BACHELOR'S THESIS COMPUTING SCIENCE

RADBOUD UNIVERSITY NIJMEGEN

Assumption-based optimizations for $L^{\#}$ using special outputs

---

Author:
Marijn Meuleman
s1063102

First supervisor/assessor:
Prof. Dr. Frits Vaandrager

Second assessor:
Dr. Jurriaan Rot

March 11, 2026

Abstract

Active automata learning algorithms are generally designed with a black-box philosophy. This results in the user of such an algorithm having very limited influence over the execution thereof. In reality, through familiarity or manual analysis, a user may already know or assume properties of the system under learning. To harness this knowledge for efficiency gains, we present changes to $L^{\#}$ that will give users of the algorithm more options to enter assumptions about the SUL. These changes are enabled by the use of an observation tree in $L^{\#}$. Users will pass SUL outputs that they know translate to specific behavior, and optimizations based on this behavior are applied. We focus on three realistic scenarios: transitions that do not change the state, transitions that all lead to the same state, and transitions that put the SUL in an unrecoverable state. A framework will be given to combine these optimizations and future optimizations with the same structure. Though these changes do not decrease asymptotic complexity, experiments using a Python implementation show significant performance improvements on both cherry-picked and realistic SULs.

Contents

1 **Introduction** **3**

2 **Preliminaries** **5**
2.1 Mealy machines . . . 5
2.2 Apartness . . . 7
2.3 Active automata learning and the $L^{\#}$ algorithm . . . 9
2.4 Notes on interacting with the SUL . . . 11

3 **Example model** **14**

4 **Research** **16**
4.1 Infinite trees . . . 16
4.2 Crash optimization . . . 17
4.2.1 Rationale . . . 17
4.2.2 Definition . . . 17
4.2.3 Example . . . 18
4.2.4 Correctness . . . 19
4.2.5 Theoretical improvements . . . 20
4.3 Retry optimization . . . 21
4.3.1 Rationale . . . 21
4.3.2 Definition . . . 22
4.3.3 Example . . . 23
4.3.4 Correctness . . . 25
4.3.5 Theoretical improvements . . . 26
4.4 Goto optimization . . . 27
4.4.1 Rationale . . . 27
4.4.2 Definition . . . 28
4.4.3 Example . . . 29
4.4.4 Correctness . . . 31
4.4.5 Theoretical improvements . . . 32
4.5 Combining the optimizations . . . 33
4.6 Implementing the optimizations . . . 35
4.6.1 Storage of infinite trees . . . 36
4.6.2 Apartness and ADS on infinite trees . . . 36
4.6.3 preprocessing apartness for states with special outputs . . . 37

5 Experimental evaluation **38**
5.1 Test models . . . 38
5.2 Setup and reproducibility . . . 39
5.3 Results . . . 39
5.3.1 Coffee Machine . . . 39
5.3.2 RSA BSAFE for C . . . 40
5.3.3 SMTP . . . 40
5.3.4 OpenSSH . . . 41
5.4 Interpretation and discussion of results . . . 41

6 Discussion **43**
6.1 Limitations and future work . . . 43
6.2 Conclusion . . . 44

A Proofs **48**

B AI Disclosure **56**

Chapter 1

Introduction

Active automata learning is a field that is focused on learning the behavior of black-box systems by interactively determining an abstraction of that program in the form of an automaton. Approaches for active automata learning generally build on a framework—the *learning game*—proposed originally by Dana Angluin [4]. In this learning game, the system under learning is considered a *minimally adequate teacher*; a teacher that can be interacted with using two types of queries. The first are output queries, in which the system is prompted to perform some operations (and thus change its state). The second are equivalence queries, in which the system is provided an attempt at an abstraction and the system verifies the correctness of that abstraction, providing a counterexample if it does not pass. The game is won when an abstraction is produced that passes the equivalence query, and is thus an accurate abstraction of the program.

Over time, many approaches have been developed that are dedicated to solving this problem as efficiently as possible. Notably, the classic L^* algorithm for regular languages/DFAs constructed by Angluin alongside the learning game [4], many transformations of L^* to work with other types of automata [2, 3, 7, 16, 15, 19, 24, 25, 32, 33, 37, 38, 41], and some popular evolutions of L^* like the table-based Rivest-Shapire (RS) algorithm [29] or the tree-based Kearns-Vazirani (KV) [20], ADT [12] and TTT [18] algorithms. Only very few learning algorithms were developed that were fundamentally distinct from L^*, and most of these were focused on more expressive or complex types of automata. Examples are non-deterministic automata [28], quantitative automata [17] and register automata [8]. The $L^{\#}$ algorithm [36], which this paper builds on, instead takes a novel approach to active automata learning for Mealy machines. It is based on apartness and using an observation tree, opposed to the traditional use of observation equivalence stored in a table or distinguishing tree. Because of this new strategy, we can cleanly formalize the optimizations in this paper using properties of that observation tree.

The vast majority of new versions or improvements in active automata learning still share the minimalist black-box assumption on the system under learning that L^* was defined with. This assumption limits the used knowledge about the underlying system in the algorithm to merely the type of automaton of its abstraction, and occasionally some already known observations. As a consequence, the user of a learning algorithm has little influence over the learning process. They are largely limited to configuring the *oracle* that simulates equivalence queries, or providing a set of known observations which the algorithm will take as a seed. This approach is sensible; changing the algorithm to reflect specific assumptions on the SUL takes effort, is difficult to get right, and—perhaps most importantly—is not *needed*. After all, the algorithm correctly learns the SUL abstraction without interference by the user as well. However, that is not to say that there are no advantages to giving users more options to introduce knowledge or educated guesses about the workings of the system. By default, the learning algorithm perceives

the SUL as a complete black box, and treats its learning accordingly. In this state, learning takes as much effort as it possibly could for the used algorithm. The introduction of rules that the SUL is expected to follow must then reduce (or, at worst, keep the same) the effort that it takes to learn the SUL.

To partially remove the conflict between wanting low specialization effort but high efficiency, this paper is about giving users increased control over the learning process of the $L^{\#}$ automata learning algorithm without them manually needing to change the underlying algorithm. Instead, users will be able to represent behaviors by using specific output symbols to map SUL outputs to in the *mapper layer/adapter*, the translator between the inputs and outputs of the SUL and those of the model created via the learning algorithm. Such an approach is already widely used in model-based testing. We will explore optimizations for three realistic assumptions on the SUL. These optimizations will be defined formally and proven to be correct. The number of queries they can save will also be computed. At the end, we will combine the optimizations to work in tandem, so they form one big improvement together. The optimizations are implemented in AALpy [26], tested, and compared to the original $L^{\#}$ algorithm. We also believe they may be of use as templates for other related improvements, as all three definitions have some similarities.

Related work

The area of involving background knowledge of the SUL in active automata learning is relatively underdeveloped compared to the rest of the field. Fica and Otop [10] developed a framework for including assumptions about equivalences between input sequences for DFAs (with an included extension to Mealy machines). They achieved by allowing users to pass a *string rewriting system* consistent with SUL behavior, that is used to normalize any input sequence before it is sent to the SUL for evaluation. While just input preprocessing suffices for DFAs and is also useful for Mealy machines, the more expressive nature of Mealy machines allows us to take a more reactive approach based on output symbols. Berthon et al. [5] introduces an upper bound of the domain of possible output transducers, to reduce the search space for a correct output and get smaller outputs in general. Garhewal et al. [13] approaches its optimization similarly, by bounding domains of input and output parameters. In Wallner et al. [39], it was demonstrated that a timed SUL could be learned through an untimed skeleton of the–cheaply obtained–hidden Mealy machine using genetic programming. Tangentially related, there are also various papers looking at compositional automata; automata representing a big system made up of various smaller subsystems [21, 23, 27]. The knowledge of the internal modules of a system is also a form of background knowledge. Lastly, a more active subtopic of this field, *adaptive* learning, has quite some research behind it [6, 9, 14, 22, 40]. Adaptive active automata learning concerns itself with learning new models from old models that are very similar. It is often used in the context of software updates, because programs tend to stay largely the same after an update. Another important application is learning various different implementations of a protocol.

Chapter 2

Preliminaries

In this chapter, we provide the foundational knowledge from past research that is needed to understand the research we have done. The information discussed here revolves primarily around the $L^{\#}$ algorithm, presented in Vaandrager et al. [36] which this research expands upon and whose concepts are extensively used in the research. In the first two sections, concepts that are needed for understanding $L^{\#}$ are explained. In the third section, these concepts will be combined to explain the $L^{\#}$ algorithm itself. The final section contains relevant information about the practical structure and inner workings of active automata learning algorithms.

2.1 Mealy machines

Real systems (known as **SUL**s, systems under learning) are complex, and it is rarely the case that every bit-flip, every line of code, or even every part of functionality is important to learn. Therefore, in automata learning, we learn abstractions based on the SUL to discover something about the system itself. As the name suggests, the abstraction we choose is a type of automaton. Many types of automata can be used for abstractions of a SUL, and automata learning algorithms like $L^{\#}$—the algorithm that this paper is based on—have been implemented for various types as well; Mealy machines, Moore machines and DFAs, in the case of $L^{\#}$. In this paper, we opt for the variant defined on Mealy machines. This type of automaton is the one that the original $L^{\#}$ used as well.

Mealy machines differ from DFAs because of their output functionality; rather than just having a state change on an input, a Mealy machine will also yield an output belonging to that transition. Those outputs being transition-bound rather than state-bound is what separates Mealy machines from Moore machines. The existence of outputs, as well as them being connected to the transitions, is essential for the optimizations in this paper. We make use of Mealy machines both for the hidden representation of the SUL and the maintained data structure during the learning process. The former is inherently a *complete* machine while the latter is a special type of *partial* machine called an **observation tree**; see Definition 2.4 (*Tree*).

We first fix the definition of a partial map, which allows us to properly define Mealy machines as well as reason using the notion of definedness. Intuitively, this is simply a function that may not be defined on the entire input domain.

Definition 2.1 (Partial maps)

A **partial map** is a function that is only defined on a subset of the input domain. The following notions and notations are associated with partial maps:

- $f : X \rightharpoonup Y$: f is a partial map from set X to set Y;
- $f(x)\downarrow$ for $x \in X$: f is defined on x, $\exists_{y \in Y} f(x) = y$;
- $f(x)\uparrow$ for $x \in X$: f is not defined on x, $\neg\exists_{y \in Y} f(x) = y$.

Mealy machines are a kind of automaton which produces an output alongside a resulting state for transitions. This makes them well-suited for an abstract representation of the functionality of a computer program; one could model the main functionality of a program as transitions between states, and any side effects as the output of those transitions.

Definition 2.2 (Mealy machines)

A **Mealy machine** is a tuple $\mathcal{M} = (Q, q_0, I, O, \delta, \lambda)$, where:

- Q is a finite set of **states** and $q_0 \in Q$ is the **initial state**;
- $\delta : Q \times I \rightharpoonup Q$ the **transition function** which maps state-input pairs to resulting states;
- $\lambda : Q \times I \rightharpoonup O$ the **output function** which maps state-input pairs to resulting outputs;
- for $(q, i) \in Q \times I$, $\delta(q, i)\downarrow \iff \lambda(q, i)\downarrow$.

A Mealy machine is called **complete** if $\delta(q, i)\downarrow$ and $\lambda(q, i)\downarrow$ for all $(q, i) \in Q \times I$. As a consequence, $\delta : Q \times I \to Q$ and $\lambda : Q \times I \to O$. We trivially extend δ and λ to input strings of any length $n \in \mathbb{N}$ as $\delta : Q \times I^n \rightharpoonup Q$ the resulting state, $\lambda : Q \times I^n \rightharpoonup O^n$ all outputs along the path.

Comparing the behavior of two complete Mealy machines is simple; if the outputs in the one machine match the outputs in the other for every input sequence, they are equivalent. However, we will also work with partially defined Mealy machines. We therefore also define a weaker comparator to identify whether one machine *includes the behavior* of another—that is, whether every transition defined in the former machine is also present in the latter, possibly with additional transitions. Intuitively, a simulation means that one machine can "act as" another by mapping its states appropriately.

Definition 2.3 (Functional simulation)

For Mealy machines $\mathcal{M}$ and $\mathcal{N}$ with $I^{\mathcal{M}} = I^{\mathcal{N}}$ and $O^{\mathcal{M}} = O^{\mathcal{N}}$, a **functional simulation** $f : \mathcal{M} \to \mathcal{N}$ is a map $f : Q^{\mathcal{M}} \to Q^{\mathcal{N}}$ such that:

$$f(q_0^{\mathcal{M}}) = q_0^{\mathcal{N}} \quad \text{and} \quad \forall_{q,q' \in Q^{\mathcal{M}}} \forall_{\mathrm{i} \in I;\, \mathrm{o} \in O} [q \xrightarrow{\mathrm{i/o}} q' \quad \implies \quad f(q) \xrightarrow{\mathrm{i/o}} f(q')]$$

thus a functional simulation is any function from one machine to another that preserves transitions. We say $\mathcal{M}$ **simulates** $\mathcal{N}$ ($\mathcal{M} \preccurlyeq \mathcal{N}$) if such a functional simulation exists. Moreover, we say $\mathcal{M}$ **is equivalent to** $\mathcal{N}$ ($\mathcal{M} \approx \mathcal{N}$) if $\mathcal{M} \preccurlyeq \mathcal{N}$ *and* f is surjective ($f(Q^{\mathcal{M}}) = Q^{\mathcal{N}}$), or vice versa.

A particular kind of partial Mealy machine that we use in this research is one where every

state (except the root) has a unique predecessor. We call a machine like this a tree. One way to get a tree is by querying another Mealy machine and simply keeping track of the information by creating new states for every new observation.

Definition 2.4 (Tree)

A Mealy machine $\mathcal{T}$ is called a **tree** if, for all combinations $\sigma, \sigma' \in I^*$ such that $\delta(q_0, \sigma)\downarrow$ and $\delta(q_0, \sigma')\downarrow$:

$$\delta(q_0, \sigma) = \delta(q_0, \sigma') \implies \sigma = \sigma'$$

i.e. every state is only reachable via one path from the initial state. We also define $\texttt{pred} : Q \backslash \{q_0\} \to Q$ as the direct predecessor of a state in $\mathcal{T}$; only one such predecessor can exist. Moreover, $\mathcal{T}$ is called an **observation tree** of a machine $\mathcal{M}$ if $\mathcal{T} \preccurlyeq \mathcal{M}$.

2.2 Apartness

We know that knowledge about SUL behavior translates directly to knowledge about its observation tree. After all, an observation tree simulates, by definition, the hidden Mealy machine representing the program. One central question in automata learning is how to make the inverse connection. How to find the relation between observation tree states and SUL states, specifically *without* knowing anything about the internal states of the SUL? We need this knowledge to reconstruct the hidden Mealy machine representing the system from the observation tree.

To this end, the concept of apartness is used. Two states being apart means they behave differently for some input sequence. Since we cannot test what states they go to in the source machine, we establish this apartness from the outputs that their transitions give. Based on this definition, two states that are apart must represent different internal states in the SUL, as we assume the machine representing the SUL is deterministic. This concept is worked out in Theorem 2.6 (*Correspondence apartness source and tree*).

Beyond helping us reconstruct the hidden Mealy machine from its observation tree, apartness also informs our querying strategies. A query that we know will not create relevant, new apartness pairs is generally worse than one that will, as there is no guarantee it will get us closer to that goal of reconstruction of the SUL machine. The details of the approach $L^\#$ takes, and the influence of apartness therein, are explained in Section 2.3 (*Active automata learning and the $L^\#$ algorithm*).

Definition 2.5 (Apartness)

For a Mealy machine $\mathcal{M}$, we say that states $p, q \in Q$ are **apart** ($p \# q$) if there exists some $\sigma \in I^*$ such that $\lambda(p, \sigma)\downarrow$, $\lambda(q, \sigma)\downarrow$ and $\lambda(p, \sigma) \neq \lambda(q, \sigma)$. We then call such a σ a **witness** for the apartness of p and q ($\sigma \vdash p \# q$).

▶ Troelstra and Schwichtenberg [34]

A consequence of defining apartness based on the existence of a witness is that *apartness* implies *inequality*, but the converse is not true per se. The concept of the observation tree is a testament to this. However, even among only complete Mealy machines this may not be the case. One might consider a machine with two distinct states behaving the exact same way; same outputs, same destination states. These states will never be apart, but they are not equal. Apartness is therefore a stronger claim than inequality. We also use this concept in the be-

low theorem. Here, we prove that apartness of states in the observation tree corresponds to apartness of their origin states in the source Mealy machine. This is the crucial connection that enables us to reconstruct the source system from its observations.

Theorem 2.6 (Correspondence apartness source and tree)

For $\mathcal{M}$ a Mealy machine, $\mathcal{T}$ an observation tree with $p, q \in Q^{\mathcal{T}}$ and $f : \mathcal{T} \to \mathcal{M}$ its simulation:

$$p \#^{\mathcal{T}} q \quad \implies \quad f(p) \#^{\mathcal{M}} f(q)$$

Appendix A (*Proof of Theorem 2.6*) □

Now that the road from the observation tree back to the hidden SUL Mealy machine has been paved, we should establish a way to know where we are, so that we can deduce how to best proceed. We create a set of states that we know to represent different states in the hidden Mealy machine. These states, their transitions (incoming and outgoing) and the states that their children represent are all important for forming a guess on what the SUL machine looks like. This set of states can and must take on a tree shape for the algorithm to work.

Definition 2.7 (Basis and frontier)

For $\mathcal{T}$ an observation tree, we call $S \subseteq Q$ a **basis** for $\mathcal{T}$ if:

(1) the states in S are pairwise apart, i.e.

$$\forall_{p,q \in S | p \neq q}[p \# q]$$

(2) the states in S form a subtree of $\mathcal{T}$ with root q_0, i.e.

$$q_0 \in S \land \forall_{b \in S \setminus \{q_0\}}[\texttt{pred}(b) \in S]$$

Moreover, the **frontier** F is the set of states in $\mathcal{T}$ that are direct successors of states in B.

▶ Vaandrager et al. [36], introduction chapter 3

To prevent having to recompute a basis after every expansion of the observation tree, we maintain the basis throughout the algorithm, expanding it with new states whenever possible. Integral to this approach is that basis states remain valid throughout the execution of the algorithm, even when new observations are added to the tree.

Theorem 2.8 (Basis valid after expansion)

Let $\mathcal{S}$ and $\mathcal{T}$ be observation trees of a complete Mealy machine $\mathcal{M}$ such that $\mathcal{S}$ is a subtree of $\mathcal{T}$ with the same root. Then, if some S is a basis for $\mathcal{S}$, S is a basis for $\mathcal{T}$.

Appendix A (*Proof of Theorem 2.8*) □

The basis that is now maintained serves as a proxy for states in the hidden Mealy machine. We compare the non-basis states to the basis states to determine which SUL state they represent. The definition used for this is defined here.

Definition 2.9 (Candidate set)

For $\mathcal{T}$ an observation tree with basis B and $q \in Q$, we define $C(q) = \{b \in B \mid \neg(q \# b)\}$ the **candidate set** of q. We say q is **identified** if $|C(q)| = 1$, and **isolated** if $|C(q)| = 0$. A set of states is identified if all its elements are identified.

▶ Vaandrager et al. [36], Def. 3.5

We now have all the tools to try to reconstruct the source Mealy machine from its observation tree. Such an attempted reconstruction is called a **hypothesis**. Not all hypotheses are created equal; it is, for example, possible to generate a hypothesis without making a single observation. However, there is also no guarantee that a hypothesis will be equivalent to the SUL, even if it is thoroughly thought through. Testing the correctness of the hypothesis is generally costly, so we want to make the most of the observations we already have before trying to do so. The following criterion notes the properties we want a hypothesis should typically meet before it is considered worth comparing to the SUL.

Proposition 2.10 (Sensible hypothesis test criterion)

A hypothesis should generally only be tested against the SUL if it is generated once each basis state is complete and all frontier states are identified, and there are no inconsistencies between it and the observation tree.

2.3 Active automata learning and the $L^\#$ algorithm

The field of *active automata learning* is concerned with finding an optimal strategy for the learner in the following game:

Definition 2.11 (Active automata learning game)

In the active automata learning game, the *teacher* has a complete Mealy machine $\mathcal{M}$ and answers the following queries from the *learner*:

> `OutputQuery`(σ): for $\sigma \in I^*$, the teacher replies with the output sequence $\lambda^{\mathcal{M}}(q_0^{\mathcal{M}}, \sigma)$.
>
> `EquivQuery`$(\mathcal{H})$: for a complete Mealy machine $\mathcal{H}$, the teacher replies $\top$ if $\mathcal{H} \approx \mathcal{M}$, or $\sigma \in I^*$ such that $\lambda^{\mathcal{M}}(q_0^{\mathcal{M}}, \sigma) \neq \lambda^{\mathcal{H}}(q_0^{\mathcal{H}}, \sigma)$ otherwise.

The learner wins when it finds some $\mathcal{H}$ such that `EquivQuery`$(\mathcal{H})$ returns $\top$.

▶ Derived from Angluin [4]

"Optimal" can refer to multiple things here; the time taken by the learner to win, the amount of (a certain type of) queries, or the underlying number of steps and resets of the SUL that happened. In real scenarios, communications with and internal computations of the SUL are commonly significantly slower than the learning algorithm—even more so because we can choose the hardware to run the learning algorithm on. Hence, the numbers of SUL manipulations are often considered good measures of efficiency of a learning algorithm. This paper will also formally reason about its optimizations using steps and resets of the SUL as metrics. How these metrics are derived is described in Section 2.4 (*Notes on interacting with the SUL*).

The $L^\#$ algorithm makes use of the concepts discussed in Section 2.1 (*Mealy machines*), Section 2.2 (*Apartness*) to play this learning game and learn the behavior of a SUL. In brief, the

$L^{\#}$ algorithm can be described as follows.

Definition 2.12 ($L^{\#}$)

Given a hidden Mealy machine $\mathcal{M}$ with input alphabet I and output alphabet O, and an observation tree $\mathcal{T}$ with basis S with frontier F. The $L^{\#}$ algorithm repeatedly applies the following rules in arbitrary order.

(R1) **Promotion:** if some $f \in F$ is isolated, f is added to S and F is updated appropriately.

(R2) **Extension:** if the frontier state for $i \in I$ of $q \in S$ is undiscovered, send $\texttt{OutputQuery}(\text{access}(q)\, i)$ to the SUL and a new state is added to F with the resulting output.

(R3) **Identification:** if some $f \in F$ is not yet identified, take $c_1, c_2 \in C(f)$ with $\sigma \vdash c_1 \# c_2$, send $\texttt{OutputQuery}(\text{access}(f)\, \sigma)$ and expand $\mathcal{T}$ from f accordingly.

(R4) **Equivalence:** construct a hypothesis from S and F and verify its consistency with $\mathcal{T}$. If no contradiction is found in $\mathcal{T}$, pose $\texttt{EquivQuery}(\mathcal{H})$. Then, if the verdict of the query is affirmative, output the hypothesis. If instead a counterexample was found in either check, update $\mathcal{T}$ in a way that resolves the counterexample[a] respectively from $\mathcal{T}$ or yielded by the teacher.

A variant of $L^{\#}$ called **strategic** $L^{\#}$ only performs (R4) if the preconditions of none of the other rules are met. Going forward, when we mention $L^{\#}$, it refers to *strategic* $L^{\#}$.

► Vaandrager et al. [36], section 3.2 and Def. 3.13

[a] See Vaandrager et al. [36], section 3.4 for an explanation about counterexample processing in $L^{\#}$.

Note that strategic $L^{\#}$ satisfies the requirements outlined in Proposition 2.10 (*Sensible hypothesis test criterion*); mostly directly, and indirectly for the case of an isolated state in the tree in (R4) itself (an isolated state implies an inconsistency). Equivalence queries are very costly compared to regular output queries, so waiting as long as possible to use them helps reduce the time and the internal operations of the SUL.

A generic optimization for $L^{\#}$, meant to reduce both input symbols and total output queries for $L^{\#}$, is the use of cleverly generated suffixes for the output queries in (R2) and (R3). These suffixes can be used to gather additional information about a node without needing to reset and access that node all over again. The approach taken in the $L^{\#}$ paper, called **adaptive distinguishing sequences** (ADS), is forming a decision tree based on (a part of) the basis of a tree. The decision tree would decide the next input based on previous inputs and their outputs. The decision tree is designed to generate inputs that are optimal for gaining apartness pairs. The formal theory behind ADS is out of scope for this paper, but a definition of a version of $L^{\#}$ using ADS is defined below.

Definition 2.13 ($L^{\#}_{\text{ADS}}$)

Consider $L^{\#}$ with the output queries changed to:

(R2) `OutputQuery`$(\text{access}(q)\ i\ \text{ADS}(S))$;

(R3) `OutputQuery`$(\text{access}(f)\ \sigma\ \text{ADS}(\{b \in S \mid \neg(b \# q)\}))$.

We call this optimized $L^{\#}$ variant $L^{\#}_{\text{ADS}}$. A strategic variant of $L^{\#}_{\text{ADS}}$ exists analogously to strategic $L^{\#}$. Going forward, when we mention $L^{\#}_{\text{ADS}}$, it refers to *strategic* $L^{\#}_{\text{ADS}}$.

▶ Vaandrager et al. [36], section 3.5, particularly Prop. 3.12

The $L^{\#}$ and $L^{\#}_{\text{ADS}}$ algorithms both have the same equivalence query, output query and symbol complexities, which we will give here, as they are relevant in measuring the advantage of the optimizations in the rest of this paper.

Theorem 2.14 ($L^{\#}$ complexity)

Consider a complete hidden Mealy machine $\mathcal{M}$, with $n = |Q|$, $k = |I|$ and m the size of the longest counterexample. Then $L^{\#}$ and $L^{\#}_{\text{ADS}}$ learn $\mathcal{M}$ within:

- at most $n - 1$ equivalence queries;
- $\mathcal{O}(kn^2 + n \log_2 m)$ reset queries;
- $\mathcal{O}(kmn^2 + mn \log_2 m)$ symbol queries (assuming $n \in \mathcal{O}(m)$).

▶ Vaandrager et al. [36], Thm. 3.15

2.4 Notes on interacting with the SUL

In the past sections, we have spoken of hidden Mealy machines with certain outputs, that we can learn by performing `OutputQueries` and `EquivalenceQueries` on them. However, such abstractions are neither part of the program itself, nor known in advance by the learner. We essentially assume that such a Mealy machine based on the program behavior exists, and infer it step by step by mapping Mealy inputs to real system inputs and Mealy outputs to real system outputs. This mapping happens in an appropriately named **mapper (layer)**, which stands between the learning algorithm and the actual SUL. An overview of the architecture for active automata learning, which shows the position of the mapper, is given in Figure 2.1.

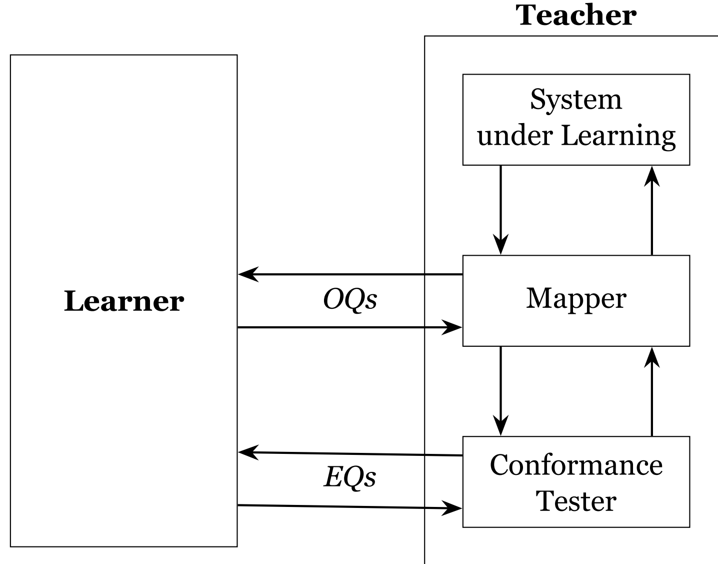


Figure 2.1: High level architecture of the active automata learning game [35]

Through limiting the inputs to only those useful for our learning goals, or grouping certain types of outputs together into one output symbol, we can adjust the learning scope to fit our needs. It is good to keep in mind that these manipulations are possible, even necessary to some degree, while evaluating the significance of the improvements suggested in this paper.

Much like model inputs and outputs generally not existing in that same form in the SUL, `OutputQueries` and `EquivalenceQueries` are also not actually part of the majority of programs; they are *simulated* queries. An equivalence query can be viewed as a very large set of consecutive output queries, generated by a *conformance tester* (or *equivalence oracle*), meant to determine with high certainty whether a hypothesis correctly represents the SUL. Further technical explanation is out of scope for this paper. However, the simulation of output queries is relevant to our optimizations, so we define it in pseudocode below.

Algorithm 1 An `OutputQuery` in $L^{\#}$

```
procedure OutputQuery(σ ∈ I*, 𝒯 the observation tree, S the SUL)
    if σ = ϵ then
        S.reset()                                   ▷ reset represents a SUL restart
        set current state to root in 𝒯
    end if

    o ← S.step(σ[0])              ▷ step represents a SUL input, like a button press
    add (σ[0], o) as observation from current state in 𝒯
    set current state to newly added state in 𝒯

    OutputQuery(σ[1 . . . ], 𝒯, S)
end procedure
```

As can be seen here, an output query is not just one operation provided to the SUL. Instead, every symbol in that output query is evaluated by the program separately, and its output is processed after every provided input symbol. When “reset queries” or “symbol queries” are

used in the paper, they refer to the `reset` and `step` operations that can be seen in the algorithm. We use these in particular when evaluating the advantage the optimizations have over regular $L^{\#}$. Moreover, some parts of optimizations may be based on extending or prematurely ending an output query based on some intermediate output; it should be trivial from this pseudocode that such changes can exist within the existing algorithm through some minor changes.

Chapter 3

Example model

To easily demonstrate the effect of the optimizations throughout the rest of this paper, we describe a simple example of a coffee machine, copied with small changes from Steffen et al. [31], a popular sample automaton in the literature [1]. The happy flow for the machine works as follows:

1. the user inserts a pod or water into the machine;
2. the user inserts a pod or water, whichever is not yet in the machine;
3. the user presses a button, and coffee comes out;
4. the user cleans the machine.

The user can clean the machine at any time. Adding water or adding a pod when they are already present will cause an alert for the user to tell them they are doing something wrong, but they will be allowed to proceed. Pressing the button early or doing something else than cleaning after getting the coffee, however, will cause the machine to malfunction and stop working altogether. Consider the following domains for a Mealy machine abstraction of the coffee machine:

- $Q = \{$*clean*, *pod*, *water*, *ready*, *done*, *fail*$\}$ respectively the cleaned, only pod added, only water added, ready to make coffee, coffee made, and malfunction states;
- $q_0 =$ *clean*;
- $I = \{pod, water, button, clean\}$ respectively the insert pod, add water, press button and clean machine input symbols;
- $O = \{\checkmark, !, \lightning, \text{☕}, \ast\}$ respectively the success, warning, crash, coffee and cleaned output symbols.

An abstraction for these domains is presented in Figure 3.1. Note that the *retry* state is drawn twice; this is for the purpose of readability.

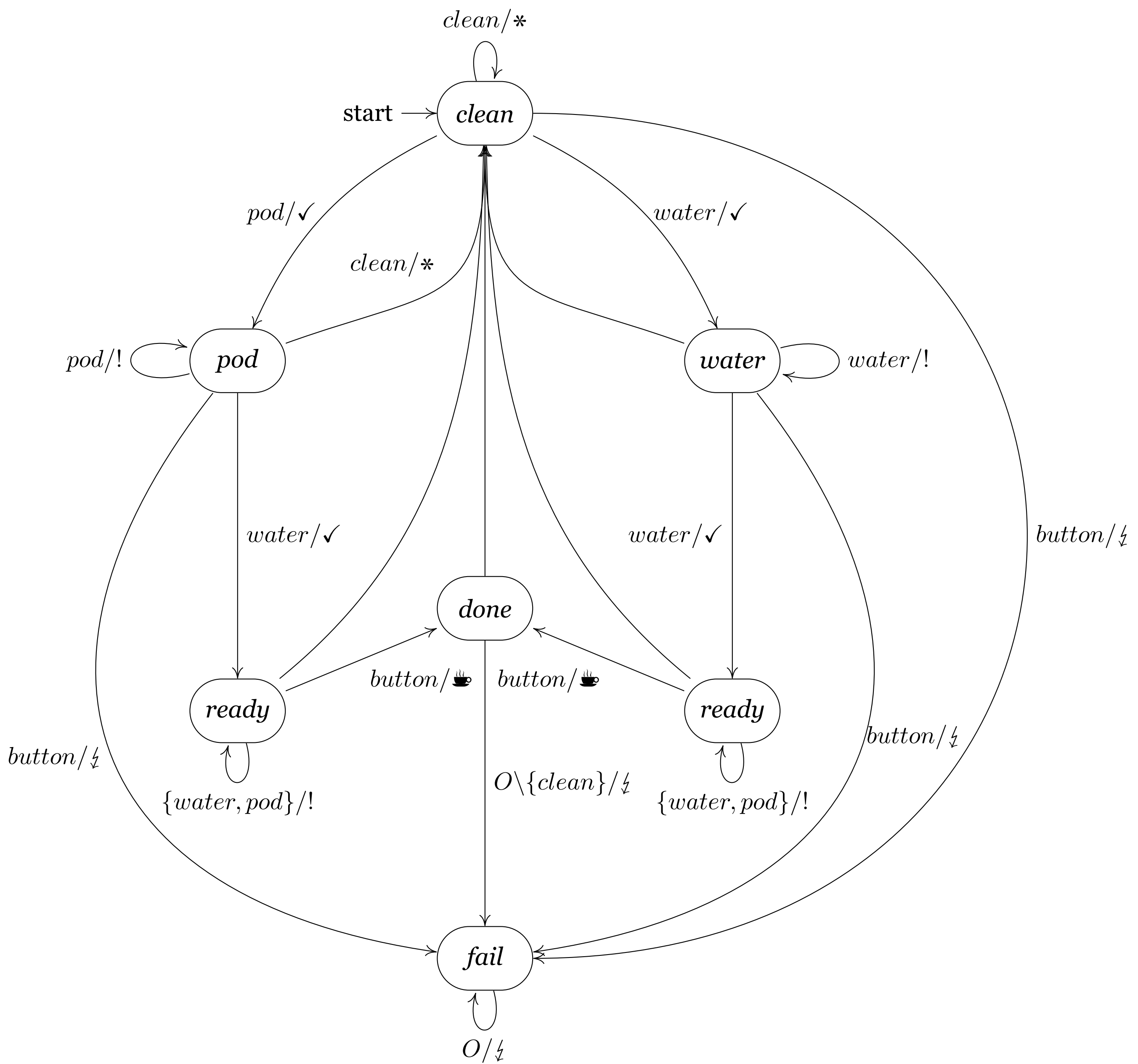


Figure 3.1: Example coffee machine model

Chapter 4

Research

This chapter describes the optimizations of $L^{\#}$ proposed in this thesis. In every section we will name the output that is used in the optimization, give a short description of the type of SUL outputs that should map to that output, define the optimization, use it on an example and prove its correctness and complexity. In Chapter 5 (*Experimental evaluation*), the implementation of the improvements will actually be tested and compared against the original $L^{\#}$ algorithm.

It should be noted that the optimizations may have other applications than those indicated in the description; the description should serve as a *guideline*, not a hard rule. For example, if some SUL output marks entering a sub-process that does not produce side effects and is not useful to your testing, it could be desirable to map this to a crash output (explained in Section 4.2 (*Crash optimization*)) to cut off exploration of that sub-process.

4.1 Infinite trees

Throughout Chapter 2 (*Preliminaries*), we have only reasoned about finite hidden Mealy machines, finite hypotheses, and finite observation trees. After all, $L^{\#}$ works with SULs with finitely many internal states, and the observation tree only contains the finite number of observations made. However, our optimizations will work based on inferring SUL behavior–and thus, gaining observations without actually observing them. This can lead to an observation tree with infinitely long branches. We extend the definition of Mealy machines to allow for these infinite trees.

Definition 4.1 (Mealy machines (updated))

Extend the domain of Mealy machines to cover Definition 2.2 (*Mealy machines*) *without* the "finite" restriction on Q. The newly added infinite Mealy machines will be treated as regular Mealy machines. However, hidden Mealy machines representing SULs in active automata learning are always required to be finite.

Fortunately, the properties, relations, and operations discussed in Chapter 2 (*Preliminaries*) trivially extend to these new infinite Mealy machines as well. However, just because there is no issue in theory with, for example, checking equivalence between two infinite Mealy machines, does not mean that it is feasible in practice. Fortunately, in the context of this thesis, the presence of infinite subtrees in the optimizations is limited to certain scenarios. We are able to argue that, in those scenarios, we can perform all necessary operations for infinite trees by just considering a finite subtree. The implementation details are expanded upon in Section 4.6 (*Implementing the optimizations*).

4.2 Crash optimization

The first optimization we will work out is one pertaining to *sink states*; states that "inescapable" and therefore do not need further exploration. In particular, we consider scenarios of fatal errors, where all inputs accessing and within the sink state yield the same error output. We call the output that indicates such behavior in the destination state of its transition the *crash output* ϕ. Such an optimization is already present in some implementations of automata learning algorithms[1], but there is no paper formalizing the optimization as far as we could find, and it has not been implemented in AALpy before.

4.2.1 Rationale

Treating states that act as sink states for the purposes of learning separately from other states trivially causes a reduction in learning effort. By definition, all transitions with ϕ land in a single sink state, and on every subsequent input they will stay in that state. We therefore already know the behavior of any transition that comes after a ϕ transition, so we do not need to evaluate a query for this anymore. In practice, this simply means that once a crash output is discovered during an output query, the learning program stops executing the rest of the query. Additionally, we do not need to explore or identify states representing the sink state, as we know what they are already, and any queries done after reaching one will just yield ϕ as output. This might seem like a minor improvement, but some larger models have tens of inputs of which only a small fraction are defined over a given specification state. We will also rely on the maximum number of defined inputs across states of the hidden Mealy machine to give an estimate of the performance improvements of this optimization over regular $L^{\#}$.

4.2.2 Definition

We first define a restriction on the use of ϕ, which serves as a precondition for using the optimized algorithm. This amounts to the assumptions that are introduced when we use ϕ; these assumptions enable the algorithm to work more effectively in all cases where ϕ is used, but using ϕ in a way that does not adhere to these requirements may cause undesirable behavior.

Definition 4.2 (Crash-conformity)

Consider $\mathcal{M}$ a (complete) Mealy machine representing a program. For some output ϕ, if there exists a $q_{crash} \in Q$ such that:

1. $\forall_{i \in I}[\lambda(q_{crash}, i) = \phi]$;
2. $\forall_{(q,i) \in Q \times I}[\lambda(q, i) = \phi \implies \delta(q, i) = q_{crash}]$

we call $\mathcal{M}$ **crash-conform through** ϕ, or $\langle \mathcal{M}; \phi \rangle$ **crash-conform**.

We then define the improvement on some crash-conform Mealy machine. As explained in Section 4.2.1 (*Rationale*), the main idea of the optimization is to cut output queries off early and identify less frontier states. Both are achieved in this optimization: the subtrees of ϕ-states are (recursively) made complete to make identifications unnecessary, and queries are cut off early through [C4].

[1] An example of an AAL library that allows for sink states is LearnLib; see `https://learnlib.de/learnlib/maven-site/0.9.1/apidocs/de/learnlib/cache/mealy/MealyCacheOracle.html`

Definition 4.3 ($L^\#_{crash}$)

Consider $L^\#$ performed on $\langle \mathcal{M}; \phi \rangle$ crash-conform with the following changes:

[C1] **(Crash invariant)**: any $q \in \mathcal{T}$ that has ϕ as the output for its unique access transition is maintained to be complete, with all outgoing transitions having crash outputs;

[C4] during an `OutputQuery` for σ, whenever ϕ is encountered, the tree is updated according to [C1]. If the query is now redundant because σ is fully contained in $\mathcal{T}$, execution of the query is halted and the SUL is reset. Otherwise, the query continues where it left off.

We denote this $L^\#$ variant, which provides optimizations for crash-conform Mealy machines, $L^\#_{crash}$.

This change can be made, because the behavior of the subtree after a ϕ is already fully known, represented by the invariant. Note that on every occurrence of ϕ, we get an infinite subtree as discussed in Section 4.1 (*Infinite trees*). While we will discuss the treatment of infinite subtrees in detail in Section 4.6 (*Implementing the optimizations*), one might already realize the lack of added distinguishing information that the children of states reached by crash outputs give. We will use this property to cut off exploration early, mainly in the context of apartness and ADS computations.

4.2.3 Example

In Figure 3.1 (*Example coffee machine model*), the ↯-output looks much like ϕ in terms of behavior. We know that all transitions with ↯-outputs go to a single state (*fail*), and outgoing outputs from *fail* have a ↯-output. We can therefore conclude that the coffee machine model is indeed crash-conform through ↯. Consider the following observation tree for the coffee machine, where double outlined states are basis states, and with the actual states included as subscript for convenience:

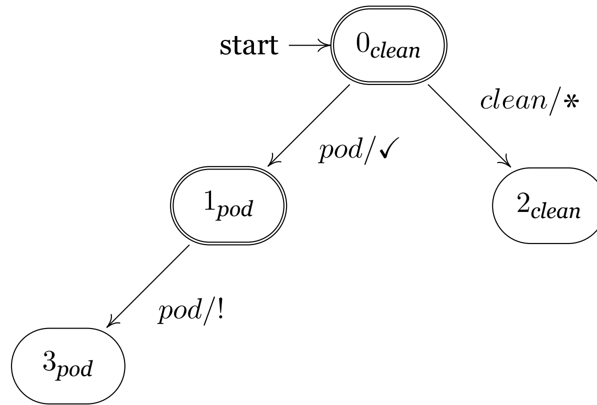
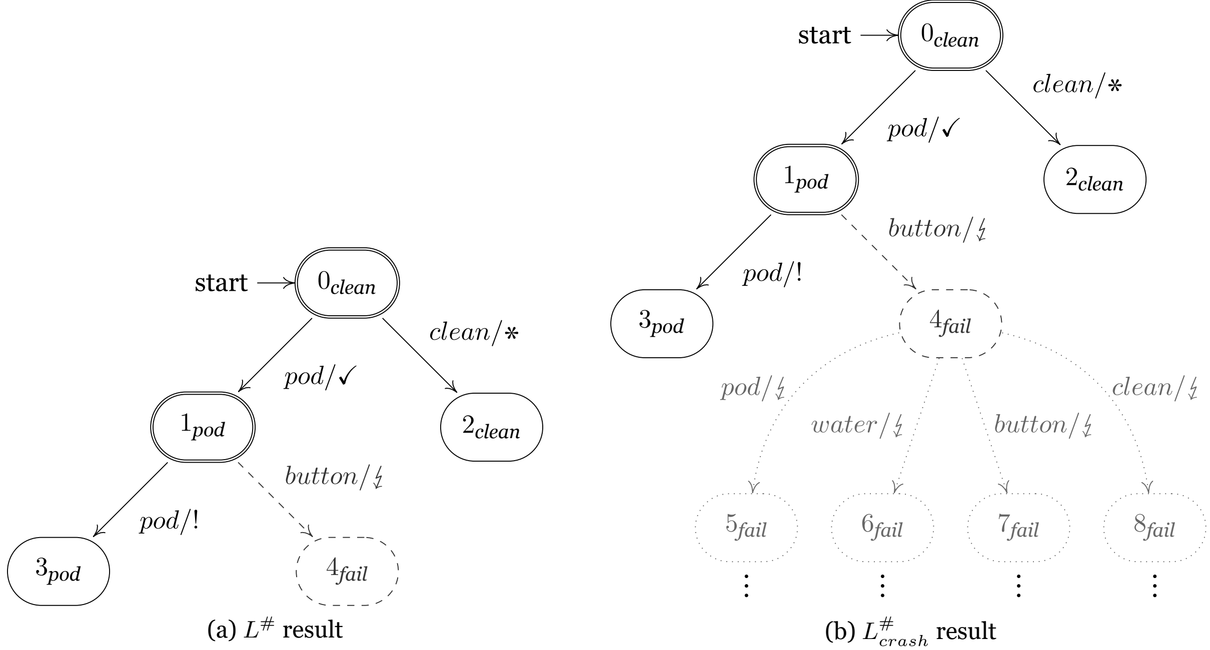


Figure 4.1: Example observation tree for the coffee machine

From this point, we consider a scenario in which both $L^\#$ and $L^\#_{crash}$ want to apply [R2] to discover *button* in state 1. This is a possibility, because that state was not discovered yet. We send `OutputQuery`(*pod button*) to the SUL (via the mapper). This query is evaluated; first, ✓is returned (which we already knew), and then ↯ is observed. The tree is kept up to date during

the evaluation, so it now contains the new $button/\lightning$ edge. For $L^{\#}$, this is all that happens.

However, for $L^{\#}_{crash}$, this observation tree does not satisfy the invariant. The new state reached by the $button/\lightning$ transition is updated with new transitions for all inputs, all having $\lightning$ outputs. As a consequence, the invariant now also triggers for all these descendants. This recursively leads to an infinite subtree with the new state as its root. Consequently, the resulting trees look as follows (newly added information dashed, inferred information from the invariant dotted):



(a) $L^{\#}$ result

(b) $L^{\#}_{crash}$ result

Figure 4.2: Comparison $L^{\#}$ and $L^{\#}_{crash}$ trees after rule application

This one query has provided $L^{\#}_{crash}$ with a lot more information that $L^{\#}$ with the same query. This additional information has real implications; for example, state 4 is known for $L^{\#}_{crash}$ to be a basis state, but the same cannot be said for $L^{\#}$. Moreover, after its promotion via [R1], its children (which are then frontier state) will already be identified, saving on [R3] queries.

4.2.4 Correctness

Generally, we would want to show that the invariant is valid with respect to crash-conform Mealy machines. The other change effectively follows from that invariant, so the invariant proof validates the complete optimization. In the other optimizations this approach is taken, but this particular invariant follows directly from Definition 4.2 (*Crash-conformity*). We have therefore decided to not supply an explicit proof.

Proposition 4.4 (Crash invariant soundness)

Consider $\langle \mathcal{M}; \phi \rangle$ crash-conform with $\mathcal{T}$ an observation tree for $\mathcal{M}$. Let $\mathcal{T}'$ be $\mathcal{T}$, expanded so that it conforms to the crash invariant. Then $\mathcal{T}'$ is an observation tree for $\mathcal{M}$.

We then show that $L^{\#}_{crash}$ can correctly learn a program. In essence, showing this amounts to showing that every query in $L^{\#}_{crash}$ will find the same information as that query in regular

$L^{\#}$, and probably some more. Since we know the extra observations due to the changes correctly correspond to behavior of the Mealy machine (from Proposition 4.4 (*Crash invariant soundness*)), any additional discoveries are simply nice to have. This reduction from $L^{\#}_{crash}$ operations to $L^{\#}$ operations allows us to implicitly use the $L^{\#}$ correctness proof for $L^{\#}_{crash}$ also.

Theorem 4.5 ($L^{\#}_{crash}$ correctness)

$L^{\#}_{crash}$ learns a crash-conform $\langle \mathcal{M}; \phi \rangle$ for $\mathcal{M}$ a complete Mealy machine.

Appendix A (*Proof of Theorem 4.5*) □

This theorem show that $L^{\#}_{crash}$ can be used wherever $L^{\#}$ is currently used, with the only difference being that the user needs to provide a crash output to use the optimization with. In the case where the optimization is not wanted, a user can add an output which is designed to never be used by the SUL, and have that output be ϕ. If the optimization is wanted, a user can define a proper crash output, and it will be used by the algorithm.

4.2.5 Theoretical improvements

To determine the effect of the optimization on the number of symbol and reset queries, we compute the reduction of those for two scenarios of the learning progression. The first is a general case, where the distribution of crash outputs, as well as the shape of the basis subtree, is unknown. The second is a realistic best case—*best* in terms of the improvements we can make—in which crash outputs are distributed uniformly, and the learning process yields an inefficient basis subtree.

Theorem 4.6 ($L^{\#}_{crash}$ improvements – general case)

Consider a complete hidden Mealy machine $\mathcal{M}$ that is crash-conform through ϕ, with $n = |Q|$ and $k = |I|$, and where all states in Q are reachable from q_0. *Consider* t_{crash} *the total number of transitions with a crash output, and assume* $t_{crash} > 0$. Then $L^{\#}_{crash}$ saves at least the following number of queries compared to $L^{\#}$ learning $\mathcal{M}$:

- $k + t_{crash} + 1$ reset queries;
- $\max(3t_{crash} + k, 2t_{crash} + 2k)$ symbol queries.

Appendix A (*Proof of Theorem 4.6*) □

These numbers unfortunately do not make a proper dent in the reset queries and symbol queries that are used by $L^{\#}$ (stated in Theorem 2.14 ($L^{\#}$ *complexity*)). Even when running the values with t_{crash} taking on its theoretical upper bound of kn, we would be off by a factor $\mathcal{O}(n)$ for reset queries and even more ($\mathcal{O}(mn)$) for symbol queries. Having that said, the numbers are minima, and they were computed for deliberately unfortunate distributions of the crash outputs. We will evaluate whether making some helpful assumptions makes a difference.

Theorem 4.7 ($L^{\#}_{crash}$ improvements – best case)

Consider a complete hidden Mealy machine $\mathcal{M}$ that is retry-conform through id, with $n = |Q|$ and $k = |I|$, and where all states in Q are reachable from q_0. *Consider* $\#_{crash}(q)$ *the number of transitions with retry outputs out of* q*, and let* $k_{retry} = \min_q(\#_{crash}(q))$. *Assume that each basis node has at most one basis child, so that* S *forms a long branch.* Then $L^{\#}_{crash}$ saves at least the following number of queries compared to $L^{\#}$ learning $\mathcal{M}$:

- $k + nk_{crash} + 1$ reset queries;
- $kn + \frac{1}{2}(n-1)nk_{crash}$ symbol queries.

Appendix A (*Proof of Theorem 4.7*) □

The reset queries unfortunately did not get reduced further; the distribution of crash outputs and basis states does not change the queries used. The symbol query reductions show promise, however. Doing some simplifications, we can find n^2k_{crash} in the second term of the addition. This is only $\mathcal{O}(m)$ removed from its counterpart in Theorem 2.14 ($L^{\#}$ *complexity*) when setting k_{crash} to its upper bound of k. While m, the length of the longest counterexample, does not influence our improvement necessarily, it is likely that longer counterexamples imply longer distinguishing sequences, which would mean better performance of the optimization also.

We believe that, for a relatively uniformly distributed set of transitions with crash outputs, one should generally expect savings close to those in the best case, with negative outliers potentially only reaching improvements noted in the general case. If the hidden Mealy machine is very unbalanced in terms of the distribution of crash outputs from the get-go, results close to those in the general case should be more common instead. In Chapter 5 (*Experimental evaluation*), we will evaluate the actual impact of this optimization and the others.

4.3 Retry optimization

The second optimization builds on the background of the first. Not all invalid inputs lead to an unrecoverable state that would warrant the use of ϕ. One might imagine an invalid command could simply be ignored. In a case like this, the system will effectively prompt the user to retry entering a command, without there being a change in state. However, faulty inputs are not the only cases in which such self-looping behavior can be observed. The scope of a learning process can be configured by the user of a learning algorithm in the mapping layer. There can be situation where some input performs a proper state change for two states, but the difference from the source state is only relevant to the process for one of the two cases. This improvement has the user map the SUL output of such irrelevant transitions to a **retry output** id.

4.3.1 Rationale

The approach for using the knowledge of id to improve performance is very similar to that of ϕ. After all, in the situations in which the retry optimization described here can be used, the crash optimization described in Section 4.2 (*Crash optimization*) can be used instead. This stems from the fact that no information is found by expanding the resulting state of a self-loop that would not otherwise be found by expanding the source state itself. This notion of there being no use exploring a state further is what the crash optimization is based on. However, there is merit to not just treating retry transitions like crashes. Whereas the latter are designed to be

used for transitions that lead you to a sink state, transitions with retry outputs still lead to a state that is part of the normal program flow. Expansion of the tree can then simply continue from there—an option not present for ϕ transitions. In fact, any exploration after such a self-loop can be treated as exploration on its source, saving a reset and the symbols to get back for further discoveries. We define changes to the algorithm to achieve this formally below.

4.3.2 Definition

We first define a restriction on the use of id, akin to Definition 4.2 (*Crash-conformity*) for the crash optimization.

Definition 4.8 (Retry-conformity)

Consider $\mathcal{M}$ a (complete) Mealy machine representing a program. For $\mathsf{id} \in O$, if:

$$\forall_{(q,i)\in Q\times I}[\lambda(q,i) = \mathsf{id} \implies \delta(q,i) = q]$$

we call $\mathcal{M}$ **retry-conform through** id, or $\langle \mathcal{M}; \mathsf{id} \rangle$ **retry-conform**.

We then define the improvement on some retry-conform Mealy machine. This optimization consists of three components. The first is the use of the retry invariant. This is a property we maintain on the observation tree to maximally exploit our meta-knowledge of the hidden machine, as it is an inference based on retry-conformity taken to its extreme. A proof of the soundness of the invariant is given in Theorem 4.11 (*Retry invariant soundness*). The second is a pre- and post-processing of output queries. Through Definition 4.8 (*Retry-conformity*), we know that the results after a transition with id are the same as they would have been without that transition. By removing those transitions from the input, we reduce the number of queried symbols. We then simply reconstruct the correct output based on the result of the output query. The last is the clever extension of queries if we are in a position to do so. This takes advantage of the invariant to get additional information from a basis state if we end a query in a looped state.

Definition 4.9 ($L^{\#}_{retry}$)

Consider $L^{\#}$ performed on $\langle\mathcal{M}; \mathsf{id}\rangle$ retry-conform with the following changes:

[C1] **(Retry invariant)**: for any q, q' such that $q \xrightarrow{i/\mathsf{id}} q'$ in $\mathcal{T}$ for some $i \in I$, the subtree with root q is maintained to be the same as the subtree with root q' (and vice versa);

[C2] before the evaluation of an OutputQuery for σ, get the output sequence ω for the longest prefix of σ contained in $\mathcal{T}$, and remove input characters from σ corresponding to the retry outputs in ω;

[C3] before the ResetQuery of an OutputQuery that ended in a frontier state, if the evaluation ended in $q \xrightarrow{i/\mathsf{id}} q'$, the input sequence will be extended by one character that is not yet explored in q (if possible). This finds a new frontier state for q, equivalently to [R2] but without the reset (this may happen repeatedly). This also works for a state known to represent a frontier state, as is the case after e.g. an application of [C3] in Definition 4.16 ($L^{\#}_{goto}$);

[C4] during an `OutputQuery` for σ, whenever id is encountered, the tree is updated according to [C1]. If the query is now redundant because σ is fully contained in $\mathcal{T}$, execution of the query is halted and the SUL is reset. Otherwise, the query continues where it left off.

We call this $L^{\#}$ variant, which provides optimizations for retry-conform machines, $L^{\#}_{retry}$.

Interesting here is the distinction between $L^{\#}$ and $L^{\#}_{\text{ADS}}$; the variant with Ads continues after an application of [R2] or [R3] to get even more information before a reset. This change means we can still use the optimization to great effect, while maintaining the benefit that $L^{\#}_{\text{ADS}}$ gives compared to the original algorithm. One might also notice the retry invariant leads to infinite length observation trees again. However, the implementation can simply maintain just the parent tree, and the looped children can refer to observations made in the parent. The specifics are explained in Section 4.6 (*Implementing the optimizations*).

4.3.3 Example

Much like the $\lightning$-output represented ϕ, we find in Figure 3.1 (*Example coffee machine model*) that ! represents id. We find that every occurrence of ! is in a self-loop, verifying that the coffee machine model is indeed retry-conform through !. Consider once again the following observation tree for the coffee machine, where double outlined states are basis states, and with the actual states included as subscript for convenience:

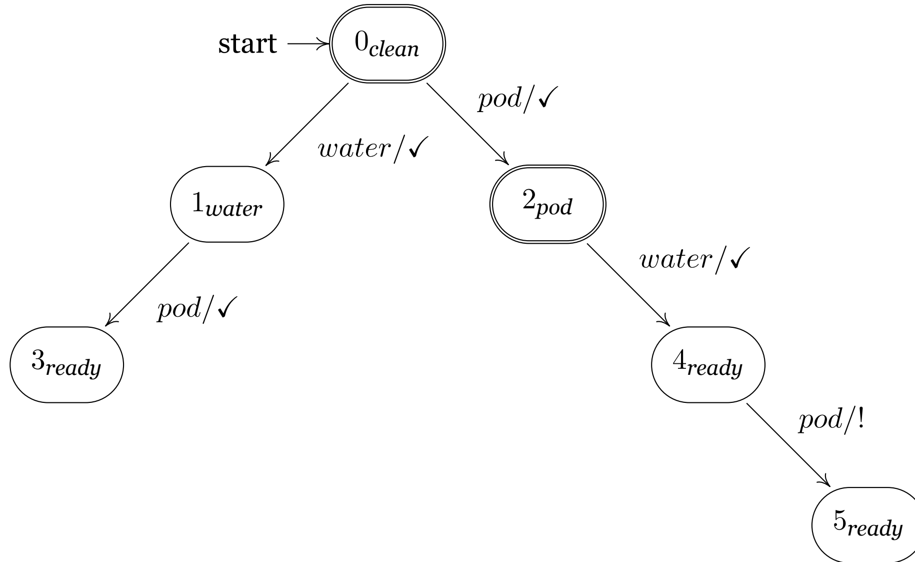
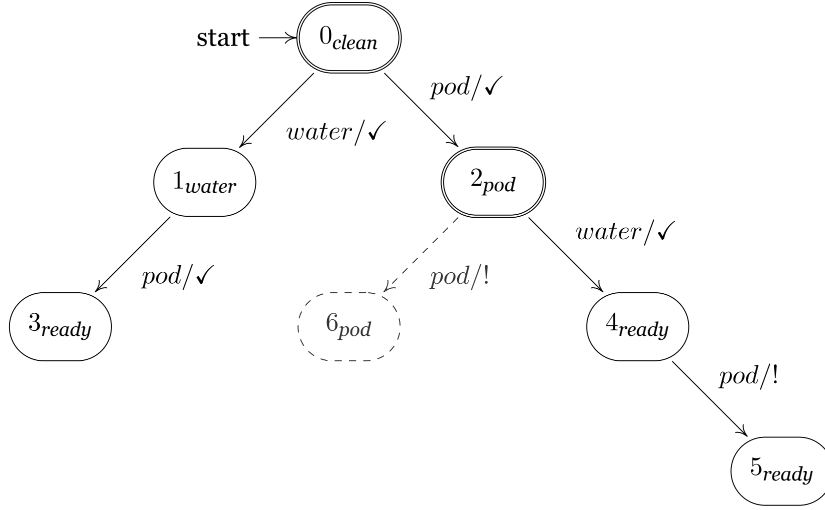


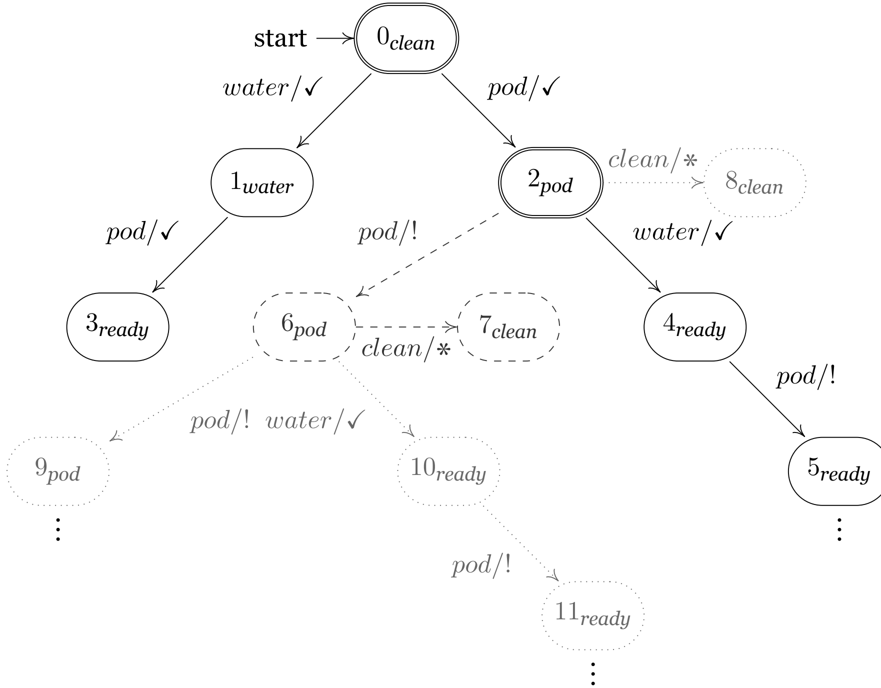
Figure 4.3: Example observation tree for the coffee machine

Before we start the simulation of a query, we can already notice something going wrong for $L^{\#}_{retry}$: state 5 does not have a $pod/!$ outgoing transition, violating the retry invariant. For $L^{\#}_{retry}$, we may simply imagine it is there (and so there is an infinite subtree). From this point, we consider a scenario in which both $L^{\#}$ and $L^{\#}_{retry}$ want to apply [R2] for pod on 2. This is a possibility, because that state was not discovered yet. We send `OutputQuery`($pod\ pod$) to the SUL (via the mapper). This query is evaluated; first, ✓is returned (which we already knew), and then ! is observed. The tree is kept up to date during the evaluation, so it now contains the new $pod/!$ edge. For $L^{\#}$, this is all that happens.

However, for $L^{\#}_{retry}$, this observation tree does not satisfy the invariant. The new state reached by the $pod/!$ transition is updated with a subtree that matches that of its parent, state 2. This means it gains both the $pod/!$ transition (yielding an infinite subtree), and the branch from 2 starting with $water/✓$ (also yielding an infinite subtree via 5). After the tree is updated, [C3] takes effect, as its conditions were met. Another query from the new child of 2 is placed to discover $clean$. The output is ✓. It gets added to the new child of 2, but also to 2 itself and all the children of the new child reached by pod. These are all connected with !-outputs, and thus by the invariant their subtrees should match exactly. Consequently, the resulting trees look as follows (newly added information dashed, inferred information from the invariant dotted):



(a) $L^{\#}$ result



(b) $L^{\#}_{retry}$ result

Figure 4.4: Comparison $L^{\#}$ and $L^{\#}_{retry}$ trees after rule application

A lot of additional information got introduced by using $L^{\#}_{retry}$ instead of $L^{\#}_{crash}$ for this query. The most obvious direct benefit is the one given by [C3], which provided a free (nearly; just one symbol) discovery of a second frontier state in just one rule application. Furthermore, 6 is now also identified, meaning we forego at least one [R3] query.

4.3.4 Correctness

We start by showing that the retry invariant that we maintain is actually sound. To do this, we construct a lemma that establishes equality between program states if their access sequence only differs by some transitions with retry outputs.

Lemma 4.10

Consider $\langle \mathcal{M}; \mathsf{id} \rangle$ retry-conform. Let $\sigma, \sigma' \in I^*$ be access sequences for states $q, q' \in Q$. If $\lambda(q_0, \sigma)$ with retry outputs removed is equal to $\lambda(q_0, \sigma')$ with retry outputs removed, then $q = q'$.

Appendix A (*Proof of Lemma 4.10*) □

We use this proof to establish that a valid observation tree remains valid after the retry invariant is applied.

Theorem 4.11 (Retry invariant soundness)

Consider $\langle \mathcal{M}; \mathsf{id} \rangle$ retry-conform with $\mathcal{T}$ an observation tree for $\mathcal{M}$. Let $\mathcal{T}'$ be $\mathcal{T}$, expanded so that it conforms to the retry invariant. Then $\mathcal{T}'$ is an observation tree for $\mathcal{M}$.

Appendix A (*Proof of Theorem 4.11*) □

Now that the optimization is shown to be sound, we will prove that it is correct as well. We show that $L^{\#}_{retry}$ can correctly learn a complete hidden Mealy machine representing a program. The proof does not depend on whether id is actually used in the machine, only that it is defined (as per Definition 4.8 (*Retry-conformity*)). The form of the argument is analogous to that for Theorem 4.5 ($L^{\#}_{crash}$ *correctness*).

Theorem 4.12 ($L^{\#}_{retry}$ correctness)

$L^{\#}_{retry}$ learns retry-conform $\langle \mathcal{M}; \mathsf{id} \rangle$ with $\mathcal{M}$ a complete Mealy machine.

Appendix A (*Proof of Theorem 4.12*) □

To reiterate, the correctness proof has not only argued correctness, but has also shown that every evaluation of $L^{\#}$ on $\mathcal{M}$ has a counterpart evaluation of $L^{\#}_{retry}$ on $\langle \mathcal{M}; \mathsf{id} \rangle$ in which at least the same observations are obtained.

4.3.5 Theoretical improvements

We again show the minimum theoretical improvements both for a general number of retry outputs and based on a minimum number per state. We start with the generic case, in which we make no assumptions about the distribution of retry outputs.

Theorem 4.13 ($L^{\#}_{retry}$ improvements – general case)

Consider a complete hidden Mealy machine $\mathcal{M}$ that is retry-conform through id, with $n = |Q|$ and $k = |I|$, and where all states in Q are reachable from q_0. *Consider* t_{retry} *the total number of transitions with a retry output.* Assume the extension in [C3] is always possible. Then $L^{\#}_{retry}$ saves at least the following number of queries compared to $L^{\#}$ learning $\mathcal{M}$:

- $2t_{retry}$ reset queries;
- $\max(2t_{retry}, 4t_{retry} - 2k)$ symbol queries.

Appendix A (*Proof of Theorem 4.13*) □

We then consider a more realistic variant in which we assume an even distribution of retry outputs throughout the hidden Mealy machine, and moreover a learning progression that led to a basis tree beneficial to the optimization.

Theorem 4.14 ($L^{\#}_{retry}$ improvements – best case)

Consider a complete hidden Mealy machine $\mathcal{M}$ that is retry-conform through id, with $n = |Q|$ and $k = |I|$, and where all states in Q are reachable from q_0. *Consider* $\#_{retry}(q)$ *the number of transitions with retry outputs out of* q*, and let* $k_{retry} = \min_q(\#_{retry}(q))$*. Assume that each basis node has at most one basis child, so that* S *forms a long branch.* Assume the extension in [C3] is always possible. Then $L^{\#}_{retry}$ saves at least the following number of queries compared to $L^{\#}$ learning $\mathcal{M}$:

- $2nk_{retry}$ reset queries;
- $n(n+1)k_{retry}$ symbol queries.

Appendix A (*Proof of Theorem 4.14*) □

The results we find (for both theorems) are very similar to those of Section 4.2.5 (*Theoretical improvements*), at least with respect to their most significant terms. Our analysis therefore closely mimics that of the crash optimization performance improvements. Most differences can be explained through the advantage of using query extensions as defined by [C3]. This allows us to basically save a reset query, as well as many symbol queries, per id discovered. In general then, using retry outputs is better than using crash outputs; this is in line with the reasoning in Section 4.2.1 (*Rationale*), and it is backed by numbers here. As with the crash optimization, we will find out the actual performance benefit in the Chapter 5 (*Experimental evaluation*) section of this paper.

4.4 Goto optimization

We lastly consider the most versatile of the optimizations. This optimization models the behavior of going to a specific state, independent of what state the system is currently in. A very simple real implementation of this can be found in the menu bar of a website, where links to other pages are generally not dependent on the page one is currently visiting. To also tie this optimization back to the types of error behaviors in Section 4.2 (*Crash optimization*), this optimization would also cover (partial) resets and some other state changes as a result of errors; for example, an error screen is usually not dependent on the page it came from. This improvement has the user map the SUL outputs of transitions that are known to go to the same SUL state to a **goto output** ▷.

Unlike for the other optimizations, it is reasonable to allow for multiple goto outputs. After all, each separate SUL state that a user wants to map to requires a different symbol. In the definition of this optimization, we only use ▷; applying the optimization for multiple symbols is effectively the same as applying the optimization for each of those symbols separately.

4.4.1 Rationale

The approach for using the knowledge of ▷ to improve performance takes many things from the playbook of id. id and ▷ have in common that their subtrees are inferred based on observations

from other subtrees; for id it is the subtree of the parent, and for $\triangleright$ that of the other states reached by $\triangleright$. Where they differ is in their treatment of the special output at the end of an output query in [R2]. For id, this meant we were still in the same state, and so we could do an extra symbol query to discover another frontier state. With $\triangleright$, this is probably not the case. In fact, we do not even know whether a state reached by a goto output will become a basis; the basis state might also be another state that represents that same SUL state.

4.4.2 Definition

We first define a restriction on the use of $\triangleright$, akin to Definition 4.2 (*Crash-conformity*) and Definition 4.8 (*Retry-conformity*).

Definition 4.15 (Goto-conformity)

Consider $\mathcal{M}$ a (complete) Mealy machine representing a program. For $\triangleright \in O$, if there exists a state $t \in Q$ such that:

$$\forall_{(q,i)\in Q\times I}[\lambda(q,i) = \triangleright \quad \Longrightarrow \quad \delta(q,i) = t]$$

we call $\mathcal{M}$ **goto-conform through** $\triangleright$, or $\langle \mathcal{M}; \triangleright \rangle$ **goto-conform**.

The improvement for goto-conform machines has some overlap with $L^{\#}_{retry}$ (Definition 4.9 ($L^{\#}_{retry}$)). In particular, the invariant with the synchronized subtrees is present here, except rather than with the parent node the synchronization runs across all states that are accessed with id. In that sense, it is rather similar to $L^{\#}_{crash}$ (Definition 4.3 ($L^{\#}_{crash}$)) also; the subtree is the same for all sink states, though actually sharing the subtrees was not fitting for that optimization. The extension rule for [R2] also exists, with an additional requirement to prevent unnecessary queries. Different from both optimizations is the total replacement of prefixes of input sequence, contrasting with the removal of single symbols in $L^{\#}_{retry}$ and the premature ending of queries in $L^{\#}_{crash}$.

Definition 4.16 ($L^{\#}_{goto}$)

Consider $L^{\#}$ performed on $\langle \mathcal{M}; \triangleright \rangle$ goto-conform with the following changes:

[C1] **(Goto invariant)**: for any $q, q' \in \mathcal{T}$ with $\triangleright$ as the output for their unique access transition, the subtree with root q is maintained to be the same as the subtree with root q' (and vice versa);

[C2] let γ be the shortest input sequence that ends in $\triangleright$. Before the evaluation of an OUTPUTQUERY for σ, consider $\sigma = \sigma' \rho$ with σ' the longest prefix such that $\lambda(q_0, \sigma') = \ldots \triangleright$. Instead of σ, query for $\gamma\ \rho$;

[C3] if a state b that has $\triangleright$ as unique access transition output is in the basis, before the RESETQUERY of any OUTPUTQUERY, if the evaluation ended in a goto output, the input sequence will be extended by one character that is not yet explored in b (if possible). This finds a new frontier state for b, equivalently to [R2] but without the reset (this may happen repeatedly);

[C4] during an `OutputQuery` for σ, whenever ϕ is encountered, the tree is updated according to [C1]. If the query is now redundant because σ is fully contained in $\mathcal{T}$, execution of the query is halted and the SUL is reset. Otherwise, the query continues where it left off.

We call this $L^{\#}$ variant, which provides optimizations for retry-conform machines, $L^{\#}_{goto}$.

Though it is less apparent than in the other optimizations, we may find infinite length observation trees here as well, through $\triangleright$ somewhere in the subtree of a state reached by $\triangleright$. The implementation approach will be distinct from $L^{\#}_{retry}$, because simply referring to the parent will not work in this case. We will need a form of global variable to store where the node is that has the actual observations. This is the primary difference between this optimization and the other approaches; $L^{\#}_{crash}$ requires no context about the rest of the tree, $L^{\#}_{retry}$ only needs to know its parent and so its context is bounded. $L^{\#}_{goto}$ is the only one which will need global context, because it needs to be able to query a node completely elsewhere in the tree.

4.4.3 Example

We see in Figure 3.1 (*Example coffee machine model*) that $*$ represents $\triangleright$. We find that every occurrence of $*$ is in a transition leading to the same state, verifying that the coffee machine model is indeed goto-conform through $*$. Consider once again the following observation tree for the coffee machine, where double outlined states are basis states, and with the actual states included as subscript for convenience:

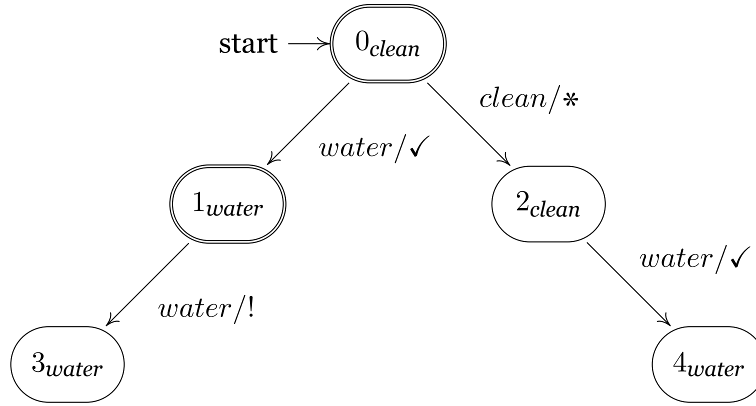
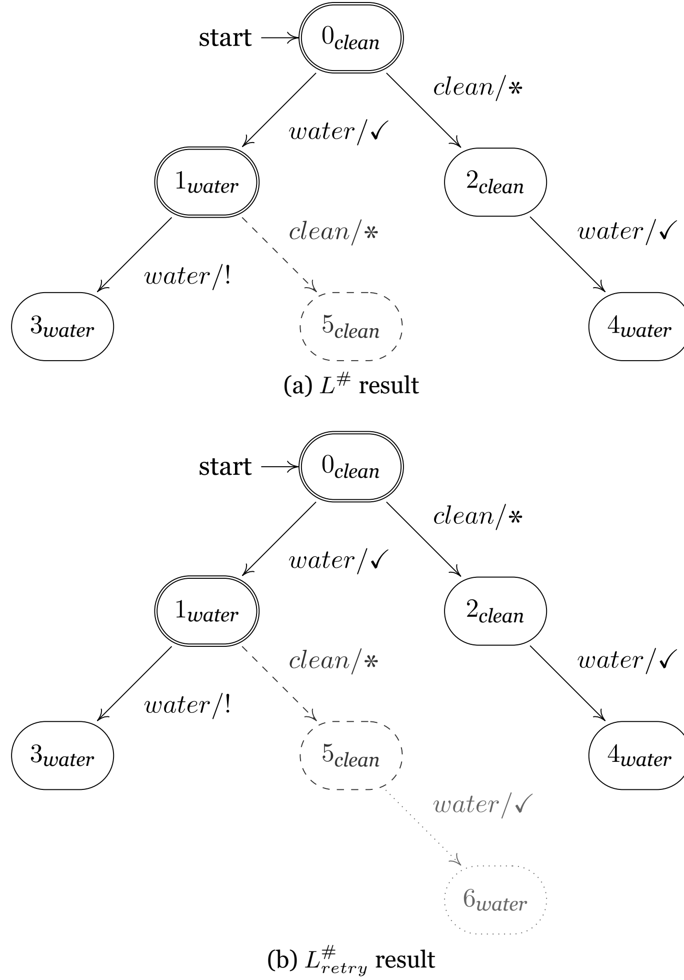


Figure 4.5: Example observation tree for the coffee machine

We consider a scenario in which both $L^{\#}$ and $L^{\#}_{retry}$ want to apply [R2] for $clean$ on 1. This is a possibility, because that state was not discovered yet. We send `OutputQuery`$(water\, clean)$ to the SUL (via the mapper). This query is evaluated; first, ✓ is returned (which we already knew), and then $*$ is observed. The tree is kept up to date during the evaluation, so it now contains the new $clean/*$ edge. For $L^{\#}$, this is all that happens.

However, for $L^{\#}_{retry}$, this observation tree does not satisfy the invariant. The new state reached by the $clean/*$ transition has the transitions from other states reached with a $*$ output added to its subtree. This means it also gains the $water/\checkmark$ observation from 2. Consequently, the resulting trees look as follows (newly added information dashed, inferred information from the invariant dotted):



(a) $L^{\#}$ result

(b) $L^{\#}_{retry}$ result

Figure 4.6: Comparison $L^{\#}$ and $L^{\#}_{retry}$ trees after rule application

Some additional information got introduced by using $L^{\#}_{goto}$ instead of $L^{\#}_{crash}$ for this query. As a direct effect, 5 is now identified, meaning we forego at least one [R3] query that we would have needed otherwise. Sadly, because of the structure of Figure 3.1 (*Example coffee machine model*), it was not possible to show the effect of [C3] and not feasible to demonstrate infinite subtree behavior in a realistic scenario.

4.4.4 Correctness

We first show an implication of goto outputs on the source Mealy machine itself.

Lemma 4.17

Consider $\langle \mathcal{M}; \triangleright \rangle$ goto-conform. Let $\sigma, \sigma' \in I^*$ be access sequences for states $q, q' \in Q$. If σ and σ' have a shared suffix that starts with a transition with a goto output, then $q = q'$.

Appendix A (*Proof of Lemma 4.17*) □

We use this lemma to establish that a valid observation tree remains valid after the goto invariant is applied.

Theorem 4.18 (Goto invariant soundness)

Consider $\langle \mathcal{M}; \triangleright \rangle$ goto-conform with $\mathcal{T}$ an observation tree for $\mathcal{M}$. Let $\mathcal{T}'$ be $\mathcal{T}$, expanded so that it conforms to the goto invariant. Then $\mathcal{T}'$ is an observation tree for $\mathcal{M}$.

Appendix A (*Proof of Theorem 4.18*) □

We lastly prove that $L^{\#}_{goto}$ is able to learn a Mealy machine. The form is analogous to that for Theorem 4.5 ($L^{\#}_{crash}$ *correctness*) and Theorem 4.12 ($L^{\#}_{retry}$ *correctness*).

Theorem 4.19 ($L^{\#}_{goto}$ correctness)

$L^{\#}_{goto}$ learns goto-conform $\langle \mathcal{M}; \triangleright \rangle$ with $\mathcal{M}$ a complete Mealy machine.

Appendix A (*Proof of Theorem 4.19*) □

4.4.5 Theoretical improvements

We again show the minimum theoretical improvements both for a general number of retry outputs and based on a minimum number per state, given a beneficial basis subtree. We start with the generic case, in which we make no assumptions about the distribution of retry outputs.

Theorem 4.20 ($L^{\#}_{goto}$ improvements – general case)

Consider a complete hidden Mealy machine $\mathcal{M}$ that is goto-conform through $\triangleright$, with $n = |Q|$ and $k = |I|$, and where all states in Q are reachable from q_0. *Consider* t_{goto} *the total number of transitions with a goto output.* Assume the extension in [C3] is always possible. Then $L^{\#}_{goto}$ saves at least the following number of queries compared to $L^{\#}$ learning $\mathcal{M}$:

- $\max(t_{goto}, k) + t_{goto} - 1$ reset queries;
- $\max(2(t_{goto} - 1), 3(t_{goto} - 1) - k)$ symbol queries.

Appendix A (*Proof of Theorem 4.20*) □

We then consider a more realistic variant in which we assume an even distribution of goto outputs throughout the hidden Mealy machine.

Theorem 4.21 ($L^{\#}_{goto}$ improvements – best case)

Consider a complete hidden Mealy machine $\mathcal{M}$ that is goto-conform through $\triangleright$, with $n = |Q|$ and $k = |I|$, and where all states in Q are reachable from q_0. *Consider* $\#_{goto}(q)$ *the number of transitions with goto outputs out of* q*, and let* $k_{goto} = \min_q(\#_{goto}(q))$. *Assume that each basis node has at most one basis child, so that* S *forms a long branch.* Assume the extension in [C3] is always possible. Then $L^{\#}_{goto}$ saves at least the following number of queries compared to $L^{\#}$ learning $\mathcal{M}$:

- $\max(nk_{goto}, k) + nk_{goto} - 1$ reset queries;
- $\frac{1}{2}(n(n+3) + q(q+1))k_{goto} - (n+1)$ symbol queries for $q = \lfloor \frac{k}{k_{goto}} \rfloor$.

Appendix A (*Proof of Theorem 4.21*) □

The results are, again, very close to those of Section 4.2.5 (*Theoretical improvements*) and Section 4.3.5 (*Theoretical improvements*), at least asymptotically. The primary difference from the result for id—and moreover, the reason these results are a bit ugly—is the limited number of extensions with [C3] we can perform. The basis state that is accessed with output $\triangleright$ has only k frontier states to discover; once all of these states are discovered, no more extensions can be done. These results should, however, be viewed in the context they are used in. It is very unlikely outputs that can be mapped to $\triangleright$ will be present in a program as much as outputs that can be mapped to id and ϕ. This will also be reflected in Chapter 5 (*Experimental evaluation*); they are only introduced sparsely, whereas the other two output symbols occur frequently.

4.5 Combining the optimizations

All three optimizations separately already allow for some performance improvement for compatible hidden Mealy machines. However, the additional benefit of defining optimizations without changing the foundational algorithm is that they can generally be implemented independently and then combined with little extra work. To this end, the optimizations in this paper followed a specific structure. The optimizations were structured as follows:

[C1] is an invariant on the observation tree, meant to reflect the impact of the conformity property on the learning process;

[C2] is an input preprocessing done before an output query, meant to reduce the size of inputs using implications of the invariant[2];

[C3] is an input extension done if the symbols of an output query ended in the special output, meant to get extra information out of the query based on the invariant. If $L^{\#}_{\text{Ads}}$ is used, applicable extensions (even recursively) takes place before Ads does;

[C4] is a conditional early-termination rule for when a special output encountered during the evaluation of an output query infers enough information to finish the rest of the query.

[2]One might note [C2] seems quite similar to the optimizations proposed by Fica and Otop [10]. However, a major difference is that these rewriting steps are dependent on previously observed outputs. This means we can only rewrite parts of a query that are already in the observation tree. This contrasts with that paper, because their rewriting system is already included at the start of the algorithm.

The exact form of these changes is unique per optimization, but they follow these rules. For example, $L^{\#}_{crash}$ does not have [C2] and [C3] at all. We propose a strategy to combine any number of optimizations that abide by this structure.

Proposition 4.22 (Combining optimizations)

Let $\mathbb{O}$ be a set of independent optimizations that are of the structure in this paper, that all have conformity definitions and whose changes are correctly derived from those conformity definitions. Then, the following optimization for $L^{\#}$ or $L^{\#}_{\text{ADS}}$, where:

[C1] combines the [C1]s of all optimizations;

[C2] combines the [C2]s of all optimizations, evaluated in any order, until no simplification can be applied anymore;

[C3] combines the [C3]s of all optimizations, with any [C3] evaluated if multiple are applicable at a given time;

[C4] combines the [C4]s of all optimizations,

learns a hidden Mealy machine that is conform for all optimizations in $\mathbb{O}$.

Though we do not prove this strategy formally, its correctness is primarily based on the independence of the rules between optimizations. The individual [C1] rules are all based on the individual conformity definitions, which do not depend on any of the other conformity definitions and so can be combined. The individual [C2] rules all reduce input size, so the input will eventually reach an irreducible form when they are combined. The individual [C3] and [C4] rules can only occur for one optimization at a time, as only one output can be the last one of a query, and they can thus be combined. Notably, all this proposition implies is correctness of the combined optimization, not necessarily that it is the most efficient form. After all, the order in which the preprocessing rules in [C2] are applied may make for a longer or shorter input sequence. One extension in [C3] may be more powerful than another in a given context, while both could be applied at the same time. The actual construction of this combination in a high-performance form must therefore still be manually done.

We would like to combine the optimizations in this paper (Definition 4.3 ($L^{\#}_{crash}$), Definition 4.9 ($L^{\#}_{retry}$) and Definition 4.16 ($L^{\#}_{goto}$)) using this strategy. One aspect of note for $L^{\#}_{goto}$ specifically is that it is reasonable to have multiple outputs with goto behavior that potentially represent different states. As such, our optimization will work for any number of outputs that behave like this.

Definition 4.23 ($L^\#_{CRG}$)

Consider $L^\#$ performed on $\mathcal{M}$, which is crash-conform through ϕ, retry-conform through id, and goto-conform through $\triangleright_1, \triangleright_2, \ldots, \triangleright_N$, with the following changes:

[C1] **(CRG invariant)**:

- any $q \in \mathcal{T}$ that has ϕ as the output for its unique access transition is maintained to be complete, with all outgoing transitions having crash outputs;
- for any q, q' such that $q \xrightarrow{i/\mathsf{id}} q'$ in $\mathcal{T}$ for some $i \in I$, the subtree with root q is maintained to be the same as the subtree with root q' (and vice versa);
- for all $1 \leq i \leq N$, for any $q, q' \in \mathcal{T}$ with $\triangleright_i$ as the output for their unique access transition, the subtree with root q is maintained to be the same as the subtree with root q' (and vice versa);

[C2] repeatedly apply the first possible rule of the following, until neither is possible anymore:

1. let γ be the shortest input sequence that ends in a $\triangleright_i$-output for some $1 \leq i \leq N$. Before the evaluation of an OutputQuery for σ, consider $\sigma = \sigma' \rho$ with σ' the longest prefix such that $\lambda(q_0, \sigma') = \ldots \triangleright_i$. Instead of σ, query for $\gamma \rho$;
2. before the evaluation of an OutputQuery for σ, get the output sequence ω for the longest prefix of σ contained in $\mathcal{T}$, and remove input characters from σ corresponding to the retry outputs in ω;

[C3]
- before the ResetQuery of an OutputQuery that ended in a frontier state, if the evaluation ended in $q \xrightarrow{i/\mathsf{id}} q'$, the input sequence will be extended by one character that is not yet explored in q (if possible). This finds a new frontier state for q, equivalently to [R2] but without the reset (this may happen repeatedly);
- if a state b that has $\triangleright$ as unique access transition output is in the basis, before the ResetQuery of any OutputQuery, if the evaluation ended in a goto output, the input sequence will be extended by one character that is not yet explored in b (if possible). This finds a new frontier state for b, equivalently to [R2] but without the reset (this may happen repeatedly);

[C4] during an `OutputQuery` for σ, whenever any of ϕ, id or $\triangleright_i$ for any $1 \leq i \leq N$ is encountered, the tree is updated according to [C1]. If the query is now redundant because σ is fully contained in $\mathcal{T}$, execution of the query is halted and the SUL is reset. Otherwise, the query continues where it left off.

We call this $L^\#$ variant, which provides optimizations for crash-conform, retry-conform and goto-conform machines, $L^\#_{CRG}$.

4.6 Implementing the optimizations

To get an idea of the actual performance improvements by $L^\#_{CRG}$ compared to regular $L^\#$ on real systems, we have implemented $L^\#_{CRG}$, using the $L^\#$ implementation in AALpy [26] as a base. This process was anything but simple, as parts of the optimization were fundamentally

incompatible with the way $L^{\#}$ was implemented, or extended the theory in such a way that approaches that worked before would fail for $L^{\#}_{CRG}$. In this section, we will describe and defend the most important non-trivial parts of the implementation of the theory in this thesis.

4.6.1 Storage of infinite trees

A computer has finite memory and therefore cannot store an infinite tree. Yet, we have found that all individual optimizations (and consequently, their combination) may yield an infinite subtree in the observation. However, the subtrees in question are *inferred*; they do not display arbitrary behavior, but are infinite because of either meta-information or information that is already contained within the tree. By designing and referencing the source of the information in an (infinitely) recursive way, we can then store these infinite subtrees in only a small amount of space. This approach translates to the following implementations:

- for subtrees accessed through ϕ: a single node is added, which only has transitions to itself and with ϕ as output;
- for subtrees accessed through id: a single node is added, which references its parent node for all its transitions;
- for subtrees accessed through $\triangleright$: on the first instance, a single *source* node is added that behaves just like regular nodes; on every subsequent instance, a single node is added, which references the source node for all transitions.

These implementations trivially produce the same outputs as the actual infinite subtrees would for the same input sequences. There is a downside to this approach: parent states are no longer properly maintained. However, improper parent management is a sacrifice inherent to using infinite trees on a limited-memory system. It fortunately does not interfere with our implementation of the algorithm.

4.6.2 Apartness and ADS on infinite trees

Apartness checking and ADS generation are algorithms that explore the tree: apartness checking in search of a witness (or lack thereof), and ADS generation accumulating information to produce the best possible distinguishing tree. This causes problems when introducing infinite distinguishing trees; the traversal of an infinite tree will never conclude. We therefore need to decide on rules to terminate apartness checking and ADS early—for subtrees with special outputs in particular—without breaking the core promises of these algorithms. We settled on the following rules:

- for apartness checking: dual BFS is terminated early if a transition pair consisting of twice the same special output is discovered;
- for ADS generation: a leaf ADS node is returned when an output-based partition for a special output is processed.

These rules only apply when all involved special outputs are the same. Note here that the respective algorithms have special handling for a pair/set of *different* special outputs: apartness checking would have found a witness and thus returns, and ADS generation would have partitioned based on outputs and thus separated the set. In practice, our rules amount to removing subtrees of states that are accessed through special outputs for these algorithms. Although it may seem like we are throwing away potentially useful information, this is not actually the case;

all information that we are removing either cannot lead to a new witness or is already present elsewhere in the observation tree. To show this, we can simply look at all possible cases:

- subtrees accessed through ϕ are the exact same as one another, and thus can never provide additional distinguishing information;
- subtrees accessed through id are the same as their parents' subtrees, and the distinguishing information is therefore already contained elsewhere in the tree;
- subtrees accessed through $\triangleright$ are the exact same as one another, and thus can never provide additional distinguishing information.

4.6.3 preprocessing apartness for states with special outputs

The previous subsection discussed the implementation of apartness checking for infinite trees. One might note that the rule for apartness checking only applies to special outputs found during the algorithm, not states reached through special outputs as input to the algorithm. Such cases, on paper, do not cause any problems; any branch that continues infinitely must pass a special output infinitely many times, so the rule will inevitably come into play. However, the mostly dead-end BFS search of these special subtrees caused a major spike in apartness checking iterations. The solution is to preprocess apartness checks for which one or both subtrees was reached through a special output. In particular, we want to discover if we know the two states to have an equivalent subtree. This yields the following conditions:

- two states accessed through ϕ always have equivalent subtrees;
- one state accessed through id and some other state have equivalent subtrees if their access sequences are the same after removing inputs that yield retry outputs (see also Lemma 4.10);
- two states accessed through $\triangleright$ always have equivalent subtrees.

If one of these conditions is met, the apartness checking algorithm returns False without evaluating. Otherwise, the algorithm executes as normal, even with states reached through special outputs. This measure has massively reduced the running time of apartness checking.

Chapter 5

Experimental evaluation

5.1 Test models

We tested $L^{\#}_{CRG}$ on four different models:

- Figure 3.1 (*Example coffee machine model*), with `O` $= \checkmark$, `W` $= !$, `F` $= \lightning$, `*` $= *$, and `COFFEE` = ☕:
  - ϕ: `F` (appears 10 times);
  - id: `W` (appears 4 times);
  - goto output: `*` (appears 5 times).

  Total number of states: 6; size of input alphabet: 4.

- RSA BSAFE for C, version 4.0.4, learned by Ruiter and Poll [30], with all outputs for transitions from state 8 changed to `Alert Fatal (Unexpected message)` in order to use the crash optimization:
  - ϕ: `Alert Fatal (Unexpected message)` (appears 32 times);
  - goto output: `ChangeCipherSpec & Finished` (appears 2 times).

  Total number of states: 9; size of input alphabet: 8.

- SMTP, learned by Bernas et al.[1]:
  - ϕ: `221` (appears 45 times);
  - id: `500` (appears 49 times).

  Total number of states: 20; size of input alphabet: 31.

- OpenSSH version 8.8, learned by Fiterau-Brostean et al. [11], with states 1 and 3 merged and all outputs for their outgoing transitions changed to `DISCONNECT`:
  - ϕ: `DISCONNECT` (appears 39 times);
  - id: `CH_NONE` (appears 60 times).

  Total number of states: 36; size of input alphabet: 12.

[1] We could not find a paper associated with this model, but it has been featured in the conference SKILL 2025 (in Potsdam), in a session with title “Cross-Checking SMTP Implementations with Active Automata Learning”.

5.2 Setup and reproducibility

We ran five rounds of evaluations of the algorithm: every round, we ran the algorithm on each model, for each combination of extension and separation methods, and with and without using the optimizations. We did so in a `python:3.11-slim` Docker container, limited to 8 GB of RAM and 8 logical CPU cores, with only the necessary `pydot` package installed. It was run on the AMD Ryzen 7 5700U processor with a clock-speed of 1.8 GHz, and 3200 MHz DDR4 RAM. The Docker image used, as well as instructions on how to replicate these experiments, can be found at `https://github.com/Scarletto/AALpy/tree/lsharp_optimizations`. For $L^{\#}$, we left caching on, and used the Wp-method for generating an equivalence testing suite.

5.3 Results

The raw results can be found at `https://github.com/Scarletto/AALpy/tree/lsharp_optimizations`.

5.3.1 Coffee Machine

Comparison between learning phases

Extension/ Separation	Resets			Symbols			Time		
	base	***opt.***	**impr.**	***base***	***opt.***	**impr.**	***base***	***opt.***	**impr.**
None / SepSeq	*48.4*	*19*	**60.7%**	*152.4*	*46*	**69.8%**	*0.00s*	*0.00s*	**-**
None / ADS	*43.2*	*18*	**58.3%**	*140.4*	*42*	**70.1%**	*0.00s*	*0.00s*	**-**
SepSeq / SepSeq	*37.6*	*17*	**54.8%**	*126.2*	*41*	**67.5%**	*0.00s*	*0.00s*	**-**
SepSeq / ADS	*33*	*17*	**48.5%**	*112.8*	*41*	**63.7%**	*0.00s*	*0.00s*	**-**
ADS / SepSeq	*35.8*	*16*	**55.3%**	*124*	*38.2*	**69.2%**	*0.00s*	*0.00s*	**-**
ADS / ADS	*31.2*	*16*	**48.7%**	*110.6*	*39*	**64.7%**	*0.00s*	*0.00s*	**-**

Table 5.1: Learning phase results for coffee machine model, averaged

Other interesting results

The regular algorithm always outperformed the optimized algorithm in terms of equivalence testing queries and symbols: 98 compared to 110–117 queries, and 311 compared to 370–392 symbols. The optimized algorithm finished in one fewer learning round than the regular algorithm—4 compared to 5—in about half of the tests, with no clear pattern.

5.3.2 RSA BSAFE for C

Comparison between learning phases

Extension/ Separation	Resets			Symbols			Time		
	base	*opt.*	impr.	*base*	*opt.*	impr.	*base*	*opt.*	impr.
None / SepSeq	*192*	*131*	**31.8%**	*716*	*453*	**36.7%**	*0.01s*	*0.01s*	**-**
None / ADS	*171.2*	*118*	**31.1%**	*622.2*	*397*	**36.2%**	*0.04s*	*0.02s*	**50%**
SepSeq / SepSeq	*147*	*112*	**23.8%**	*571*	*393*	**31.2%**	*0.01s*	*0.01s*	**-**
SepSeq / ADS	*163*	*116.4*	**28.6%**	*658.6*	*413.4*	**37.2%**	*0.03s*	*0.02s*	**33.3%**
ADS / SepSeq	*137*	*102*	**25.5%**	*538*	*356.4*	**33.8%**	*0.03s*	*0.02s*	**33.3%**
ADS / ADS	*125.2*	*96*	**23.3%**	*485.4*	*327*	**32.6%**	*0.04s*	*0.02s*	**50%**

Table 5.2: Learning phase results for the RSA BSAFE for C model, averaged

Other interesting results

Every round, all evaluations that share their separation method had the exact same amount of learning rounds and equivalence testing queries and symbols.

5.3.3 SMTP

Comparison between learning phases

Extension/ Separation	Resets			Symbols			Time		
	base	*opt.*	impr.	*base*	*opt.*	impr.	*base*	*opt.*	impr.
None / SepSeq	*2499.2*	*2434.8*	**2.6%**	*13200.2*	*12999*	**1.5%**	*1.08s*	*1.06s*	**1.5%**
None / ADS	*1952.8*	*1737.4*	**11%**	*10734.6*	*9791.2*	**8.8%**	*15.23s*	*11.04s*	**27.5%**
SepSeq / SepSeq	*2002*	*2028*	*-1.3%*	*11058.8*	*11145.8*	**-0.8%**	*1.05s*	*1.04s*	**0.2%**
SepSeq / ADS	*1714.2*	*1578*	**7.9%**	*9864*	*9139.8*	**7.3%**	*4.99s*	*4.89s*	**1.9%**
ADS / SepSeq	*1509.8*	*1386.4*	**8.2%**	*8814.6*	*8171.8*	**7.3%**	*12.59s*	*9.8s*	**22.1%**
ADS / ADS	*1472.8*	*1337.2*	**9.2%**	*8641.8*	*7917*	**8.4%**	*14.16s*	*9.272s*	**34.5%**

Table 5.3: Learning phase results for the SMTP model, averaged

Other interesting results

Optimized evaluations using ADS as separation method more than halved the number of equivalence testing queries and symbols compared to all runs using separating sequences for separation. Base evaluations using ADS sometimes had the same effect, but only in about half of the cases, without an apparent pattern.

5.3.4 OpenSSH

Comparison between learning phases

Extension/ Separation	Resets			Symbols			Time		
	base	*opt.*	impr.	*base*	*opt.*	impr.	*base*	*opt.*	impr.
None / SepSeq	1589.6	1295	**18.5%**	15499	12987.6	**16.2%**	0.57s	0.48s	**15.5%**
None / ADS	1182.6	977	**17.4%**	11837.6	10213.4	**13.7%**	2.93s	1.96s	**33.1%**
SepSeq / SepSeq	1289.4	1065.8	**17.3%**	13030.2	10968.6	**15.8%**	0.54s	0.45s	**16.8%**
SepSeq / ADS	1103	896.4	**18.7%**	11608	9806.8	**15.5%**	2.26s	1.49s	**34.1%**
ADS / SepSeq	1188.4	927.4	**14.8%**	11355.4	9836.6	**13.4%**	2.53s	1.71s	**32.6%**
ADS / ADS	959.6	665.8	**30.6%**	10824.4	9200	**15%**	3.27s	2.35s	**28.1%**

Table 5.4: Learning phase results for the OpenSSH model, averaged

Other interesting results

The equivalence testing queries and symbols are roughly the same with or without optimizations for the runs using separating sequences as the separation method, but the optimized algorithm does perform slightly better in this regard with ADS for separation. There was a large variety in the number of learning rounds to learn this model. While most evaluations with separating sequences as separation method ended in 22 rounds, the optimized algorithm using ADS for both extension and separation had a run in which termination took 31 rounds.

5.4 Interpretation and discussion of results

It is very clear that the optimizations generally provided a significant performance benefit in the learning phase on all measures. The actual impact varied depending on the model as one would expect, but all models were able to achieve some advantage by using the optimizations. In particular, consistent improvements of around fifteen percent on a larger model like the *OpenSSH* one is an exciting result. We believe this data shows that our optimization proposal has the potential to be useful in real world scenarios.

The only scenario in which the optimizations negatively affected the algorithm's performance was on the *SMTP* model with separating sequences as both the extension and separation method. This difference is however extremely minor, and can be reasonably discounted as a consequence of the non-determinism of the algorithm. As our experiment only had a few rounds, outliers are likely to occur.

One pattern that is consistent throughout all models is the great impact of the optimizations on the execution time of evaluations using ADS. The difference between those improvements and improvements for evaluations not using ADS can not be explained by the formal changes in the optimization alone. We believe the ADS measure described in Section 4.6 (*Implementing the optimizations*) contributes a lot to this difference; it allows ADS generation to terminate quicker, as states reached by special outputs are not explored further.

Positive effects that may be linked to the updates to ADS in the optimization are also present in the equivalence testing numbers for the *SMTP* and *OpenSSH* models. In both cases, the only evaluations to see an improvement in equivalence queries and symbols when switching to the optimized algorithm are those with ADS as the separation method. This may be caused by the optimized algorithm simply generating different ADS trees than the base algorithm. It is

plausible that this can be seen specifically when dealing with retry outputs; the id is the only special output that gets cut off in the optimized ADS generation despite still having potentially useful information (see also Section 4.6 (*Implementing the optimizations*)). We did not verify this theory, and it is also possible that this is just another outlier.

Only the coffee machine model sees an increase in equivalence testing queries and symbols when evaluating the optimized version. This is not paired with extra learning rounds. A potential explanation is that the optimized algorithm gets information more quickly and thus produces more advanced models early on, which cause a higher number of queries and symbols for equivalence testing on the first hypotheses. As the coffee machine model is small and takes only a few learning rounds, it might not be able to offset this initial effort by reaching a correct hypothesis faster.

Chapter 6

Discussion

6.1 Limitations and future work

In the definition of the optimizations, we limited ourselves to only a few categories of changes which we assumed would combine nicely with one another and with other optimizations of the same kind. It is likely that even more improvements could be made based on the conformity criteria that we established for each optimization, though we believe these would be of minimal influence. Moreover, though we have shown that the optimizations in this paper work well in tandem, we have not proven that it is generally true that optimizations defined using these categories of changes are easily combinable.

The optimizations were only run a few times, on only a few sample models. More extensive testing is needed to get a good picture of the general performance gains from using the improvements. In particular, it would be interesting to evaluate some large-scale models using the optimizations. However, we have seen that it is difficult to apply the optimizations directly to models that were already learned. We attribute this to the strict definitions of conformity. Thus, we believe that the choice of optimizations needs be accounted for during the creation of the mapping between SUL output and learner outputs.

The real-world applicability of these optimizations remains to be seen. Though the models used in the experimental evaluation of this thesis were based on real models (slightly modified), we only cherry-picked the models after they were already fully learned. We are not sure how often users can guarantee behavior corresponding to our conformity definitions purely from background knowledge or personal experience.

Though the effects of the optimization as a whole are thoroughly researched in this thesis, the impact of each individual part of the optimization is not considered in the experimental evaluation. Due to time constraints, we did not get around to doing this properly. It would be interesting to see the results of an experiment with tactically placed counters to represent the application of a rule in an optimization. Research into this would also allow for a consideration whether to keep or drop a part based on the tradeoff between computations used and queries saved.

We believe this paper provides a good foundation for future assumption-based optimizations for $L^{\#}$ using special outputs. One could choose conformance criteria that are more lax or more expressive than those defined in this paper to adjust the current optimizations. For example, crash-conformance could easily be adjusted to allow the self-loops in the sink state to have different outputs than the crash output, or maybe even different outputs than one another. Alternatively, new optimizations could be defined, potentially using or building on the format of the optimizations in this paper. Some examples of future optimizations that may have potential are the following:

- an optimization for outputs that indicate a reversal of the most recent change (and thus the return to the previous state);
- an adjustment to the optimizations in this paper to allow sequences of outputs to represent certain behavior, rather than just singular outputs;
- optimizations in which the behaviors of outputs depend on which “flags” are set, essentially modeling a form of conditional behavior.

We also think there is merit in trying to use the optimizations for the equivalence queries. An equivalence query might be cut off early if it reaches a special output, or the test suite generation algorithm might not generate inputs that go through a transition with a special output.

6.2 Conclusion

Active automata learning algorithms are a powerful tool for model learning and verification, but they are held back by their long execution time, especially for larger systems. In this active field, even small improvements are therefore significant. By approaching the problem from an underdeveloped angle, we have achieved such a small victory, albeit a conditional one. We have shown that one can effectively leverage background knowledge about some behaviors related to certain outputs to speed up the $L^{\#}$ algorithm. This thesis also lays the foundation for further development of this niche; by proposing a generalized framework for these types of optimizations and explicitly documenting the implementation process, future researchers should be able to use this thesis as a blueprint for their own research. We believe this niche is a piece in the larger puzzle that is optimizing active automata learning, and we are excited to see how it is expanded upon in the future.

Bibliography

[1] Bernhard K. Aichernig, Martin Tappler, and Felix Wallner. 2020. Benchmarking Combinations of Learning and Testing Algorithms for Active Automata Learning. In *International Conference on Tests and Proofs*. Springer, Bergen, Norway, 3–22. ISBN: 978-3-030-50995-8. doi:10.1007/978-3-030-50995-8_1.

[2] Jie An, Mingshuai Chen, Bohua Zhan, Naijun Zhan, and Miaomiao Zhang. 2020. Learning One-Clock Timed Automata. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Dublin, Ireland, 444–462. ISBN: 978-3-030-45190-5. doi:10.1007/978-3-030-45190-5_25.

[3] Jie An, Lingtai Wang, Bohua Zhan, Naijun Zhan, and Miaomiao Zhang. 2021. Learning real-time automata. *Science China Information Sciences*, 64, 9, 192103. doi:10.1007/s11432-019-2767-4.

[4] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and Computation*, 75, 2, 87–106. doi:10.1016/0890-5401(87)90052-6.

[5] Raphaël Berthon, Adrien Boiret, Guillermo A. Pérez, and Jean-François Raskin. 2021. Active Learning of Sequential Transducers with Side Information About the Domain. In *International Conference on Developments in Language Theory*. Springer, Porto, Portugal, 54–65. ISBN: 978-3-030-81508-0. doi:10.1007/978-3-030-81508-0_5.

[6] Carlos Diego N. Damasceno, Mohammad Reza Mousavi, and Adenilso da Silva Simao. 2019. Learning to Reuse: Adaptive Model Learning for Evolving Systems. In *International Conferences of Integrated Formal Methods*. Wolfgang Ahrendt and Silvia Lizeth Tapia Tarifa, (Eds.) Springer, Bergen, Norway, 138–156. ISBN: 978-3-030-34968-4. doi:10.1007/978-3-030-34968-4_8.

[7] Normann Decker, Peter Habermehl, Martin Leucker, and Daniel Thoma. 2014. Learning Transparent Data Automata. In *International Conference on Application and Theory of Petri Nets and Concurrency*. Springer, Tunis, Tunesia, 130–149. ISBN: 978-3-319-07734-5. doi:10.1007/978-3-319-07734-5_8.

[8] Simon Dierl, Paul Fiterau-Brostean, Falk Howar, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist. 2024. Scalable Tree-based Register Automata Learning. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Luxembourg City, Luxembourg, 87–108. ISBN: 978-3-031-57249-4. doi:10.1007/978-3-031-57249-4_5.

[9] Tiago Ferreira, Gerco van Heerdt, and Alexandra Silva. 2022. Tree-Based Adaptive Model Learning. In *A Journey from Process Algebra via Timed Automata to Model Learning : Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday*. Nils Jansen, Mariëlle Stoelinga, and Petra van den Bos, (Eds.) Springer, Cham, Switzerland, 164–179. ISBN: 978-3-031-15629-8. doi:10.1007/978-3-031-15629-8_10.

[10] Michał Fica and Jan Otop. 2025. Active Automata Learning with Advice. In *European Conference on Artificial Intelligence (ECAI 2025)*. IOS Press, Bologna, Italy, 1655–1662. ISBN: 978-1-64368-631-8. doi:10.3233/FAIA250992.

[11] Paul Fiterau-Brostean, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist. 2023. Automata-Based Automated Detection of State Machine Bugs in Protocol Implementations. In *Network and Distributed System Security Symposium*. Internet Society, San Diego, CA, USA. ISBN: 9781891562839. doi:10.14722/ndss.2023.23068.

[12] Markus Theo Frohme. 2019. Active Automata Learning with Adaptive Distinguishing Sequences. (Feb. 4, 2019). arXiv: 1902.01139 [cs]. doi:10.48550/arXiv.1902.01139. Pre-published.

[13] Bharat Garhewal, Frits Vaandrager, Falk Howar, Timo Schrijvers, Toon Lenaerts, and Rob Smits. 2020. Grey-Box Learning of Register Automata. In *International Conference on Integrated Formal Methods*. Springer, Lugano, Switzerland, 22–40. ISBN: 978-3-030-63461-2. doi:10.1007/978-3-030-63461-2_2.

[14] Alex Groce, Doron Peled, and Mihalis Yannakakis. 2002. Adaptive Model Checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Grenoble, France, 357–370. ISBN: 978-3-540-46002-2. doi:10.1007/3-540-46002-0_25.

[15] Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. 2012. Inferring Canonical Register Automata. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, Philadelphia, PA, USA, 251–266. ISBN: 978-3-642-27940-9. doi:10.1007/978-3-642-27940-9_17.

[16] Falk M. Howar. 2012. *Active Learning of Interface Programs*. PhD Dissertation. Technische Universität Dortmund, (June 26, 2012). HDL: 2003/29486. doi:10.17877/DE290R-4817.

[17] Eric Hsiung, Swarat Chaudhuri, and Joydeep Biswas. 2024. Learning Quantitative Automata Modulo Theories. (Nov. 15, 2024). arXiv: 2411.10601 [cs]. doi:10.48550/arXiv.2411.10601. Pre-published.

[18] Malte Isberner, Falk Howar, and Bernhard Steffen. 2014. The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning. In *International Conference on Runtime Verification*. Borzoo Bonakdarpour and Scott A. Smolka, (Eds.) Springer, Toronto, ON, Canada, 307–322. ISBN: 978-3-319-11164-3. doi:10.1007/978-3-319-11164-3_26.

[19] Xiaodong Jia and Gang Tan. 2024. V-Star: Learning Visibly Pushdown Grammars from Program Inputs. *Proceedings of the ACM on Programming Languages*, 8, PLDI, Article 228, 2003–2026. doi:10.1145/3656458.

[20] Michael J. Kearns and Umesh Vazirani. 1994. *An Introduction to Computational Learning Theory*. MIT Press. 230 pp. ISBN: 978-0-262-11193-5. Google Books: vCA01wY6iywC.

[21] Rick Koenders and Joshua Moerman. 2024. Output-decomposed Learning of Mealy Machines. (May 14, 2024). arXiv: 2405.08647 [cs]. doi:10.48550/arXiv.2405.08647. Pre-published.

[22] Loes Kruger, Sebastian Junges, and Jurriaan Rot. 2025. State Matching and Multiple References in Adaptive Active Automata Learning. In *International Symposium on Formal Methods*. Springer, Milan, Italy, 267–284. ISBN: 978-3-031-71162-6. doi:10.1007/978-3-031-71162-6_14.

[23] Faezeh Labbaf, Jan Friso Groote, Hossein Hojjat, and Mohammad Reza Mousavi. 2023. Compositional Learning for Interleaving Parallel Automata. In *International Conference on Foundations of Software Science and Computation Structures*. Springer, Paris, France, 413–435. ISBN: 978-3-031-30829-1. doi:10.1007/978-3-031-30829-1_20.

[24] Jakub Michaliszyn and Jan Otop. 2022. Learning Deterministic Visibly Pushdown Automata Under Accessible Stack. In *International Symposium on Mathematical Foundations of Computer Science (MFCS 2022)* Article 74. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany. ISBN: 978-3-95977-256-3. doi:10.4230/LIPIcs.MFCS.2022.74.

[25] Jakub Michaliszyn and Jan Otop. 2022. Learning infinite-word automata with loop-index queries. *Artificial Intelligence*, 307. doi:10.1016/j.artint.2022.103710.

[26] Edi Muškardin, Bernhard K. Aichernig, Andrea Pferscher, and Martin Tappler. 2022. AALpy: An Active Automata Learning Library. *Innovations in Systems and Software Engineering*, 18, (Mar. 2022), 1–10. doi:10.1007/s11334-022-00449-3.

[27] Thomas Neele and Matteo Sammartino. 2023. Compositional Automata Learning of Synchronous Systems. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, Paris, France, 47–66. ISBN: 978-3-031-30826-0. doi:10.1007/978-3-031-30826-0_3.

[28] Andrea Pferscher and Bernhard K. Aichernig. 2020. Learning Abstracted Non-deterministic Finite State Machines. In *International Conference on Testing Software and Systems*. Valentina Casola, Alessandra De Benedictis, and Massimiliano Rak, (Eds.) Springer, Naples, Italy, 52–69. ISBN: 978-3-030-64880-0. doi:10.1007/978-3-030-64881-7_4.

[29] R. L. Rivest and R. E. Schapire. 1989. Inference of finite automata using homing sequences. In *Annual ACM Symposium on Theory of Computing (STOC '89)*. Association for Computing Machinery, New York, NY, USA, 411–420. ISBN: 978-0-89791-307-2. doi:10.1145/73007.73047.

[30] Joeri de Ruiter and Erik Poll. 2015. Protocol State Fuzzing of TLS Implementations. In *USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., USA, 193–206. ISBN: 978-1-939133-11-3. Retrieved Jan. 16, 2026 from https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter.

[31] Bernhard Steffen, Falk Howar, and Maik Merten. 2011. Introduction to Active Automata Learning from a Practical Perspective. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*. Springer, Bertinoro, Italy, 256–296. ISBN: 978-3-642-21455-4. doi:10.1007/978-3-642-21455-4_8.

[32] Martin Tappler, Edi Muškardin, Bernhard K. Aichernig, and Ingo Pill. 2021. Active Model Learning of Stochastic Reactive Systems. In *International Conference on Software Engineering and Formal Methods*. Springer, virtual, 481–500. ISBN: 978-3-030-92124-8. doi:10.1007/978-3-030-92124-8_27.

[33] Yu Teng, Miaomiao Zhang, and Jie An. 2024. Learning Deterministic Multi-Clock Timed Automata. In *ACM International Conference on Hybrid Systems: Computation and Control (HSCC '24)*. Association for Computing Machinery, Hong Kong, China, 1–11. ISBN: 979-8-4007-0522-9. doi:10.1145/3641513.3650124.

[34] A. S. Troelstra and H. Schwichtenberg. 2000. *Basic Proof Theory*. (2nd ed.). *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK. ISBN: 978-0-521-77911-1. doi:10.1017/CBO9781139168717.

[35] Frits Vaandrager. 2017. Model learning. *Communications of the ACM*, 60, 2, 86–95. doi:10.1145/2967606.

[36] Frits Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. 2022. A New Approach for Active Automata Learning Based on Apartness. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Munich, Germany, 223–243. ISBN: 978-3-030-99524-9. doi:10.1007/978-3-030-99524-9_12.

[37] Michele Volpato and Jan Tretmans. 2015. Approximate Active Learning of Nondeterministic Input Output Transition Systems. *Electronic Communications of the EASST*, 72. doi:10.14279/tuj.eceasst.72.1008.

[38] Masaki Waga. 2023. Active Learning of Deterministic Timed Automata with Myhill-Nerode Style Characterization. In *International Conference on Computer Aided Verification*. Springer, Paris, France, 3–26. ISBN: 978-3-031-37706-8. doi:10.1007/978-3-031-37706-8_1.

[39] Felix Wallner, Bernhard K. Aichernig, Florian Lorber, and Martin Tappler. 2025. Mutating Skeletons: Learning Timed Automata via Domain Knowledge. In *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2025)*. IEEE, Naples, Italy, 67–77. doi:10.1109/ICSTW64639.2025.10962513.

[40] Stephan Windmüller, Johannes Neubauer, Bernhard Steffen, Falk Howar, and Oliver Bauer. 2013. Active continuous quality control. In *International ACM Sigsoft Symposium on Component-based Software Engineering (CBSE '13)*. Association for Computing Machinery, Vancouver, BC, Canada, 111–120. ISBN: 978-1-4503-2122-8. doi:10.1145/2465449.2465469.

[41] Runqing Xu, Jie An, and Bohua Zhan. 2022. Active Learning of One-Clock Timed Automata Using Constraint Solving. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, virtual, 249–265. ISBN: 978-3-031-19992-9. doi:10.1007/978-3-031-19992-9_16.

Appendix A

Proofs

Proof of Theorem 2.6

Proves Theorem 2.6 (*Correspondence apartness source and tree*).

Proof. As $p \#^{\mathcal{T}} q$, there exists a witness $\sigma \in I^*$ such that $\lambda^{\mathcal{T}}(p, \sigma)\downarrow$, $\lambda^{\mathcal{T}}(q, \sigma)\downarrow$ and $\lambda^{\mathcal{T}}(p, \sigma) \neq \lambda^{\mathcal{T}}(q, \sigma)$ Def. 2.5. Because $\mathcal{T}$ simulates $\mathcal{M}$, for $p' = f(p)$ and $q' = f(q)$, $\lambda^{\mathcal{M}}(p', \sigma) = \lambda^{\mathcal{T}}(p, \sigma)$ and $\lambda^{\mathcal{M}}(q', \sigma) = \lambda^{\mathcal{T}}(q, \sigma)$ Def. 2.3. Consequently, $\lambda^{\mathcal{M}}(q', \sigma) \neq \lambda^{\mathcal{M}}(p', \sigma)$ and thus $\sigma \vdash f(p) \#^{\mathcal{M}} f(q)$. □

Proof of Theorem 2.8

Proves Theorem 2.8 (*Basis valid after expansion*).

Proof. All states in S are pairwise apart (Def. 2.7). So, for arbitrary $p, q \in S$, there exists some σ such that $\sigma \vdash p \#^{\mathcal{S}} q$ (Def. 2.5). As $\mathcal{T}$ is a supertree of $\mathcal{S}$, $\lambda^{\mathcal{T}}(p, \sigma) = \lambda^{\mathcal{S}}(p, \sigma)$ and $\lambda^{\mathcal{T}}(q, \sigma) = \lambda^{\mathcal{S}}(q, \sigma)$. As a result, $\sigma \vdash p \#^{\mathcal{T}} q$. By generality, all states in $B^{\mathcal{S}}$ are thus pairwise apart in $\mathcal{T}$ also, fulfilling property (1). Property (2) trivially carries over from $\mathcal{S}$ to $\mathcal{T}$. Consequently, S is a basis for $\mathcal{T}$. □

Proof of Theorem 4.5

Proves Theorem 4.5 ($L^{\#}_{crash}$ *correctness*).

Proof. The conditions to reach termination (in [R4]) are completely unchanged compared to the regular $L^{\#}$. Moreover, no other ways to terminate the program are added in $L^{\#}_{crash}$. This means that, like in regular $L^{\#}$, correctness amounts to showing termination. $L^{\#}_{crash}$ trivially does not block; no rule had its activation prerequisites changed.

As $L^{\#}$ terminates through the regular learning process, it then suffices to show that any $L^{\#}_{crash}$ rule application gets the same or more information than the counterpart in $L^{\#}$. This implies that, at any point in the algorithm, the $L^{\#}$ observation tree is a subtree of the $L^{\#}_{crash}$ one when both algorithms are evaluating the same rules. We make a rule-based inductive argument that this is the case by verifying that at least an observation for $L^{\#}$ is also added to $L^{\#}_{crash}$ with the same operation. For simplicity, we will assume a situation in which the observation trees are exactly equal before application of the rule.

[R1] this rule is unchanged from $L^{\#}$, and thus the outcome will be the same between the two algorithms;

[R2] the crash invariant will fill in an infinite crash subtree on discovering a ϕ to a frontier state, but part of that tree is the actual frontier state which $L^{\#}$ also finds;

[R3] while this might again lead to inferred observations if a crash output is found, the destination of the identification sequence will then just be one of those inferred observations;

[R4] this rule is unchanged.

We can easily apply this argument to $L^{\#}_{\text{ADS}}$ and $L^{\#}_{crash}$ with ADS also. If no crash output is found during a query, both algorithms behave the same. However, if one is found, $L^{\#}_{\text{ADS}}$ will continue while $L^{\#}_{crash}$ with ADS stops. In the end though, all $L^{\#}_{\text{ADS}}$ will find is more crash outputs; the ones that are already inferred for $L^{\#}_{crash}$ with ADS.

We have thus shown every observation tree obtained by $L^{\#}_{crash}$ can also be obtained using $L^{\#}$ instead. We have shown that this argument translates well for $L^{\#}_{\text{ADS}}$ compared to $L^{\#}_{crash}$ with ADS also. Consequently, by correctness of $L^{\#}$ respectively $L^{\#}_{\text{ADS}}$, it is shown that $L^{\#}_{crash}$ respectively $L^{\#}_{crash}$ with ADS can correctly learn $\langle \mathcal{M}; \phi \rangle$. □

Proof of Theorem 4.6

Proves Theorem 4.6 ($L^{\#}_{crash}$ *improvements – general case*).

Proof.

- **Reset queries** reset queries are invoked by [R2], [R3] and as part of PROCCOUNTEREX in [R4].
  - [R2] is applied less, because one of the basis states will be a state with a crash output in its access transition. We do not have to discover its frontier states, and it cannot have basis children, so we save k reset queries in [R2];
  - [R3] is applied less than in $L^{\#}$, because the frontier states of the crash state do not need to be identified, and neither do the crash frontier states. At least one query is needed for identification of each frontier state. This means we save at least $t_{crash} + 1$ reset queries in [R3], with the additional one being the crash state that becomes a basis state;
  - output queries in PROCCOUNTEREX are on average applied less than in $L^{\#}$, but the worst case complexity does not change. If PROCCOUNTEREX places an output query for a sequence which passes a state representing the sink state in the observation tree, we have ensured the SUL is never actually queried through [C4]. Since there is no guarantee this ever happens, we save no queries here.

  As a result, we save at least $k + t_{crash} + 1$ reset queries using the optimization.
- **Symbol queries** the amount of symbol queries is naturally dependent on the amount of reset queries, as reset queries always follow sequences of symbol queries.
  - [R2] only saves queries for the k frontier states connected to the crash basis state. If that basis state is a direct child of the root, we save $2k$ symbol queries in [R2];

- at least one query is needed to identify a frontier state, and the needed symbols are at least two: one for the access sequence and one for the identification. The amount of frontier states connected to the root is limited to k. In the worst case, the crash basis state will be one of those. The minimum savings are then $2k + 3(t_{crash} - k)$ if $t_{crash} > k$, or $2t_{crash}$ otherwise;
- output queries in PROCCOUNTEREX are influenced by [C4], but this does not guarantee a reduction.

As a result, we save at least $3t_{crash} + k$ symbols if $t_{crash} > k$, and at least $2t_{crash} + 2k$ otherwise.

□

Proof of Theorem 4.7

Proves Theorem 4.7 ($L^{\#}_{crash}$ *improvements – best case*).

Proof.

- **Reset queries** the same savings as in THM. 4.13, as those savings were not dependent on the distribution of retry outputs; $k + nk_{crash} + 1$ (as $nk_{crash} = t_{crash}$ in that theorem).
- **Symbol queries** the amount of symbol queries is naturally dependent on the amount of reset queries, as reset queries always follow sequences of symbol queries.
  - Except for frontier states of the crash basis state, all frontier states still must be discovered. Because the crash basis state cannot have basis children, it must be at the end of the branch. This means that basis state has an access sequence of length $n - 1$, and so each of its frontier states takes n symbols to discover. Total savings in [R2] then are kn symbols;
  - every identification costs at least one symbol, plus the access sequence to get to the node to identify. Since we know the distribution of crash outputs, we can add up the length of these minimal identification costs to total $\sum_{i=2}^{n} k_{crash} i = k_{crash} \sum_{i=2}^{n} i$;
  - no symbol queries are necessarily saved in PROCCOUNTEREX.

  As a result, we save at least $kn + k_{crash} \sum_{i=2}^{n} i = kn + \frac{1}{2}(n-1)nk_{crash}$.

□

Proof of Lemma 4.10

Proves Lemma 4.10.

Proof. Let $\sigma' = \sigma[\ldots k] i \sigma[(k+1)\ldots]$ for some k, where $\lambda(\delta(q_0, \sigma[\ldots k]), i) = \mathsf{id}$. Then $\delta(q_0, \sigma[\ldots k]) = \delta(q_0, \sigma[\ldots k]\ i)$ (DEF. 4.8). As a consequence, $q = \delta(q_0, \sigma) = \delta(q_0, \sigma') = q'$.

Because equality is transitive, the above can be applied inductively to show equality between any q, q' for any σ, σ' of which the output is only different by some retry outputs. For example, if σ gets n retry outputs and σ' gets n', then $n + n'$ applications would derive the equality. In conclusion, $q = q'$. □

Proof of Theorem 4.11

Proves Theorem 4.11 (*Retry invariant soundness*).

Proof. We must show that every observation in $\mathcal{T}'$ is consistent with $\mathcal{M}$. Consider $q' \in Q^{\mathcal{T}'}$ with access sequence σ' with output sequence ω'. We know a $q \in Q^{\mathcal{T}}$ with access sequence σ, output sequence ω must exist such that ω is equal to ω' if the retry outputs were removed from both. Then it must hold that $\delta^{\mathcal{M}}(q_0, \sigma) = \delta^{\mathcal{M}}(q_0, \sigma')$ (LEM. 4.10).

Since q and q' represent the same state in $\mathcal{M}$, observations made from them will yield the same output. Moreover, we know q is consistent with $\mathcal{M}$, because $\mathcal{T}$ is an observation tree for $\mathcal{M}$. As a consequence, q and q' having the same subtree in $\mathcal{T}$ does not create an inconsistency. In conclusion, after inductively applying this argument, it is shown that $\mathcal{T}'$ is a subtree for $\mathcal{M}$. □

Proof of Theorem 4.12

Proves Theorem 4.12 ($L^{\#}_{retry}$ *correctness*).

Proof. The conditions to reach termination (in [R4]) are completely unchanged opposed to the regular $L^{\#}$. Moreover, no other ways to terminate the program are added in $L^{\#}_{retry}$. This means that, like in regular $L^{\#}$, correctness amounts to showing termination. $L^{\#}_{retry}$ trivially does not block; no rule had its activation prerequisites changed.

As $L^{\#}$ terminates through the regular learning process, it then suffices to show that any $L^{\#}_{retry}$ rule application gets the same or more information than the counterpart in $L^{\#}$. This implies that, at any point in the algorithm, the $L^{\#}$ observation tree should be a subtree of the $L^{\#}_{retry}$ one when both algorithms are evaluating the same rules. We make a rule-based inductive argument that this is the case by verifying that at least an observation for $L^{\#}$ is also added to $L^{\#}_{retry}$ with the same operation. For simplicity, we will assume a situation in which the observation trees are exactly equal before application of the rule.

[R1] this rule is unchanged from $L^{\#}$, and thus the outcome will be the same between the two algorithms;

[R2] this rule queries the same information as $L^{\#}$. The application of [C2] affects the number of symbols in the query but not the actual result, because it is a direct consequence of the invariant in [C1]. However, as is described in the description of [C3], extension of the output query is done to get addition frontier states that one would otherwise get through [R2]. This means that the information from a query with n such extensions in $L^{\#}_{retry}$ could otherwise have been gotten through $n + 1$ separate applications of [R2] in $L^{\#}$. Moreover, because of [C1], even more inferred observations could be added to the tree;

[R3] this rule is unchanged compared to $L^{\#}$ as [C2] does not cause a different observation in this rule, but the retry invariant will likely cause more inferred observations in $L^{\#}_{retry}$;

[R4] this rule is unchanged.

As adaptive distinguishing sequences are computed adaptively (as the name suggests), the actual queries for the ADS variants will likely differ between the two algorithms, as we have shown $L^{\#}_{retry}$ has the edge in terms of number of observations. However, correctness for the variant of $L^{\#}_{retry}$ with adaptive distinguishing sequences does follow the same argument as that

of $L^{\#}_{\text{ADS}}$ compared to $L^{\#}$. After all, the only difference between $L^{\#}$ and $L^{\#}_{\text{ADS}}$ is the addition of a suffix for each output query.

We have thus shown every observation tree obtained by $L^{\#}_{retry}$ can also be obtained using $L^{\#}$ instead. We have shown that the principles for advanced distinguishing sequences are not violated from $L^{\#}$ to $L^{\#}_{retry}$. Consequently, by correctness of $L^{\#}$ respectively $L^{\#}_{\text{ADS}}$, it is shown that $L^{\#}_{retry}$ respectively $L^{\#}_{retry}$ with ADS can correctly learn $\langle \mathcal{M}; \mathsf{id} \rangle$. □

Proof of Theorem 4.13

Proves Theorem 4.13 ($L^{\#}_{retry}$ *improvements – general case*).

Proof.

- **Reset queries** reset queries are invoked by [R2], [R3] and as part of PROCCOUNTEREX in [R4].
  - [R2] is applied less due to the possibility of reapplications of [R2] before resets (due to [C3]). Every extension saves one reset query. Because of our assumption, all discoveries of id turn into an extension. Therefore, we save at least t_{retry} reset queries;
  - [R3] is applied less, to a similar degree as in THM. 4.6. The reason is different however; loop frontier states may not have their full subtree known already, but they do have the same subtree as their basis predecessor. This means, by virtue of that predecessor being apart from all other basis states (DEF. 2.5), they are identified as their predecessor. This saves at least one reset query for identification per frontier state reached by id. The savings are at least t_{retry} reset queries;
  - no real savings happen in PROCCOUNTEREX, consistent with THM. 4.6 also.

  As a result, we save at least $2t_{retry}$ reset queries using the optimization.

- **Symbol queries** the amount of symbol queries is naturally dependent on the amount of reset queries, as reset queries always follow sequences of symbol queries.
  - [R2] queries will consume at least $t_{retry} - k$ less symbol queries if $t_{retry} > k$, analogous to THM. 4.6;
  - [R3] saves $2k + 3(t_{retry} - k)$ symbols if $t_{crash} > k$, or $2t_{retry}$ otherwise, by the same reasoning as THM. 4.6;
  - next to [C4], [C2] also influences the number of symbols for PROCCOUNTEREX output queries, but not consistently so.

  As a result, we save at least $4t_{retry} - 2k$ symbols if $t_{retry} > k$, and at least $2t_{retry}$ otherwise.

□

Proof of Theorem 4.14

Proves Theorem 4.14 ($L^{\#}_{retry}$ *improvements – best case*).

Proof.

- **Reset queries** the same savings as in THM. 4.13, as those savings were not dependent on the distribution of retry outputs; $2nk_{retry}$ (as $nk_{retry} = t_{retry}$ in that theorem).

- **Symbol queries** the amount of symbol queries is naturally dependent on the amount of reset queries, as reset queries always follow sequences of symbol queries.
  - We save i symbols per occurrence of id in [R2], where i is the length of the access sequence to the basis state that we want to expand. Through our assumptions of the form of S, the total benefit is then $\sum_{i=0}^{n-1} k_{retry} i = k_{retry}(\sum_{i=0}^{n-1} i)$;
  - every identification costs at least one symbol, plus the access sequence to get to the node to identify. Consequently, we save at least $\sum_{i=2}^{n+1} k_{retry} i = k_{retry}(\sum_{i=2}^{n+1} i)$ symbol queries on [R3];
  - no symbol queries are necessarily saved in ProcCounterEx.

As a result, we save at least $k_{retry}(\sum_{i=0}^{n-1} i) + k_{retry}(\sum_{i=2}^{n+1} i) = k_{retry}((\sum_{i=0}^{n-1} i) + (\sum_{i=0}^{n-1}(i + 2))) = 2k_{retry}(\sum_{i=0}^{n-1}(i+1)) = n(n+1)k_{retry}$.

□

Proof of Lemma 4.17

Proves Lemma 4.17.

Proof. Consider $\sigma = \gamma\ \rho$ and $\sigma' = \gamma'\ \rho$, such that ρ is the shared suffix which has $\triangleright$ as the first transition output. Let $p = \delta(q_0, \gamma)$ and $p' = \delta(q_0, \gamma')$. Then, it holds that $\lambda(p, \rho[0]) = \triangleright$ and $\lambda(p', \rho[0]) = \triangleright$. Thus, $\delta(p, \rho[0]) = \delta(q_0, \gamma\ \rho[0]) = \delta(q_0, \gamma'\ \rho[0]) = \delta(p', \rho[0])$ (Def. 4.15). $q = \delta(q_0, \gamma\ \rho) = \delta(q_0, \gamma'\ \rho) = q'$ must then hold, by determinism of Mealy machines. In conclusion, $q = q'$. □

Proof of Theorem 4.18

Proves Theorem 4.18 (*Goto invariant soundness*).

Proof. We must show that every observation in $\mathcal{T}'$ is consistent with $\mathcal{M}$. Consider $q' \in Q^{\mathcal{T}'}$ with access sequence σ'. We know a $q \in Q^{\mathcal{T}}$ with access sequence σ must exist, such that σ shares a suffix with σ' with the output of the first transition of that suffix being $\triangleright$ in both σ and σ'. Then it must hold that $\delta^{\mathcal{M}}(q_0, \sigma) = \delta^{\mathcal{M}}(q_0, \sigma')$ (Lem. 4.17).

The rest of the argument mimics exactly that of Appendix A (*Proof of Lemma 4.10*). □

Proof of Theorem 4.19

Proves Theorem 4.19 ($L^{\#}_{goto}$ *correctness*).

Proof. This proof is nearly identical to Appendix A (*Proof of Theorem 4.12*). The sole deviation is that [C3] is conditional and may also be applied to [R3], but that extension rule only serves to save symbol queries but does not affect the correctness (as is also established in that proof).

Consequently, by correctness of $L^{\#}$ respectively $L^{\#}_{\text{Ads}}$, it is shown that $L^{\#}_{goto}$ respectively $L^{\#}_{goto}$ with Ads can correctly learn $\langle \mathcal{M}; \triangleright \rangle$. □

Proof of Theorem 4.20

Proves Theorem 4.20 ($L^{\#}_{goto}$ *improvements – general case*).

Proof.

- **Reset queries** reset queries are invoked by [R2], [R3] and as part of ProcCounterEx in [R4].
  - [R2] is applied less due to the possibility of reapplications of [R2] before resets (due to [C3]). Every extension saves one reset query. Because of our assumption, all discoveries of $\triangleright$ due to [R2] turn into an extension. However, the number of extensions is upper bounded by the number of inputs. Therefore, we save exactly $\max(t_{goto}, k)$ reset queries;
  - since all $\triangleright$-states share a subtree, any [R3] applications on one of them helps identify all of them. This translates to reducing at least one identification query for all but one state. Thus, we save at least $t_{goto} - 1$ reset queries here;
  - no real savings happen in ProcCounterEx.

  As a result, we save at least $\max(t_{goto}, k) + t_{goto} - 1$ reset queries using the optimization.

- **Symbol queries** the amount of symbol queries is naturally dependent on the amount of reset queries, as reset queries always follow sequences of symbol queries.
  - Every extension costs one more symbol query, but saves at least one also. In the worst case, goto outputs are clustered near the root. For all frontier states of the root state, no symbol queries are saved. The upper limit of k frontier states from the root is the same as the upper limit of k for the number of extensions. In the worst case, this means no saved symbol queries;
  - at least one query is needed to identify a frontier state, and the needed symbols are at least two: one for the access sequence and one for the identification. Similarly to [R2], the amount of frontier states connected to the root is limited to k. The minimum savings are then $2k + 3(t_{goto} - k - 1)$ if $t_{goto} - 1 > k$, or $2(t_{goto} - 1)$ otherwise;
  - output queries in ProcCounterEx may be influenced by [C2], but since this is not guaranteed, no reduction of symbols takes place here.

  As a result, we save at least $3(t_{goto} - 1) - k$ symbols if $t_{goto} - 1 > k$, and at least $2(t_{goto} - 1)$ otherwise.

□

Proof of Theorem 4.21

Proves Theorem 4.21 ($L^{\#}_{goto}$ *improvements – best case*).

Proof.

- **Reset queries** the same savings as in Thm. 4.20, as those savings were not dependent on the distribution of retry outputs; $\max(nk_{goto}, k) + nk_{goto} - 1$ (as $nk_{goto} = t_{goto}$ in that theorem).

- **Symbol queries** the amount of symbol queries is naturally dependent on the amount of reset queries, as reset queries always follow sequences of symbol queries.
  - The number of extensions from [R2] is upper bounded by k. The first k_{goto} expansions come from the root and save no symbols, the second k_{goto} save one symbol, the third k_{goto} save two and so on. The total number of saved symbols then equals $\sum_{i=0}^{k-1} \lfloor \frac{i}{k_{goto}} \rfloor$. We use the slightly laxer lower bound $k_{goto} \sum_{i=0}^{\lfloor \frac{k}{k_{goto}} \rfloor} i$, to better be able to compare the results against those of the other optimizations;
  - every identification costs at least one symbol, plus the access sequence to get to the node to identify. Consequently, we save at least $\sum_{i=2}^{n+1} k_{goto} i = k_{goto}(\sum_{i=2}^{n+1} i)$ symbol queries in [R3], if we did not need to identify $\triangleright$-frontier states at all. However, we need at least one identification query for those; removing the query that saves the most symbols yields $k_{goto}(\sum_{i=2}^{n+1} i) - (n+1)$;
  - no symbol queries are saved in ProcCounterEx.

As a result, we save at least $k_{goto}(\sum_{i=2}^{n+1} i) + k_{goto}(\sum_{i=0}^{\lfloor \frac{k}{k_{goto}} \rfloor} i) - (n+1) = \frac{n(n+3) + \lfloor \frac{k}{k_{goto}} \rfloor (\lfloor \frac{k}{k_{goto}} \rfloor + 1)}{2} k_{goto} - (n+1)$.

□

Appendix B

AI Disclosure

AI Usage Card for *Assumption-based optimizations for $L^\#$ using special outputs*

CORRESPONDENCE
Marijn Meuleman

CONTACT
marijn.meuleman@ru.nl

AFFILIATION
Radboud University

PROJECT NAME
Bachelor's Thesis: Assumption-based optimizations for $L^\#$ using special outputs

IDEATION
ChatGPT 5.0/5.1

GENERATING IDEAS, OUTLINES, AND WORKFLOWS
-

IMPROVING EXISTING IDEAS
Names, symbols, and syntax for new definitions.

FINDING GAPS OR COMPARE ASPECTS OF IDEAS
-

LITERATURE REVIEW
Consensus, ChatGPT 5.0/5.1

FINDING LITERATURE
A basis of potentially related work to review and use as a source of more potentially related work.

FINDING EXAMPLES FROM KNOWN LITERATURE
-

ADDING ADDITIONAL LITERATURE FOR EXISTING STATEMENTS AND FACTS
-

COMPARING LITERATURE
Verification of some claims, particularly in the introduction.

METHODOLOGY -	PROPOSING NEW SOLUTIONS TO PROBLEMS -	FINDING ITERATIVE OPTIMIZATIONS -
	COMPARING RELATED SOLUTIONS -	
EXPERIMENTS -	DESIGNING NEW EXPERIMENTS -	EDITING EXISTING EXPERIMENTS -
	FINDING, COMPARING, AND AGGREGATING RESULTS -	
WRITING -	GENERATING NEW TEXT BASED ON INSTRUCTIONS -	ASSISTING IN IMPROVING OWN CONTENT -
	PARAPHRASING RELATED WORK -	PUTTING OTHER WORKS IN PERSPECTIVE -
PRESENTATION Claude Sonnet 4.5/4.6	GENERATING NEW ARTIFACTS -.	IMPROVING THE AESTHETICS OF ARTIFACTS All of the tables, diagrams, special blocks (definition, theorem, etc.).
	FINDING RELATIONS BETWEEN OWN OR RELATED ARTIFACTS -	
CODING GitHub Copilot, Claude Sonnet 4.5/4.6, Claude Opus 4.5/4.6	GENERATING NEW CODE BASED ON DESCRIPTIONS OR EXISTING CODE Auto-completion throughout development.	REFACTORING AND OPTIMIZING EXISTING CODE Guided refactoring (not performance-based) on essentially the entire relevant part of the existing codebase.

COMPARING ASPECTS OF EXISTING CODE
Bug finding and monitored fixing throughout development.

DATA
-

SUGGESTING NEW SOURCES FOR DATA COLLECTION
-

CLEANING, NORMALIZING, OR STANDARDIZING DATA
-

FINDING RELATIONS BETWEEN DATA AND COLLECTION METHODS
-

ETHICS

WHAT ARE THE IMPLICATIONS OF USING AI FOR THIS PROJECT?
The only certain harm as a result of our use of AI is the environmental harm as a consequence of running AI farms. A potential harm as a result of our use of AI is propagating the work of others without permission (through the model).

WHAT STEPS ARE WE TAKING TO MITIGATE ERRORS OF AI FOR THIS PROJECT?
All literature mentioned in this thesis was verified to exist and checked for relevance. Verified claims were only accepted if the AI verdict matched our own memory or interpretation, and were manually verified otherwise. Generated and adjusted code was visually checked as well as manually tested to ensure correctness.

WHAT STEPS ARE WE TAKING TO MINIMIZE THE CHANCE OF HARM OR INAPPROPRIATE USE OF AI FOR THIS PROJECT?
We tried to minimize environmental harm by doing much work by hand still, and by using clear, concise, and complete prompts. Unauthorized use of data is impossible to avoid when using AI, as one can never reach out to authors of all works in the training data. If this harm exists, we believe it is very minor, especially given the limited way we used AI.

THE CORRESPONDING AUTHORS VERIFY AND AGREE WITH THE MODIFICATIONS OR GENERATIONS OF THEIR USED AI-GENERATED CONTENT
The corresponding author (Marijn Meuleman) attests to the correctness and appropriateness of all AI-generated content in this thesis.

AI Usage Card v1.0 `https://ai-cards.org` PDF | BibTeX | XML | JSON | CSV