to the teacher for $\mathcal{H}$. If the teacher answers "Yes", then $L^{\#}_{\square}$ is finished and returns the Mealy machine $\mathcal{H}$. Otherwise, the teacher provides a counterexample w, and we pose an output query for w and all its prefixes. This ensures that $L^{\#}_{\square}$ learned from the counterexample and that $\mathcal{H}$ will not be proposed again by the SMT solver.

3.3 SMT Solver

In the validity rule, $L^{\#}_{\square}$ makes use of an SMT solver to decide whether there exists a complete Mealy machine $\mathcal{H}$ with at most n states with a morphism $f : \mathcal{T} \to \mathcal{H}$. For this, we use the SMT solver Z3. We introduce the following encodings:

- The states of $\mathcal{H}$ are numbered from 1 to $n = |Q_{\mathcal{H}}|$.
- The states of $\mathcal{T}$ are numbered from 1 to $m = |Q_{\mathcal{T}}|$. We say that I_q is the number associated with state $q \in Q_{\mathcal{T}}$.
- We ensure that $I_q \leq |S|$ $(\leq n)$ for each $q \in S$.

– The transition function is encoded as the function $\delta_{\mathcal{H}} : [1..n] \times \Sigma \to [1..n]$.

– The output function is encoded as the function $\lambda_{\mathcal{H}} : [1..n] \times \Sigma \to \Gamma$.

– The morphism is encoded by a function $f : [1..m] \to [1..n]$.

Additionally, we introduce the following constraints for the SMT solver:

$$\bigwedge_{q \in Q_{\mathcal{T}}} \bigwedge_{\sigma \in \Sigma} \delta_{\mathcal{T}}(q, \sigma) \downarrow \implies f(I_{\delta_{\mathcal{T}}(q,\sigma)}) = \delta_{\mathcal{H}}(f(I_q), \sigma) \tag{1}$$

$$\bigwedge_{q \in Q_{\mathcal{T}}} \bigwedge_{\sigma \in \Sigma} \lambda_{\mathcal{T}}(q, \sigma) \downarrow \implies \lambda_{\mathcal{T}}(q, \sigma) = \lambda_{\mathcal{H}}(f(I_q), \sigma) \tag{2}$$

$$\bigwedge_{q \in S} f(I_q) = I_q \tag{3}$$

$$\bigwedge_{q \in Q_{\mathcal{T}} \setminus S} \left(\left(\bigvee_{|S| < k \leq n} f(I_q) = k \right) \vee \left(\bigvee_{p \in C(q)} f(I_q) = I_p \right) \right) \tag{4}$$

In other words:

– Constraint (1) states that if the transition for a state q on input σ exists, then that same transition must exist in the hypothesis.

– Constraint (2) states that if an output for a state q on input σ exists, then that same output must be produced by the morphism for state q on input σ.

– Constraint (3) states that each basis state must be mapped to a unique state in the hypothesis.

– Constraint (4) states that each state q not in the basis must be mapped to either a new state or a compatible basis state (in the candidate set for state q).

3.4 Main Loop

While in theory all four rules can be applied non-deterministically, in practice it is more efficient to have a structured ordering. In our implementation, we apply the *Extension rule*, the *Promotion rule*, and the *Identification rule* repeatedly in that order until none of them can be applied anymore, before finally applying the *Validity rule*. This sequence is chosen because the validity rule is computationally the most expensive, and minimising its execution is necessary for the performance of $L^{\#}_{\square}$.

Algorithm 2 $L^{\#}_{\square}$ Pseudo-code

Require: Alphabet Σ, System Under Learning $\mathcal{S}$
$\mathcal{T} \leftarrow$ Initialise tree with root ϵ
$n \leftarrow 1$ ▷ Smallest possible number of states
loop
  while rules can be applied to $\mathcal{T}$ **do** ▷ Refining the tree
    Extension rule
    Promotion rule
    Identification rule
  end while
  $\mathcal{H} \leftarrow$ **Validity rule** $(n, \mathcal{T})$ ▷ Search for n-state complete Mealy machine
  if $\mathcal{H}$ is NULL **then**
    $n \leftarrow n + 1$ ▷ Increase search space for minimality
  else
    $ce \leftarrow$ EquivalenceQuery$(\mathcal{H})$ ▷ Ask teacher for hypothesis correctness
    if ce is NULL **then**
      return $\mathcal{H}$ ▷ Correct minimal complete Mealy machine found
    else
      AddObservation$(\mathcal{T}, ce)$ ▷ Refine tree with counterexample
    end if
  end if
end loop

As shown in Algorithm 2, the search for a hypothesis starts at $n = 1$ and only increments when the basis size exceeds the current value of n or when the SMT solver proves that no consistent model exists at the current size. This ensures that the first valid hypothesis accepted by the teacher is designed to be the minimal equivalent complete Mealy machine.

3.5 Illustrative Example

In this section, we will look at an example run of the $L^{\#}_{\square}$ algorithm.

Consider transition complete Mealy machine $\mathcal{M}$ shown in Figure 3.1.

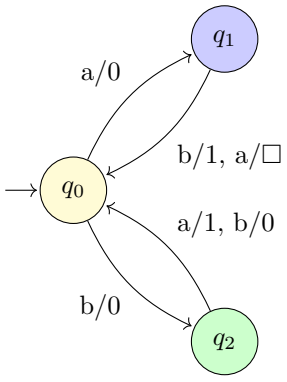


Figure 3.1: Transition complete Mealy machine $\mathcal{M}$.

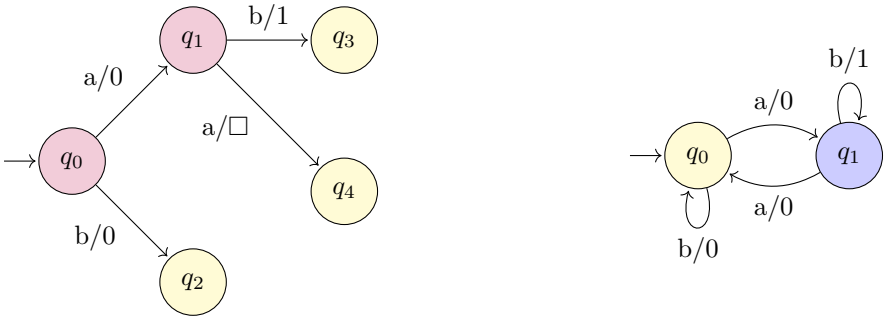
We illustrate the $L^{\#}_{\square}$ algorithm on learning this Mealy machine $\mathcal{M}$. $L^{\#}_{\square}$ initialises the observation tree with a single state q_0. Next, the extension rule is applied twice and new states q_1 and q_2 are added to the tree with access sequences a and b, respectively. Both transitions $q_0 \xrightarrow{a} q_1$ and $q_0 \xrightarrow{b} q_2$ output 0. We cannot apply any other rules, so we resort to the validity rule and we ask the SMT solver to generate a hypothesis. There is only one possible hypothesis that the SMT solver can return, which is $\mathcal{H}_1$ shown in Figure 3.2b.

(a) Observation Tree $\mathcal{T}$.

(b) Hypothesis $\mathcal{H}_1$.

Figure 3.2: Observation tree $\mathcal{T}$ and hypothesis $\mathcal{H}_1$ after first learning round.

We submit this hypothesis to the teacher, who will return a counterexample, say ab. Via output query ab, we find out that we have a new state q_3 with $q_1 \xrightarrow{b} q_3$ outputting 1. Witness b shows that states q_0 and q_1 are apart, and thus the promotion rule can be applied: state q_1 is added to the basis and n is incremented to 2. The extension rule is applied once to explore the new basis state q_1 with sequence a. An output query on this input reveals that we have a new state q_4 with $q_1 \xrightarrow{a} q_4$ returning $\square$ ("don't know"). We apply the validity rule. Suppose the SMT solver returns hypothesis $\mathcal{H}_2$ shown in Figure 3.3b (where it obtained output 0 for the $\square$). Note that when the observation tree contains a $\square$ output, the SMT solver is free to assign any concrete output value from the alphabet Γ, provided the resulting hypothesis remains consistent with all other known observations in the observation tree $\mathcal{T}$.

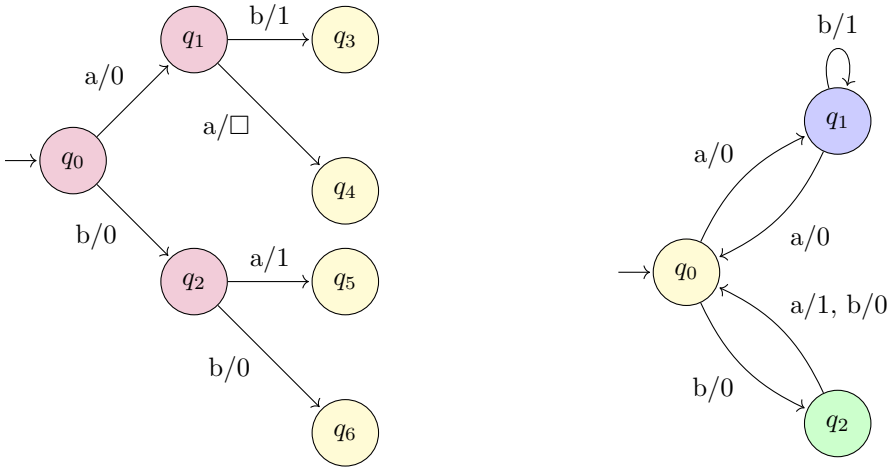


(a) Observation Tree $\mathcal{T}$.

(b) Hypothesis $\mathcal{H}_2$.

Figure 3.3: Observation tree $\mathcal{T}$ and hypothesis $\mathcal{H}_2$ after second learning round.

We submit this hypothesis to the teacher, who will return a counterexample, say ba. Via output query ba, we find out that we have a new state q_5 with $q_2 \xrightarrow{a} q_5$ outputting 1. Witness a shows that states q_0 and q_2 are apart, but we cannot yet prove that q_1 and q_2 are apart. Thus, we apply the identification rule. We pose an output query for bb and we find that we have a new state q_6 with $q_2 \xrightarrow{b} q_6$. Witness b shows that states q_1 and q_2 are apart, and thus the promotion rule can be applied: state q_2 is added to the basis and n is set to 3. We apply the validity rule. Suppose the SMT solver returns hypothesis $\mathcal{H}_3$ shown in Figure 3.4b.

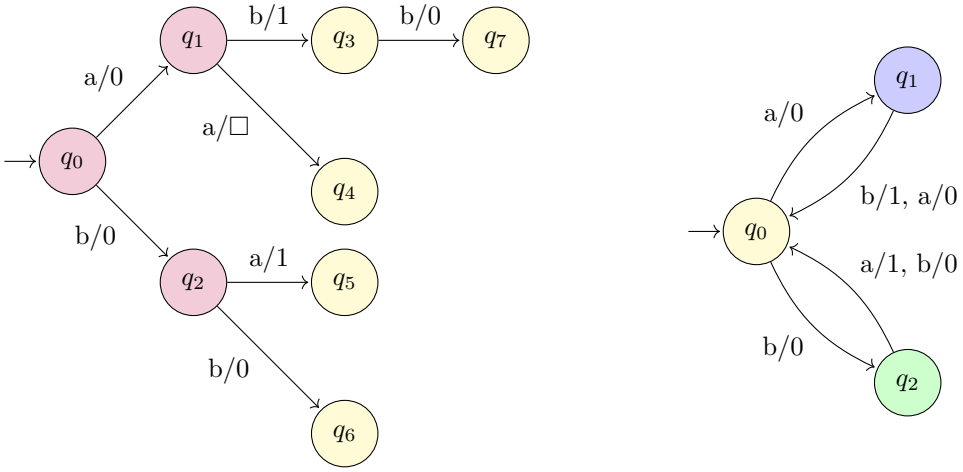


(a) Observation Tree $\mathcal{T}$.

(b) Hypothesis $\mathcal{H}_3$.

Figure 3.4: Observation tree $\mathcal{T}$ and hypothesis $\mathcal{H}_3$ after third learning round.

We submit this hypothesis to the teacher, who will return a counterexample, say abb. Via output query abb, we find out that we have a new state q_7 with $q_3 \xrightarrow{b} q_7$ outputting 0. Witness b shows that states q_1 and q_3 are apart, but q_3 is not apart from q_0. We apply the validity rule once more, and the SMT solver returns a final hypothesis $\mathcal{H}_4$ shown in Figure 3.5b.



(a) Observation Tree $\mathcal{T}$.

(b) Hypothesis $\mathcal{H}_4$.

Figure 3.5: Observation tree $\mathcal{T}$ and hypothesis $\mathcal{H}_4$ after fourth learning round.

This hypothesis is equivalent to the target machine $\mathcal{M}$ (i.e., they are consistently bisimilar) and thus the algorithm is finished. The final observation tree $\mathcal{T}$ is shown in Figure 3.5a.

Chapter 4

Implementation and Standard Evaluation

In order to properly confirm correctness of $L^{\#}_{\square}$ and test its performance, an implementation of the algorithm had to be made. For this, we modified and built on top of the $L^{\#}_{\square}$ implementation for DFAs [9] in order to make the algorithm work on Mealy machines. This transition primarily involved:

- Modifying the System Under Learning (SUL) to handle incompleteness under Mealy machines. This includes functionality for the SUL to return the last output symbol of sequences, instead of the single accepting/rejecting boolean output characteristic of the DFA implementation.
- Extending `MealyNode` class from the AALpy active automata learning library [15] by incorporating properties from the $L^{\#}_{\square}$ for DFAs `MooreNode` implementation, specifically storing each node's access sequence and adding functionality to track whether a subtree contains a known, non-$\square$ transition output.
- Reformulating the Z3 SMT solver constraints to work for Mealy output strings rather than DFA language acceptance.
- Creating a new equivalence oracle based on AALpy's `PerfectKnowledgeEqOracle` using consistent bisimulation to check for Mealy machine equivalence/counterexamples, treating $\square$ as a wildcard.

The complete source code of this $L^{\#}_{\square}$ Mealy algorithm implementation is publicly available on GitHub.[1]

4.1 Experimental Setting

Using our implementation of $L^{\#}_{\square}$, we can evaluate its empirical performance by comparing it directly to standard $L^{\#}$ via the AALpy framework. Because $L^{\#}$ strictly requires a complete

[1] `https://github.com/WoJoosen/l-sharp-square-mealy`

teacher, whereas $L^{\#}_{\square}$ is designed to handle both complete and incomplete teachers, we establish a baseline by benchmarking both algorithms against complete teachers.

Our experimental evaluation is divided into two phases:

1. **Protocol Benchmarks:** We compare both algorithms on five standard Mealy machines commonly used in the automata learning community, retrieved from the Radboud University Automata Wiki [16]. Our models include a toy example (`coffeemachine`) alongside four real-world protocol implementations: `tls-1.2-openssl-1.1.1`, `mosquitto_two_client`, `haraka`, and `DropBear2`. The complexity of these models is summarised in Table 4.1

Model	**States ($\lvert Q\rvert$)**	**Alphabet Size ($\lvert\Sigma\rvert$)**
coffeemachine	6	4
tls-1.2-openssl-1.1.1	8	11
mosquitto_two_client	16	9
haraka	20	31
DropBear2	29	13

Table 4.1: Model Complexity of the Protocol Benchmarks.

2. **Random Scaling Benchmarks:** We analyse algorithmic scaling behaviour by generating random complete and connected Mealy machines using AALpy's `generate_random_mealy_machine` function. We vary the number of states ($|Q| \in \{10, 20, 30, 40, 50, 60\}$) and the input alphabet size ($|\Sigma| \in \{2, 4, 8, 16\}$), where the output alphabet size is identical to the input alphabet size ($|\Gamma| = |\Sigma|$). To avoid statistical anomalies resulting from random generation, we generate 5 distinct Mealy machines using unique random seeds for each individual ($|Q| \times |\Sigma|$) configuration and use the averaged metrics.

For both phases, the performance is evaluated across four metrics: the number of output queries, the number of equivalence queries, the total number of System Under Learning (SUL) steps, and the total execution time in seconds. To ensure fairness, equivalence queries are answered deterministically using model comparison by computing a distinguishing sequence of minimal length. Specifically, for $L^{\#}$ we use the `PerfectKnowledgeEqOracle` (based on standard bisimulation), and for $L^{\#}_{\square}$ we use our `IncompleteKnowledgeEqOracle` (based on consistent bisimulation). For use on complete models, both oracles are functionally equivalent. Moreover, we use the default AALpy library configurations for $L^{\#}$ which utilise the `SepSeq` (separating sequences) extension rule and the `ADS` (Adaptive Distinguishing Sequence) separation rule. Under the `SepSeq` extension rule, standard $L^{\#}$ actively appends witness sequences to frontier exploration queries. Subsequently, the `ADS` separation rule resolves apartness by using an adaptive decision tree to distinguish multiple candidates simultaneously.

All experiments were executed on a single machine running Windows 11, using Python 3.13.6 and AALpy 1.5.3. The underlying SMT constraints for $L^{\#}_{\square}$ were solved using the Z3 theorem prover inside PySMT (version 0.9.6). The hardware specifications of the evaluation environment are detailed in Table 4.2.

Component	Specification
CPU	AMD Ryzen 5 3600 @ 3.60GHz (1 Core allocated)
RAM	24 GB DDR4 @ 3000 MHz

Table 4.2: Hardware Specifications of the Benchmarking Environment.

4.2 Results

In this section we will discuss the results of the two types of benchmarks previously introduced. We start with our *Protocol Benchmarks*, of which the results are visible in Figure 4.1.

4.2.1 Protocol Benchmarks

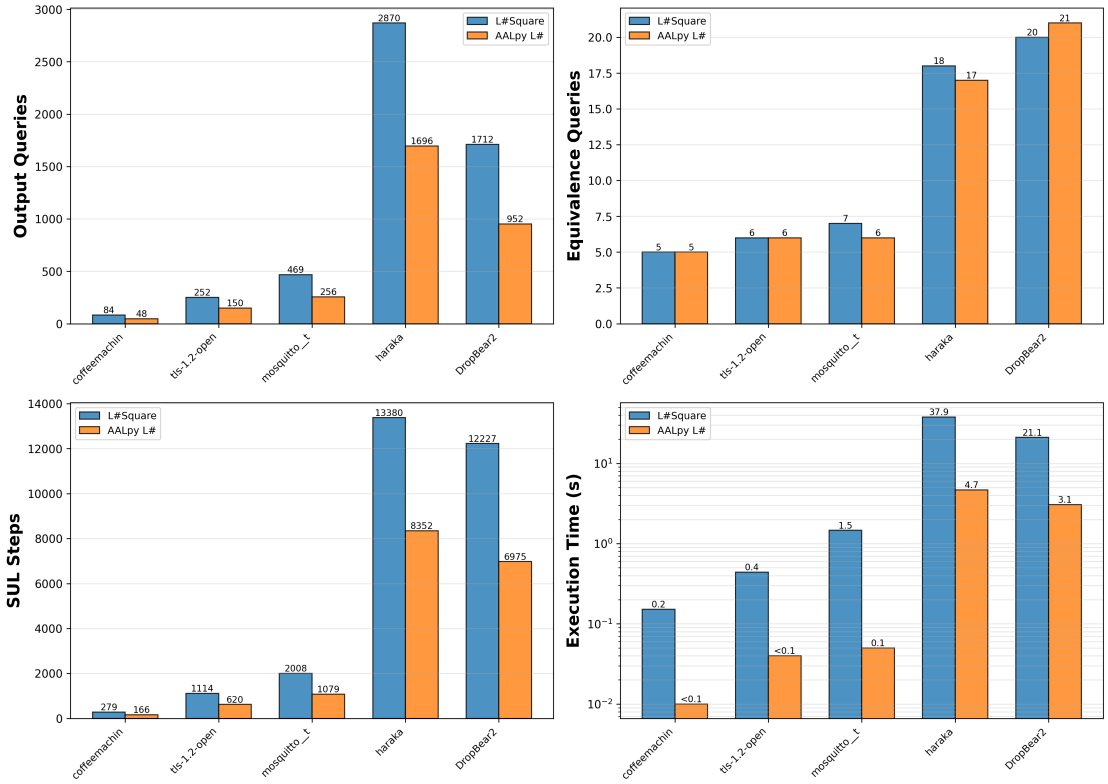


Figure 4.1: Protocol Benchmarks for $L^{\#}_{\square}$ vs. $L^{\#}$ on complete Mealy machines.

Let us start with discussing output queries and SUL steps. It is immediately clear that $L^{\#}_{\square}$ requires on average about 1.75× as many output queries as standard $L^{\#}$. This is expected, as the strategy of extension for $L^{\#}_{\square}$ is a more exhaustive method, where we pose an output query for every possible word up to length $n - |S| + 1$ from the basis states. Conversely, $L^{\#}$ uses a lazy, incremental method to build the frontier, resulting in a lower overall output query amount.

Interestingly, analysing the ratio of total SUL steps to output queries reveals that standard $L^{\#}$ executes longer sequences on average than $L^{\#}_{\square}$. This disparity arises from their differing explo-

ration strategies. As described in Section 4.1, the `SepSeq` optimisation employed by standard $L^\#$ actively appends witness sequences to frontier exploration queries, prioritising longer sequences to accelerate exploration. Conversely, the query sequences in $L^\#_\square$ are composed of a basis state's access sequence plus the $n - |S| + 1$ lookahead suffix. This accounts for the difference in the SUL steps / output queries ratio.

Furthermore, when analysing the number of equivalence queries, we observe that both $L^\#_\square$ and $L^\#$ pose a comparable total number of equivalence queries across the benchmarks. This similarity is expected, as the total number of equivalence queries for both algorithms is bounded by the number of states of the target machine being learned. However, minor variations and fluctuations in the equivalence query count do occur. These deviations are caused by the way the SMT solver works in $L^\#_\square$: while standard $L^\#$ resolves state merges deterministically, the SMT solver evaluates the entire observation tree simultaneously to find a minimal hypothesis. Depending on the observations, the solver may occasionally generate a hypothesis that resolves state ambiguities early, or conversely, it may yield a candidate that requires an additional learning round to find the correct target behaviour.

Finally, we can see that $L^\#_\square$ consistently requires more execution time than $L^\#$. This is caused by the computational expense of the SMT solver, which is a necessary trade-off to enable learning from incomplete teachers (see Chapter 5).

While $L^\#_\square$ successfully handles the larger `DropBear2` model, scaling these benchmarks to significantly larger models exceeding 65 states, such as the `BitVise` SSH server model, revealed the current scalability boundaries of the algorithm. Because of the exponential growth of the underlying observation tree as model complexity increases, these larger models result in extreme execution times.

4.2.2 Random Scaling Benchmarks

For our *random scaling benchmarks*, we will discuss important trends that are visible in our results in Figure 4.2, rather than discuss each individual metric in detail.

Looking at the first and third row of graphs in Figure 4.2, we see that both lines mirror each other perfectly as they scale upwards. The lines maintain the exact same proportional gap whether the machine has 10 states or 60 states, or an alphabet size of 2 or 16. This shows that the query overhead we discussed in our *Protocol Benchmarks* is consistent across different machines and scales predictably with machine complexity.

When analysing the execution time in the last row of graphs, we see that both standard $L^\#$ and $L^\#_\square$ move upwards significantly as machine complexity increases. Because of the logarithmic scale, the lines for both algorithms trend upwards in almost a parallel manner, preserving a relatively consistent gap of approximately one to one-and-a-half orders of magnitude. This performance gap can be explained by the computational complexity of the SMT solver, like we did for our *Protocol Benchmarks*. As the number of states and alphabet size grows, the size of the observation tree grows as well, in turn significantly increasing the number of constraints for the SMT solver to solve. If we look even closer, we see that the variance in execution time increases as the number of states increases. This shows that the SMT solver is highly sensitive to the randomness of the target machine at higher complexities.

Finally, analysing the second row of graphs, we can see completely different behaviour in the number of equivalence queries as alphabet size grows. At an alphabet size of 2, we see that $L^\#$ consistently requires a lot more equivalence queries than $L^\#_\square$. At alphabet sizes $4, 8$, and 16, the

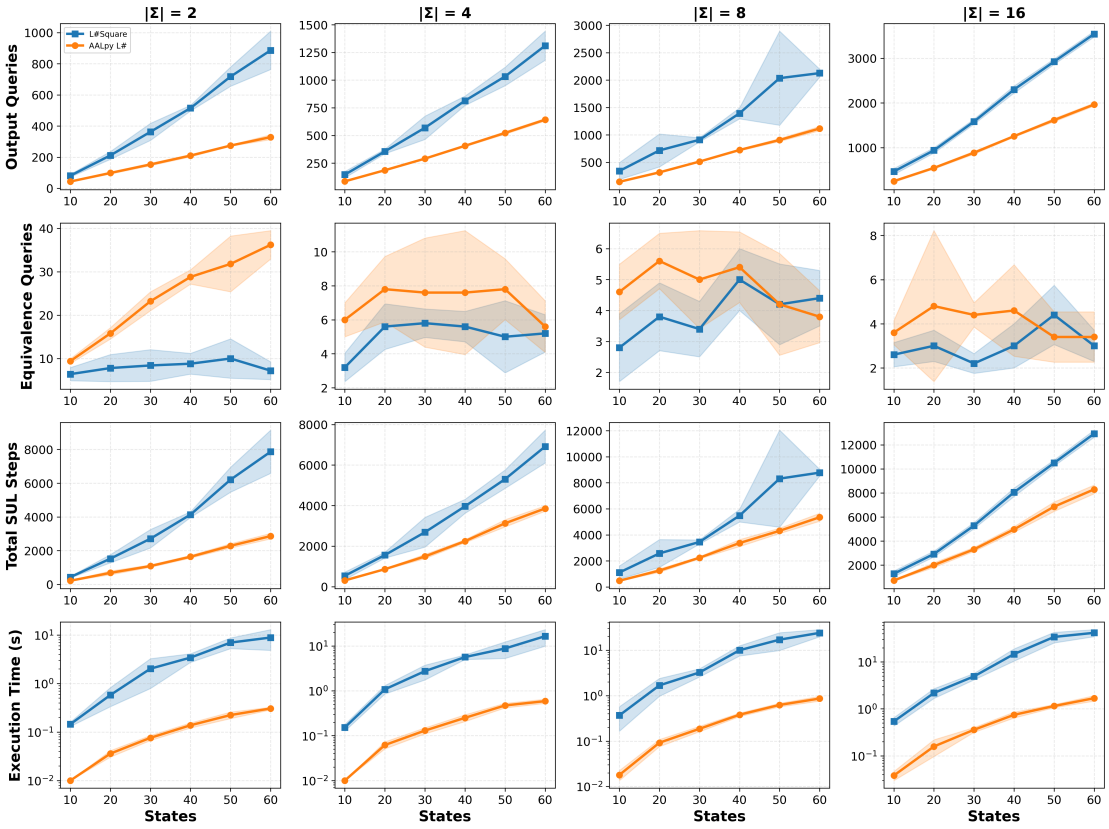


Figure 4.2: Random Scaling Benchmarks for $L^{\#}_{\square}$ vs. $L^{\#}$ on complete Mealy machines.

lines overlap and converge, and we see that standard $L^{\#}$ sometimes requires fewer equivalence queries than $L^{\#}_{\square}$. We can explain this behaviour as follows: when the alphabet is small (e.g., size 2), standard $L^{\#}$'s lazy exploration struggles to find apartness in states cleanly, leading to bad hypotheses and many counterexamples. However, as the alphabet grows larger, the denser data naturally provides a lot of distinguishing information anyway, meaning both algorithms converge to a relatively similar number of equivalence queries.

In summary, the empirical evaluation confirms that our Mealy implementation of $L^{\#}_{\square}$ on complete teachers preserves the properties of the original $L^{\#}$ algorithm. While $L^{\#}_{\square}$ experiences a linear and predictable overhead in output queries and an exponential increase in execution time due to its use of an SMT solver, it performs comparably to or better than standard $L^{\#}$ in terms of equivalence queries. Most importantly, these evaluations establish baseline performance on complete machines, verifying that the computational overhead of the SMT solver is a predictable trade-off for the algorithm's main objective: the functionality to handle incomplete teachers.

Chapter 5

Evaluation on Incomplete Teacher

While Chapter 4 evaluated the baseline performance of our $L^{\#}_{\square}$ implementation against standard $L^{\#}$ on complete machines, the real strength of $L^{\#}_{\square}$ lies in its capacity to learn transition complete models using an incomplete teacher. To understand this capability and evaluate how $L^{\#}_{\square}$ behaves when dealing with partial knowledge, this chapter presents a number of case studies and benchmarks using our $L^{\#}_{\square}$ Mealy implementation.

5.1 Experimental Setting

To evaluate the behaviour of our $L^{\#}_{\square}$ Mealy implementation under these partial conditions, we adapt the benchmarking environment from Chapter 4 to incorporate an incomplete teacher. Because standard $L^{\#}$ cannot process incomplete machines, a direct algorithmic comparison is no longer possible. Instead, this evaluation focuses on the stand-alone performance and behaviour of $L^{\#}_{\square}$ across two distinct phases:

1. **Practical Case Studies:** We examine four case studies ranging from toy examples to data structures to observe how the algorithm learns and minimises incomplete systems.
2. **Random Scaling Benchmarks:** We analyse algorithmic scaling behaviour by generating random transition-complete and connected Mealy machines using AALpy's `generate_random_mealy_machine` function. Similar to section 4.2.2, we vary the number of states ($|Q| \in \{10, 20, 30, 40, 50, 60\}$) and input alphabet size ($|\Sigma| \in \{2, 4, 8, 16\}$), where the output alphabet size is identical to the input alphabet size ($|\Gamma| = |\Sigma|$). We then randomly select a percentage ($p \in \{0\%, 20\%, 40\%, 60\%\}$) of transitions and change their output symbols to be "don't care" ($\square$). Again, to avoid statistical anomalies resulting from random generation, we generate 5 distinct Mealy machines using unique random seeds for each individual ($|Q| \times |\Sigma| \times p$) configuration and use the averaged metrics.

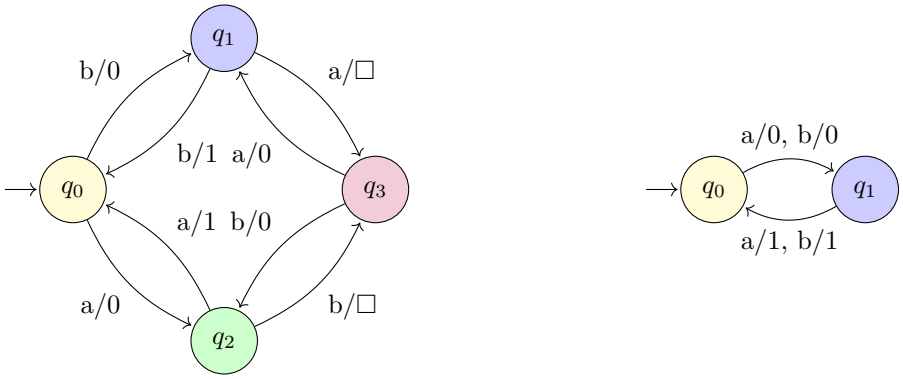
Both phases make use of our custom `IncompleteKnowledgeEqOracle` equivalence oracle based on consistent bisimulation. All benchmarks are executed using the same hardware configuration, operating system, and software dependencies (including Python, AALpy, and the Z3 SMT Solver inside PySMT) as detailed in Section 4.1 (see Table 4.2).

5.2 Case Studies

In this section, we present four case studies designed to analyse the behaviour of $L^{\#}_{\square}$ when learning practical machines with incomplete knowledge. Instead of evaluating raw performance scaling, these benchmarks focus on the semantic implications of state minimisation when the teacher returns "don't know" ($\square$) outputs.

5.2.1 Variation of Preliminaries Example

Our first case study is a variation of the complete Mealy machine introduced in the preliminaries (see bottom of Figure 2.1), modified to include unknown outputs, making it a transition-complete Mealy machine. We change transitions $q_1 \xrightarrow{a/0} q_3$ to $q_1 \xrightarrow{a/\square} q_3$, and $q_2 \xrightarrow{b/0} q_3$ to $q_2 \xrightarrow{b/\square} q_3$ to get the variation in Figure 5.1a. Letting $L^{\#}_{\square}$ learn this model results in the hypothesis shown in Figure 5.1b.



(a) Variation of machine $\mathcal{M}$ in Figure 2.1.

(b) Hypothesis learned by $L^{\#}_{\square}$ for the machine in Figure 5.1a.

Figure 5.1: Case study of variation target machine and learned machine.

We can observe two important characteristics in the result:

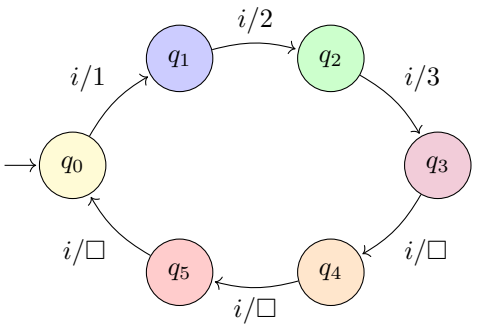
- **Merging of q_1 and q_2:** In the original target machine, states q_1 and q_2 served symmetric roles, but were separated because of their differing transitions to q_3 on inputs a and b. By replacing those transitions with undefined outputs ($a/\square$ and $b/\square$), the two states were not distinct anymore and $L^{\#}_{\square}$ recognised that q_1 and q_2 could be merged into a single state, labelled q_1 in the learned hypothesis.
- **Merging of q_0 and q_3:** Because the transitions leading into state q_3 became incomplete ($\square$), the algorithm was never able to isolate q_3 as a unique (basis) state. Any behaviour that previously went through q_3 has been taken over by the two remaining states in the learned hypothesis, resulting in the merging of states q_0 and q_3.

It is important to note that the goal of $L^{\#}_{\square}$ is to always find the smallest (minimal) machine that does not contradict its observations. Because of this, the learned hypothesis is semantically different from the original target machine. By introducing incomplete outputs, the machine became transition-complete. $L^{\#}_{\square}$ filled these "unknowns" with the simplest possible output symbols

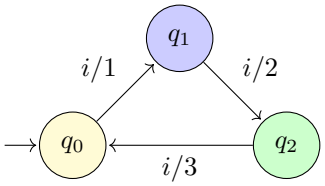
that satisfied its observations. This shows that $L^{\#}_{\square}$ will always generalise undefined behaviour to obtain the smallest possible machine.

5.2.2 Modulo Counter Machine

In our second case study we will analyse $L^{\#}_{\square}$'s behaviour on learning incomplete modulo counter machines. These machines consist of a loop of N states, where the last k states have outgoing transitions containing an unknown output ($\square$). An example can be seen in Figure 5.2a for $N = 6$ and $k = 3$, together with the hypothesis $L^{\#}_{\square}$ learned in Figure 5.2b.



(a) Modulo Counter machine with $N = 6$ and $k = 3$.



(b) Hypothesis learned by $L^{\#}_{\square}$ for the machine in Figure 5.2a.

Figure 5.2: Case study of the modulo counter machine and its corresponding learned machine.

This example shows that $L^{\#}_{\square}$ recognises that this loop of 6 states (of which 3 have unknown transitions) can be minimised into a machine of only 3 states, because the unknown outputs allow the algorithm to simplify the loop. Since the unknown outputs can match any symbol, $L^{\#}_{\square}$ repeats the $1 \to 2 \to 3$ sequence in a smaller but behaviourally identical loop.

Now, we can analyse how the algorithm behaves under different values of N and k. We let N run from 4 to 18, and k from 2 to 16. We skip any configurations where $k \geq N$, and we impose a timeout of 120 seconds after which we skip the corresponding configuration. For each configuration we record the size of the learned hypothesis. These results are shown in Table 5.1.

We can observe several interesting characteristics in the results:

- **Composite Number Factoring:** For composite values of N, as k increases we can see that $L^{\#}_{\square}$ minimises the target machine into the factors of N. For example, for $N = 12$, as k increases, the algorithm minimises the machine cleanly into the factors of N: $12 \to 6 \to 4 \to 3 \to 2 \to 1$. Similar behaviour can be observed for all other composite values of N, like $18, 16$, and 14.
- **Prime Complexity:** $L^{\#}_{\square}$ behaves completely differently when encountering prime values of N, as it is unable to minimise the machine at all, until we reach $k = N - 1$. For example, for $N = 7$, the learned state size stays at 7 consistently, until we reach $k = 6$, where $L^{\#}_{\square}$ minimises the machine to a state size of 1. The same behaviour can be observed for other primes, like $N = 11$, where the learned state size stays at 11, until $L^{\#}_{\square}$ is able to minimise the machine to a size of 1 for $k = 10$. Because a prime number has no factors (other than

itself and 1), the algorithm can not find any sub-loops. It has either too much information, where it needs to learn a hypothesis of size N, or so little information that it is able to minimise the machine into a single-state machine.

– **Timeouts:** For some larger prime values of N and larger values of k, we can see that timeouts will occur at some point. For those specific configurations, the number of known states that are left is small. Because a prime loop with $k < N-1$ cannot be minimised to a divisor of N, a valid hypothesis will always have size N. Because $L^{\#}_{\square}$ tries to find a minimal hypothesis, it will first try for output sizes of $3, 4, 5, \ldots$ until it finally finds the correct hypothesis of size N. These attempts are computationally expensive and are therefore the cause of the timeouts. Only when we reach $k = N - 1$, the algorithm immediately finds a hypothesis of size 1.

N \ k	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
4	2	1	—	—	—	—	—	—	—	—	—	—	—	—	—
5	5	5	1	—	—	—	—	—	—	—	—	—	—	—	—
6	6	3	2	1	—	—	—	—	—	—	—	—	—	—	—
7	7	7	7	7	1	—	—	—	—	—	—	—	—	—	—
8	8	8	4	4	2	1	—	—	—	—	—	—	—	—	—
9	9	9	9	9	3	3	1	—	—	—	—	—	—	—	—
10	10	10	10	5	5	5	2	1	—	—	—	—	—	—	—
11	11	11	11	11	11	11	11	11	1	—	—	—	—	—	—
12	12	12	12	12	6	6	4	3	2	1	—	—	—	—	—
13	13	13	13	13	13	13	13	13	**TO**	**TO**	1	—	—	—	—
14	14	14	14	14	14	7	7	7	7	7	2	1	—	—	—
15	15	15	15	15	15	15	15	15	5	5	3	3	1	—	—
16	16	16	16	16	16	16	8	8	8	8	4	4	2	1	—
17	17	17	17	17	17	17	17	17	17	**TO**	**TO**	**TO**	**TO**	**TO**	1
18	18	18	18	18	18	18	18	9	9	9	6	6	6	3	2

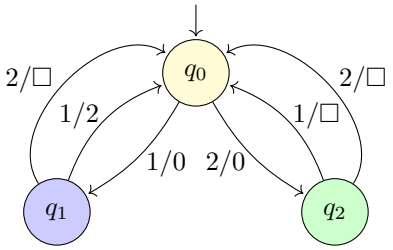
*Note: **TO** indicates a computational timeout (> 120s); — indicates unfeasible configurations ($k \geq N$).

Table 5.1: Learned number of states of $L^{\#}_{\square}$ under varying loop sizes (N) and transitions with unknown outputs (k)

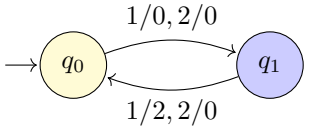
The semantic behaviour of the output machines varies significantly depending on the specific configuration of N and k. For configurations where the number of remaining known outputs ($N-k$) is a divisor of N, $L^{\#}_{\square}$ simply minimises the machine into a sub-loop of the original target machine. However, for composite values of N, where a large k leaves many outputs undefined (such as $N = 14, k = 11$), the output symbols become unpredictable. In these cases, the algorithm is forced to minimise to the next divisor (e.g., 7 states) to satisfy the known outputs. Because the target machine provides no information for the remaining unknown outputs (e.g., the last 4 outputs), $L^{\#}_{\square}$'s SMT solver fills the unknown outputs with arbitrary symbols from the output alphabet. Prime values of N behave in similar ways: since their only divisors are N and 1, the algorithm has no intermediate divisors to which the machine can be minimised, forcing the hypothesis to remain at size N until it fully minimises into a single-state machine at $k = N - 1$.

5.2.3 Adder Machine

Our next case study will analyse the behaviour of $L^{\#}_{\square}$ on learning what we will call "adder machines". Let us take for example an adder machine with a max sum of $n = 2$. Then, this machine will have an input alphabet of $\Sigma = \{1, 2\}$ and output alphabet of $\Gamma = \{0, 1, 2\}$. Its behaviour can be described as follows: for any first input $i \in \Sigma$, the machine will output a 0. Then, for any second input $j \in \Sigma$, the machine will output the sum of both inputs $(i + j)$, except when the sum exceeds $n = 2$. In that case, the machine will output "don't know" ($\square$). The adder machine with max sum of $n = 2$ can be seen in Figure 5.3a, together with the hypothesis $L^{\#}_{\square}$ learned in Figure 5.3b.



(a) Target adder machine with $n = 2$.



(b) Hypothesis learned by $L^{\#}_{\square}$ for the machine in Figure 5.3a.

Figure 5.3: Case study of the adder machine and its corresponding learned machine.

The behaviour of $L^{\#}_{\square}$ for this example is essentially identical to its behaviour for different values of n. We will analyse the results for different values of n, ranging from 2 to 20. For each value of n, we record the size of the target machine, the size of the learned hypothesis, and the alphabet size. Additionally, we also record the number of output queries and equivalence queries, as well as the total execution time. These results are shown in Table 5.2.

n	Target Size	Hyp. Size	$\|\Sigma\|$	O. Queries	Eq. Queries	Time (s)
2	3	2	2	11	2	0.048
3	4	3	3	35	2	0.023
4	5	4	4	77	2	0.042
5	6	5	5	144	2	0.072
6	7	6	6	242	2	0.120
7	8	7	7	377	2	0.201
8	9	8	8	555	2	0.326
9	10	9	9	782	2	0.486
10	11	10	10	1064	2	0.709
11	12	11	11	1407	2	1.047
12	13	12	12	1817	2	1.394
13	14	13	13	2300	2	2.138
14	15	14	14	2862	2	2.738
15	16	15	15	3509	2	4.019
16	17	16	16	4247	2	5.051
17	18	17	17	5082	2	6.835
18	19	18	18	6020	2	9.175
19	20	19	19	7067	2	12.100
20	21	20	20	8229	2	15.424

Table 5.2: Metrics for the adder machine under varying max sums (n)

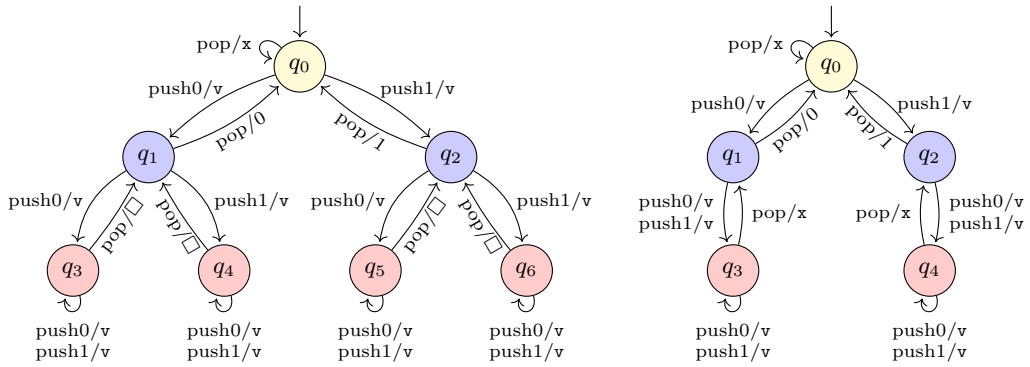
From these results, we can see that there is a clear minimisation pattern. For every single value of n, the hypothesis size is exactly 1 less than the target size. Additionally, for every value of n, the alphabet size is equal to the hypothesis size. This can be explained as follows: in the target machine, state q_n (the last state) represents the state where the first input was n. From state q_n, every transition to a second input will result in a sum greater than n. Therefore, all outgoing transitions from q_n have an unknown output ($\square$).

Because q_n only has unknown outputs, it does not have any concrete behaviour. $L^{\#}_{\square}$ realises this and sees that q_n can be merged directly into another state without causing a contradiction. States q_1 through q_{n-1} cannot merge because they still contain at least one concrete output. Therefore, $L^{\#}_{\square}$ consistently prunes exactly 1 state from the target machine in each final hypothesis.

Apart from $L^{\#}_{\square}$'s behaviour, Table 5.2 also contains valuable metrics related to performance. While the number of equivalence queries remains the same for every value of n, the number of output queries seems to scale cubically. This growth occurs because increasing n increases both the number of states as well as the input alphabet size. Total execution time seems to grow exponentially.

5.2.4 Bounded Boolean Stack

In our final case study, we will analyse the behaviour of $L^{\#}_{\square}$ on learning bounded boolean stack machines. These machines have the structure of a stack with only two input symbols, but after a certain depth n, each `pop` operation has "don't know" ($\square$) as its output. An example for the target machine for $n = 2$ can be found in Figure 5.4a and the resulting hypothesis in Figure 5.4b.



(a) Target bounded boolean stack machine with $n = 2$.

(b) Hypothesis learned by $L^{\#}_{\square}$ for the machine in Figure 5.4a.

Figure 5.4: Case study of the bounded boolean stack machine and its corresponding learned machine.

As is visible in Figure 5.4a, the machine acts as a regular boolean stack where outputs `x` and `v` act as an "exception" and a "successful operation", respectively. However, when the machine reaches its maximum depth, all `pop` inputs will output "don't know" (□) and the leaf states `push` any input back to itself to simulate the concept of infinity.

$L^{\#}_{\square}$'s behaviour for learning this machine is identical to its behaviour for all tested values of n. We analyse the results for different values of n, ranging from 1 to 5. For each value of n, we record the size of the target machine and the size of the learned hypothesis. We also record the number of output queries, the number of equivalence queries, and the total execution time. The results are visible in Table 5.3.

n	Target Size	Hyp. Size	O. Queries	Eq. Queries	Time (s)
1	3	1	3	1	0.041
2	7	5	52	3	0.047
3	15	11	134	4	0.142
4	31	23	420	6	0.699
5	63	47	1162	12	6.941

Table 5.3: Metrics for the Bounded Boolean Stack under varying capacities (n)

Analysing the results in Table 5.3 and the machine in Figure 5.4a, we can see an interesting pattern:

It is clear that the target size for each value of n follows the standard equation for boolean stacks: $2^{n+1} - 1$. Then, $L^{\#}_{\square}$ minimises the target machine in an interesting way: for each pair of leaf nodes (q_3 & q_4, and q_5 & q_6 in Figure 5.4a), it merges the two states into one, and replaces the unknown output (□) in their outgoing transitions with an `x` for "exception" (q_3 and q_4 in Figure 5.4b). This result is the same for each value of n, except for $n = 1$. Only in that case is $L^{\#}_{\square}$ able to minimise the machine into a single-state machine containing three self-loops, because

all three states show identical behaviour. For all other values of n, the hypothesis size follows the equation $3 \cdot 2^{n-1} - 1$, meaning that it effectively minimises the target machines into hypotheses with 2^{n-1} fewer states.

What makes $L^{\#}_{\square}$'s hypotheses so interesting is that the resulting machines are smaller, still functional bounded boolean stacks. By turning the unknown output $\square$ into `x`, it essentially treats an overfilled stack like an exception. Consequently, this results in the machine reusing the exact same error behaviour that was already defined for the `pop` transition in q_0. Importantly, because the target machine treats an overfilled stack's output as an unknown output, choosing any other output symbol in place of `x` would have resulted in a hypothesis that is also consistently bisimilar to the target machine. The SMT solver has no knowledge of the target machine's semantics and is unaware that it is modelling a stack. Therefore, the SMT solver's choice of `x` is likely only because it is the first output symbol defined in the target machine.

Table 5.3 also contains metrics related to the performance of $L^{\#}_{\square}$ on learning these machines. We can see that the number of output queries grows exponentially as the value of n increases, similar to the total execution time, which also grows exponentially. This is as expected, because the size of the target machine grows exponentially as well. The growth of equivalence queries seems to grow exponentially too.

5.3 Random Scaling Benchmarks

We will now focus on the performance of $L^{\#}_{\square}$ rather than the specific behaviour of the algorithm. As described in Section 5.1, we will look at the performance metrics of $L^{\#}_{\square}$ for several randomly generated transition-complete Mealy machines containing differing percentages of unknown ($\square$) outputs. We impose a strict timeout of 120 seconds, after which all 5 machines in the configuration will be skipped and their results will not be plotted. We will discuss some notable trends of the results, shown in Figure 5.5.

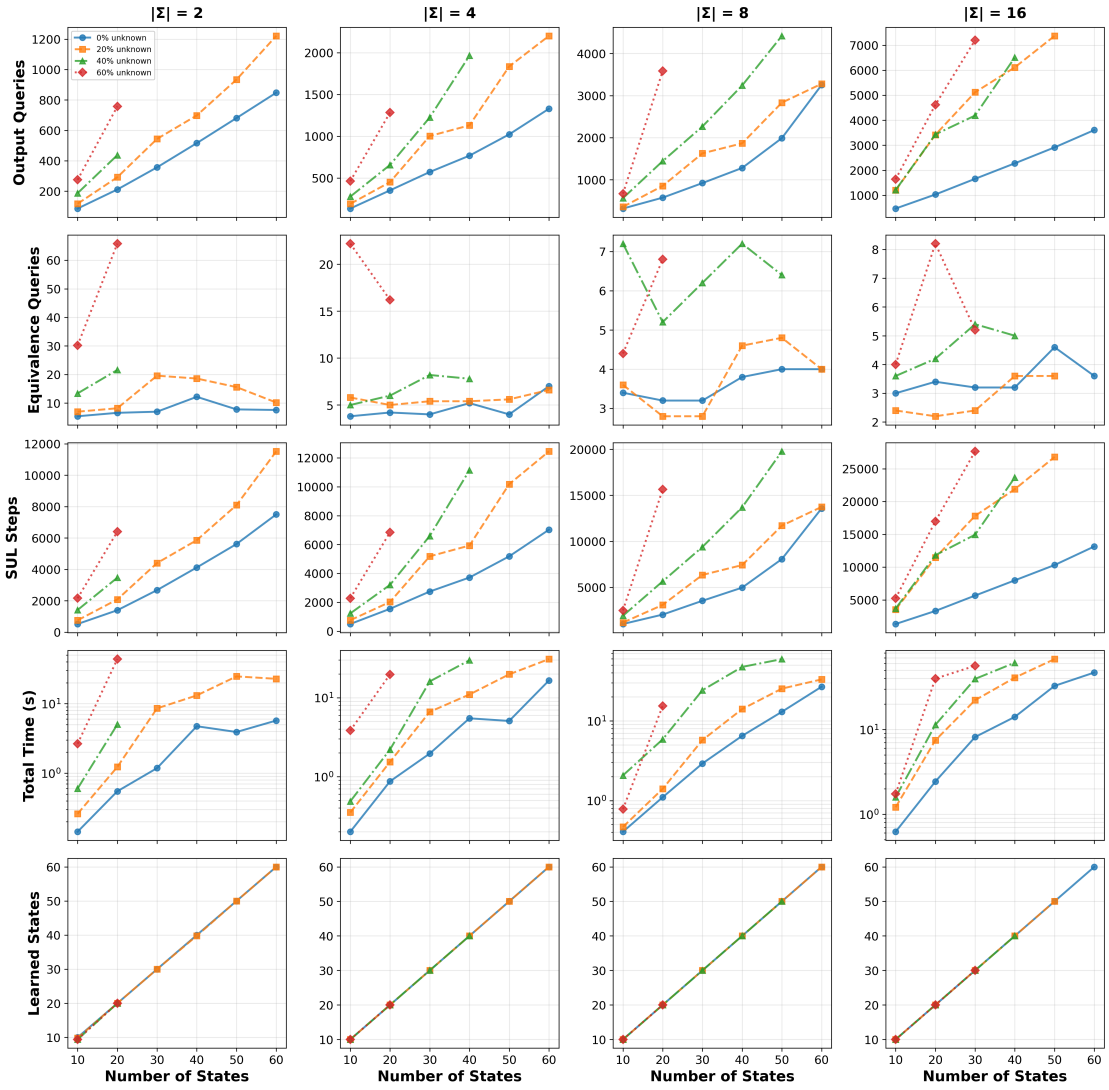


Figure 5.5: Random Scaling Benchmarks for $L_{\square}^{\#}$ on complete and incomplete Mealy machines.

Let us first analyse the impact of the percentage of unknown outputs on $L_{\square}^{\#}$'s performance. While holding the number of states and alphabet size constant, increasing the percentage of unknown outputs results in a clear increase of almost every performance metric. For example, the number of output queries and SUL steps rises significantly, where moving from 0% to 60% unknowns at 20 states and an alphabet size of 4 causes an increase of about 3.6× the number of output queries. The number of equivalence queries rises even more dramatically, where moving from 0% to 60% at 20 states and an alphabet size of 2 causes an increase of about 10× the number of equivalence queries. This occurs because unknown outputs (□) cause many valid hypotheses to exist, resulting in $L_{\square}^{\#}$ performing more equivalence queries to ensure a minimal hypothesis.

Additionally, we can see that $L_{\square}^{\#}$'s total execution time grows exponentially. When the percentage of unknown outputs is 0% or 20%, execution times are low and stable. At 40% and 60%

unknown outputs, $L^{\#}_{\square}$'s SMT solver takes longer to find the minimal correct hypothesis. For example, at 20 states and an alphabet size of 2, the execution time grows from 0.55 seconds at 0% unknown outputs to 43.75 seconds at 60% unknown outputs, an increase of almost 80×.

Because of the exponential growth in execution time, as the number of states, alphabet size, and percentage of unknowns increases, the number of timeouts increases as well. For smaller machines of 10 or 20 states, all iterations of the algorithm succeed, even with 60% unknown outputs. However, once we reach larger machines of 30 states, at 60% unknown outputs over half the iterations reach timeouts for alphabet sizes 2 and 4, and for an alphabet size of 8, no iterations are completed in time. When we reach machines of size 40-60, $L^{\#}_{\square}$ can only consistently learn machines with 0% or 20% unknowns. At 40% unknowns or higher, only some configurations are able to be learned in time.

Moreover, the size of the machines that $L^{\#}_{\square}$ learned stays extremely close to the size of the target machines. Only for a select few configurations was $L^{\#}_{\square}$ able to minimise the machine to have 1 or 2 fewer states. This behaviour occurs because for completely randomized Mealy machines, changing outputs to be unknown (□) rarely results in opportunities for minimisation, because random states are naturally more distinct from each other. Therefore, introducing unknown outputs in these machines rarely results in the merging of two states.

Finally, as discussed in Section 4.2.2, it is again clear that $L^{\#}_{\square}$ is sensitive to the randomness of the target machine at higher complexities. For example, at 30 states, alphabet sizes $2, 4$, and 8, and 60% of unknown outputs, results have been omitted because of timeouts. However, for the same configuration at alphabet size 16, all 5 iterations were completed on time. This suggests that the randomly generated machines for the alphabet size 16 benchmarks happened to yield simpler or less complex constraints for $L^{\#}_{\square}$'s SMT solver, allowing all iterations to complete on time.

5.4 Discussion of Results

While Sections 5.2 and 5.3 presented results for specific evaluations, this section combines those results to evaluate how incomplete knowledge impacts $L^{\#}_{\square}$'s efficiency and performance.

In the case studies, introducing unknown outputs allowed $L^{\#}_{\square}$ to find smaller hypotheses than the target machines. This occurs because non-random or structured systems contain specific behavioural patterns. Hiding certain outputs results in a simplification of these behaviours, creating opportunities for $L^{\#}_{\square}$ to merge states. Conversely, in the random scaling benchmarks, the learned hypothesis almost always stayed the same size as the target machine. In a completely randomised machine, states are naturally rich and provide a lot of distinguishing information. Hiding an output merely creates local uncertainty, but almost never creates an opportunity for $L^{\#}_{\square}$ to merge states. Therefore, $L^{\#}_{\square}$'s minimisation benefits are heavily dependent on the structure of the target machine.

This difference in machine structure also explains the query behaviour we observed in the random scaling benchmarks. Here, because states cannot be merged, the number of output queries and equivalence queries grows rapidly as the percentage of unknown outputs increases. Because $L^{\#}_{\square}$ has to maintain its observation tree without concrete output symbols to prove states are apart, it is forced to pose many more exploration queries. Additionally, because so many outputs are unknown, there are many different valid minimal hypotheses that can satisfy all constraints. This often leads to many counterexamples and therefore a higher number of equivalence queries.

Finally, looking at both the case studies and the random scaling benchmarks shows a clear limit caused by $L^{\#}_{\square}$'s SMT solver when dealing with incomplete machines. When a teacher provides complete outputs, we have a full observation tree. However, introducing unknown outputs forces the SMT solver to search through many possibilities to find a minimal correct hypothesis that satisfies all constraints. As established in Remark 2.13, this problem is NP-complete, which explains the exponential growth in execution time we observe.

In the case studies, this caused some systems like the modulo counter to hit bottlenecks when trying to find minimal hypotheses for specific prime loop sizes. In the random benchmarks, this caused total execution times to grow exponentially, resulting in frequent timeouts for more complex machines. At this point, the performance becomes sensitive to the specific layout of the target machine. This is shown by the case where smaller random alphabets timed out completely at 30 states, but the alphabet size 16 configuration completed all iterations on time because its complexity happened to be simpler. Ultimately, while the performance of $L^{\#}_{\square}$ decreases as the complexity of the target machine grows, the structure of the target machine has a large impact as well.

Chapter 6

Conclusion and Discussion

In this research, we have adapted the $L^{\#}_{\square}$ active automata learning algorithm to handle the learning of incomplete Mealy machines. We showed that by modifying the underlying observation tree and System Under Learning, redefining the SMT solver constraints, and creating a new equivalence oracle, $L^{\#}_{\square}$ is able to successfully and reliably learn Mealy machines from an incomplete teacher. While $L^{\#}_{\square}$ consistently performs worse than regular $L^{\#}$ on complete models, we demonstrated that $L^{\#}_{\square}$ produces correct results for all evaluated benchmarks, even when learning complete models. More importantly, our research shows that $L^{\#}_{\square}$ correctly minimises incomplete Mealy machines with diverse behaviours and that its performance is significantly dependent on the complexity and structure of the target machine.

Our findings demonstrate that $L^{\#}_{\square}$ is suited for learning systems where the result of an observation is sometimes unclear or unobservable. A concrete application for this is found in Partially Observable Markov Decision Processes (POMDPs). Recent research uses active learning algorithms like L^* or `ALERGIA` to extract finite-state controllers for POMDPs with hidden models [4, 7]. Given its natural ability to handle incompleteness, $L^{\#}_{\square}$ offers a promising alternative to optimise these existing POMDP frameworks. Furthermore, while we tested $L^{\#}_{\square}$ on different machines with up to 60 states and an alphabet size of 16 symbols, introducing more unknown outputs causes the execution time to increase significantly, making the algorithm potentially unsuitable for larger or more complex machines with a large amount of unknown outputs. Additionally, while $L^{\#}_{\square}$'s focus on minimality is considered a key feature, in some cases it can introduce limitations. When faced with unknown outputs, $L^{\#}_{\square}$'s SMT solver arbitrarily chooses output symbols to replace them with, which can yield machines that are semantically different from the target machine.

Building upon the implementation and evaluation presented in this research, several directions remain open for future work. First, to ensure that the algorithm works as intended, future research could prove correctness of $L^{\#}_{\square}$ mathematically. While we expect that this proof is highly similar to the correctness proof given for the $L^{\#}_{\square}$ DFA algorithm [10], it must be adapted to the setting of Mealy machines. Additionally, to address the execution time bottlenecks of $L^{\#}_{\square}$'s SMT solver, future research could explore even further optimisation of the solver's constraints. Related to this, it would be valuable to extend our benchmarks to evaluate models with even higher complexities (e.g., systems exceeding 100 states). Although learning such models requires extensive execution times, comparing how $L^{\#}_{\square}$ and standard $L^{\#}$ scale at these complexities could result in interesting insights. Finally, while our evaluations returned minimal counterexamples, $L^{\#}_{\square}$ is capable of processing counterexamples of any length. Future research could compare the performance of both algorithms in settings where equivalence queries return exceptionally long, non-minimal counterexamples, such as those generated by random walks.

Bibliography

[1] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75 (2):87–106, November 1987.

[2] Francesco Bergadano and Stefano Varricchio. Learning behaviors of automata from multiplicity and equivalence queries. *SIAM Journal on Computing*, 25(6):1268–1280, December 1996. ISSN 1095-7111. doi: 10.1137/s009753979326091x. URL http://dx.doi.org/10.1137/S009753979326091X.

[3] Benedikt Bollig, Peter Habermehl, Carsten Kern, and Martin Leucker. Angluin-style learning of NFA. In *IJCAI International Joint Conference on Artificial Intelligence*, pages 1004–1009, 07 2009.

[4] Debraj Chakraborty, Anirban Majumdar, Prince Mathew, Sayan Mukherjee, and Jean-François Raskin. Synthesizing POMDP policies: Sampling meets model-checking via learning, 2026. URL https://arxiv.org/abs/2605.14440.

[5] Yu-Fang Chen, Azadeh Farzan, Edmund M. Clarke, Yih-Kuen Tsay, and Bow-Yaw Wang. *Learning Minimal Separating DFA's for Compositional Verification*, page 31–45. Springer Berlin Heidelberg, 2009. ISBN 9783642007682. doi: 10.1007/978-3-642-00768-2_3. URL http://dx.doi.org/10.1007/978-3-642-00768-2_3.

[6] John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, Stanford University, 1971. URL https://dl.acm.org/doi/10.5555/891883.

[7] David Hudák, Maris F. L. Galesloot, Martin Tappler, Martin Kurečka, Nils Jansen, and Milan Češka. Finite-state controllers for (hidden-model) POMDPs using deep reinforcement learning, 2026. URL https://arxiv.org/abs/2602.08734.

[8] Muhammad Naeem Irfan, Catherine Oriat, and Roland Groz. Angluin style finite state machine inference with non-optimal counterexamples. In *Proceedings of the First International Workshop on Model Inference In Testing*, ISSTA '10, page 11–19. ACM, July 2010. doi: 10.1145/1868044.1868046. URL http://dx.doi.org/10.1145/1868044.1868046.

[9] Jasper Laumen. L#Square DFA Implementation. https://github.com/JLaumen/l-sharp-square-algorithm, 2026.

[10] Jasper Laumen, Leonne Snel, and Frits Vaandrager. An $L^{\#}$ based algorithm for active learning of minimal separating automata, 2026. URL https://arxiv.org/abs/2605.15294.

[11] Martin Leucker and Daniel Neider. *Learning Minimal Deterministic Automata from Inexperienced Teachers*, page 524–538. Springer Berlin Heidelberg, 2012. ISBN 9783642340260. doi:

10.1007/978-3-642-34026-0_39. URL http://dx.doi.org/10.1007/978-3-642-34026-0_39.

[12] O. Maler and A. Pnueli. On the learnability of infinitary regular sets. *Information and Computation*, 118(2):316–326, May 1995. ISSN 0890-5401. doi: 10.1006/inco.1995.1070. URL http://dx.doi.org/10.1006/inco.1995.1070.

[13] Maik Merten, Falk Howar, Bernhard Steffen, and Tiziana Margaria. *Automata Learning with On-the-Fly Direct Hypothesis Construction*, page 248–260. Springer Berlin Heidelberg, 2012. ISBN 9783642347818. doi: 10.1007/978-3-642-34781-8_19. URL http://dx.doi.org/10.1007/978-3-642-34781-8_19.

[14] Mark Moeller, Thomas Wiener, Alaia Solko-Breslin, Caleb Koch, Nate Foster, and Alexandra Silva. Automata Learning with an Incomplete Teacher. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:30, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-281-5. doi: 10.4230/LIPIcs.ECOOP.2023.21. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2023.21.

[15] Edi Muškardin, Bernhard Aichernig, Andrea Pferscher, and Martin Tappler. Aalpy: an active automata learning library. *Innovations in Systems and Software Engineering*, 18: 1–10, 03 2022. doi: 10.1007/s11334-022-00449-3.

[16] Daniel Neider, Rick Smetsers, Frits Vaandrager, and Harco Kuppens. *Benchmarks for Automata Learning and Conformance Testing*, pages 390–416. Springer International Publishing, Cham, 2019. ISBN 978-3-030-22348-9. doi: 10.1007/978-3-030-22348-9_23. URL https://doi.org/10.1007/978-3-030-22348-9_23.

[17] Oliver Niese. *An integrated approach to testing complex systems*. PhD thesis, University of Dortmund, 2003. URL https://doi.org/10.17877/DE290R-14871.

[18] C.P. Pfleeger. State reduction in incompletely specified finite-state machines. *IEEE Transactions on Computers*, C-22(12):1099–1102, December 1973. ISSN 0018-9340. doi: 10.1109/t-c.1973.223655. URL http://dx.doi.org/10.1109/T-C.1973.223655.

[19] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing - STOC '89*, STOC '89, page 411–420. ACM Press, 1989. doi: 10.1145/73007.73047. URL http://dx.doi.org/10.1145/73007.73047.

[20] Frits Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. *A New Approach for Active Automata Learning Based on Apartness*, page 223–243. Springer International Publishing, 2022. ISBN 9783030995249. doi: 10.1007/978-3-030-99524-9_12. URL http://dx.doi.org/10.1007/978-3-030-99524-9_12.

[21] Gerco van Heerdt, Clemens Kupke, Jurriaan Rot, and Alexandra Silva. Learning weighted automata over principal ideal domains, 2019. URL https://arxiv.org/abs/1911.04404.

Appendix A

Proofs

Proof of Lemma 2.12

Proof. Generalised claim: for all $p \in Q_{\mathcal{M}}, q \in Q_{\mathcal{N}}$:

$$p \sim q \iff \text{for all } \sigma \in \Sigma^*, \lambda^*_{\mathcal{M}}(p, \sigma) \text{ is consistent with } \lambda^*_{\mathcal{N}}(q, \sigma) \tag{5}$$

Lemma 2.12 follows by taking $p = q^0_{\mathcal{M}}, q = q^0_{\mathcal{N}}$, since $\mathcal{M} \sim \mathcal{N}$ is defined as $q^0_{\mathcal{M}} \sim q^0_{\mathcal{N}}$ and $\mathcal{M} \approx \mathcal{N}$ is defined as the right-hand side of equation (5) for $p = q^0_{\mathcal{M}}, q = q^0_{\mathcal{N}}$. Thus, proving the generalised claim also proves Lemma 2.12.

We will now prove the generalised claim:

Direction ($\Leftarrow$).

Define

$$R = \{(p', q') \in Q_{\mathcal{M}} \times Q_{\mathcal{N}} \mid \text{for all } \sigma \in \Sigma^*, \lambda^*_{\mathcal{M}}(p', \sigma) \text{ is consistent with } \lambda^*_{\mathcal{N}}(q', \sigma)\}$$

We show that R is a consistent bisimulation (Definition 2.9). Recall from Chapter 2 that $\lambda(q, a) = \square$ when $\lambda(q, a) \uparrow$. Let $(p', q') \in R$ and $a \in \Sigma$.

Condition 1. Take $\sigma = a$. By the second clause of the definition of λ^*, we have $\lambda^*_{\mathcal{M}}(p', a) = \lambda_{\mathcal{M}}(p', a)$ and $\lambda^*_{\mathcal{N}}(q', a) = \lambda_{\mathcal{N}}(q', a)$. Since $(p', q') \in R$, these are consistent, i.e. $\lambda_{\mathcal{M}}(p', a) = \square$, or $\lambda_{\mathcal{N}}(q', a) = \square$, or $\lambda_{\mathcal{M}}(p', a) = \lambda_{\mathcal{N}}(q', a)$. In particular, if $\lambda_{\mathcal{M}}(p', a) \downarrow$ and $\lambda_{\mathcal{N}}(q', a) \downarrow$ (so neither output is undefined), then $\lambda_{\mathcal{M}}(p', a) = \lambda_{\mathcal{N}}(q', a)$, which is exactly condition 1.

Condition 2. We must show $(\delta_{\mathcal{M}}(p', a), \delta_{\mathcal{N}}(q', a)) \in R$, i.e. for every $w \in \Sigma^*$, $\lambda^*_{\mathcal{M}}(\delta_{\mathcal{M}}(p', a), w)$ is consistent with $\lambda^*_{\mathcal{N}}(\delta_{\mathcal{N}}(q', a), w)$.

- If $w = \epsilon$: both sides equal ϵ (first clause of λ^*), and ϵ is consistent with ϵ.
- If $w \in \Sigma^+$: by the third clause of the definition of $\lambda^*, \lambda^*_{\mathcal{M}}(\delta_{\mathcal{M}}(p', a), w) = \lambda^*_{\mathcal{M}}(p', aw)$ and $\lambda^*_{\mathcal{N}}(\delta_{\mathcal{N}}(q', a), w) = \lambda^*_{\mathcal{N}}(q', aw)$. Since $(p', q') \in R, \lambda^*_{\mathcal{M}}(p', aw)$ is consistent with $\lambda^*_{\mathcal{N}}(q', aw)$, and thus so are $\lambda^*_{\mathcal{M}}(\delta_{\mathcal{M}}(p', a), w)$ and $\lambda^*_{\mathcal{N}}(\delta_{\mathcal{N}}(q', a), w)$.

So $(\delta_{\mathcal{M}}(p', a), \delta_{\mathcal{N}}(q', a)) \in R$, and R is a consistent bisimulation.

Now, if p and q satisfy the right-hand side of equation (5), then $(p, q) \in R$ by definition of R, and since R is a consistent bisimulation with $(p, q) \in R$, we get $p \sim q$.

Direction ($\Rightarrow$).

Suppose $p \sim q$, witnessed by a bisimulation R with $(p, q) \in R$. We prove, by induction on $n = |\sigma|$:

$$P(n)\text{: for all } (p', q') \in R \text{ and all } \sigma \in \Sigma^* \text{ with } |\sigma| = n, \lambda^*_{\mathcal{M}}(p', \sigma) \text{ is consistent with } \lambda^*_{\mathcal{N}}(q', \sigma)$$

Case $n = 0$. $\sigma = \epsilon$, and $\lambda^*_{\mathcal{M}}(p', \epsilon) = \epsilon = \lambda^*_{\mathcal{N}}(q', \epsilon)$, which is consistent.

Case $n = 1$. $\sigma = a$ for some $a \in \Sigma$. Then $\lambda^*_{\mathcal{M}}(p', a) = \lambda_{\mathcal{M}}(p', a)$ and $\lambda^*_{\mathcal{N}}(q', a) = \lambda_{\mathcal{N}}(q', a)$. Since $(p', q') \in R$ and R satisfies condition (1) of Definition 2.9: if both $\lambda_{\mathcal{M}}(p', a) \downarrow$ and $\lambda_{\mathcal{N}}(q', a) \downarrow$, then $\lambda_{\mathcal{M}}(p', a) = \lambda_{\mathcal{N}}(q', a)$, and thus consistent. Otherwise, at least one of them is $\square$, and thus consistent.

Case $n \geq 2$, assuming $P(n-1)$. $\sigma = aw$ with $a \in \Sigma, w \in \Sigma^+, |w| = n - 1$. By the third clause of the definition of λ^*,

$$\lambda^*_{\mathcal{M}}(p', aw) = \lambda^*_{\mathcal{M}}(\delta_{\mathcal{M}}(p', a), w) \qquad \lambda^*_{\mathcal{N}}(q', aw) = \lambda^*_{\mathcal{N}}(\delta_{\mathcal{N}}(q', a), w)$$

Since $(p', q') \in R$, condition (2) of Definition 2.9 gives $(\delta_{\mathcal{M}}(p', a), \delta_{\mathcal{N}}(q', a)) \in R$. By the induction hypothesis $P(n-1)$ applied to this pair and to w, $\lambda^*_{\mathcal{M}}(\delta_{\mathcal{M}}(p', a), w)$ is consistent with $\lambda^*_{\mathcal{N}}(\delta_{\mathcal{N}}(q', a), w)$. Thus, $\lambda^*_{\mathcal{M}}(p', aw)$ is consistent with $\lambda^*_{\mathcal{N}}(q', aw)$.

By induction, $P(n)$ holds for all $n \geq 0$. In particular, taking $(p', q') = (p, q) \in R$ gives consistency of $\lambda^*_{\mathcal{M}}(p, \sigma)$ with $\lambda^*_{\mathcal{N}}(q, \sigma)$ for all $\sigma \in \Sigma^*$.

Conclusion. Both directions of equation (5) hold for all p, q. Specialising to $p = q^0_{\mathcal{M}}, q = q^0_{\mathcal{N}}$:

$$\mathcal{M} \sim \mathcal{N} \iff \mathcal{M} \approx \mathcal{N}.$$

$\square$